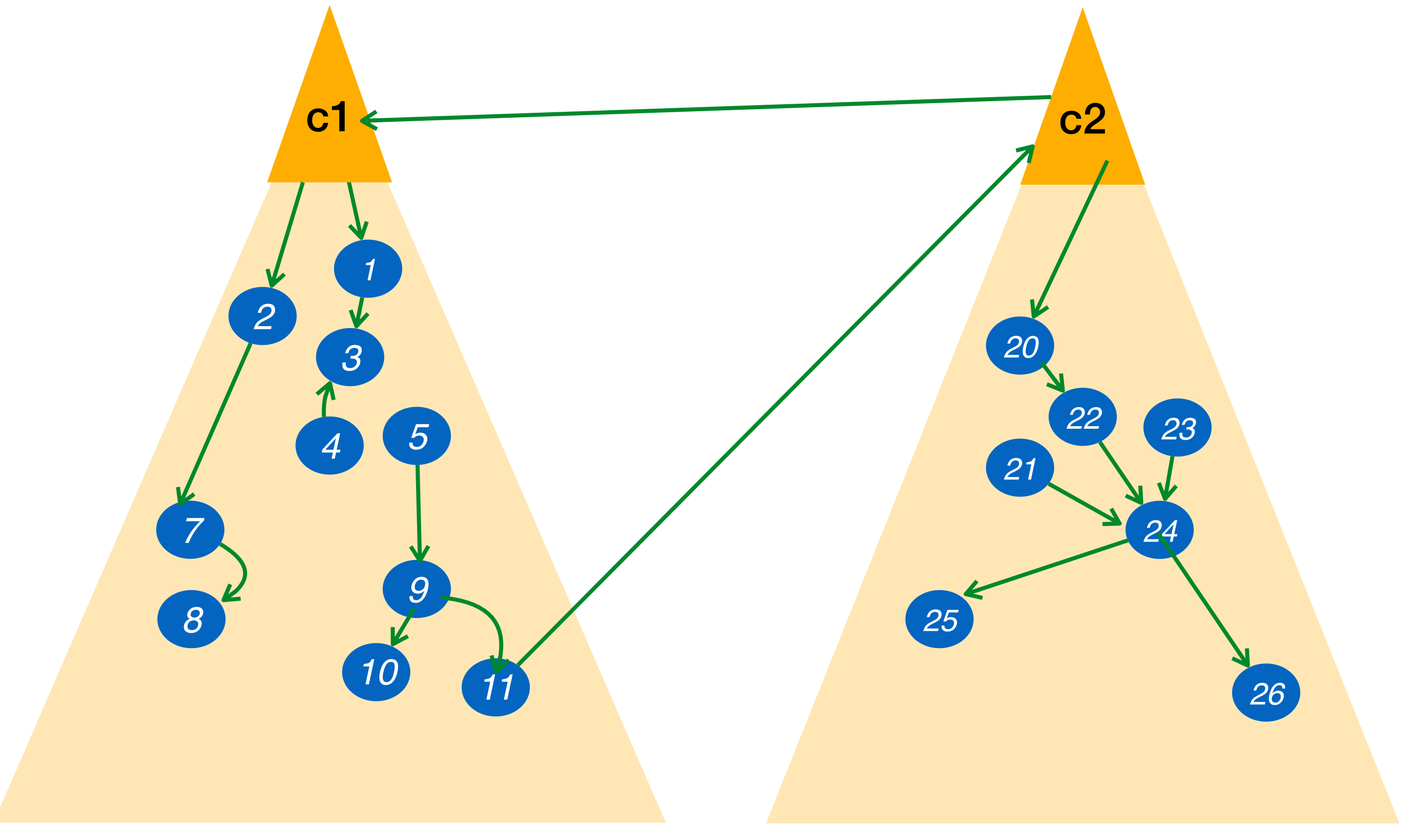
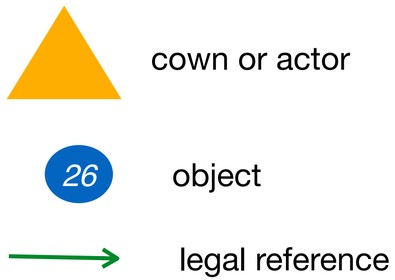
Putting Separation to work

In Pony and in Verona

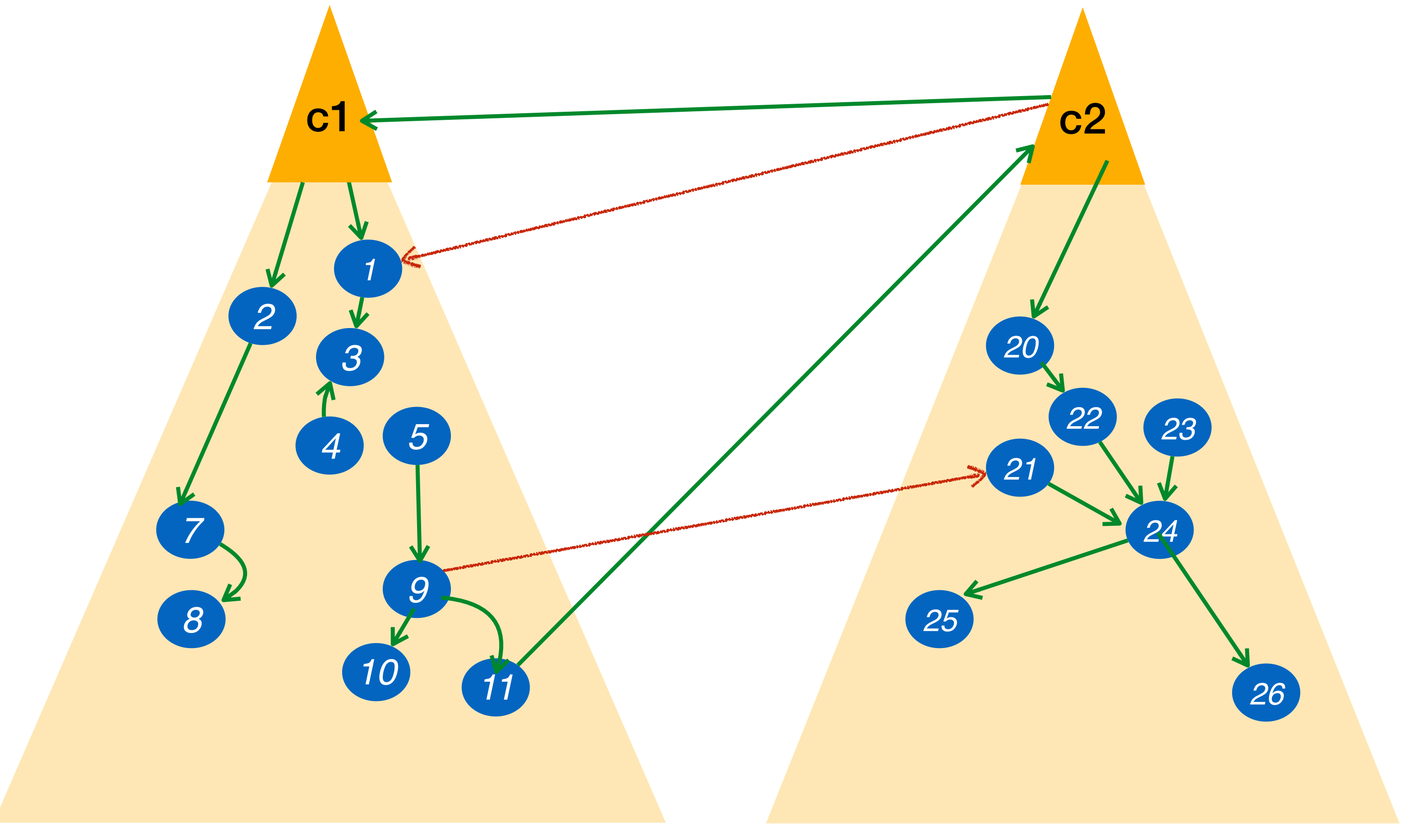
Sophia Drossopoulou and Tobias Wrigstad, PLISS 2023

Separation


c1
c2
1
2
3
4
5
7
8
9
10
11
20
21
22
23
24
25
26


cown or actor
26
object
legal reference

Separation



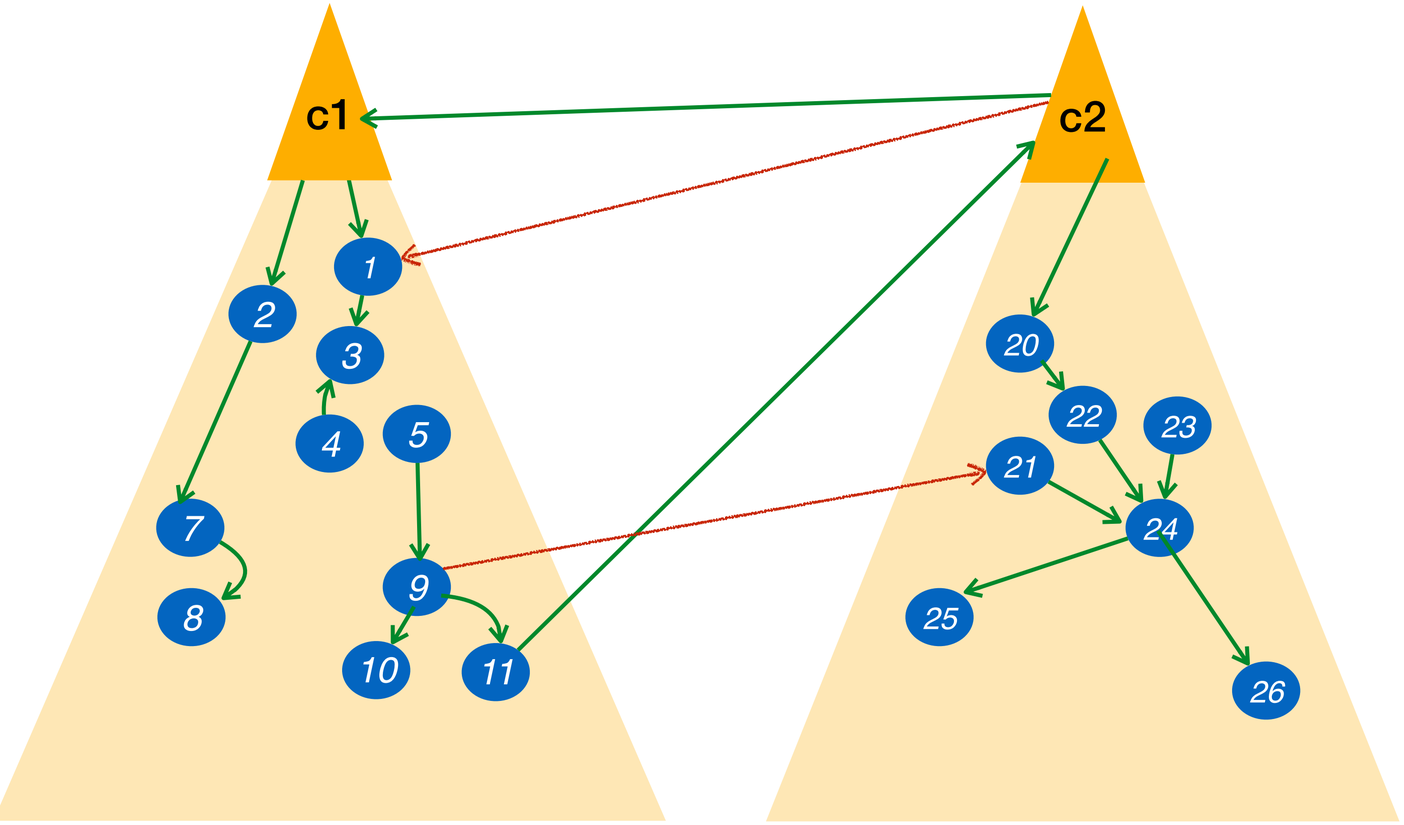

cown or actor

object

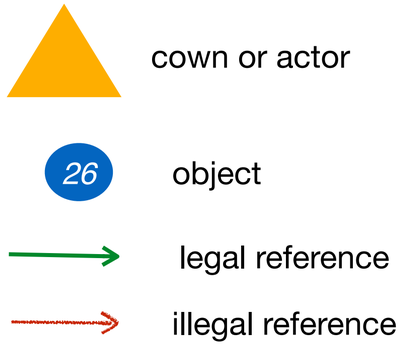

legal reference


illegal reference

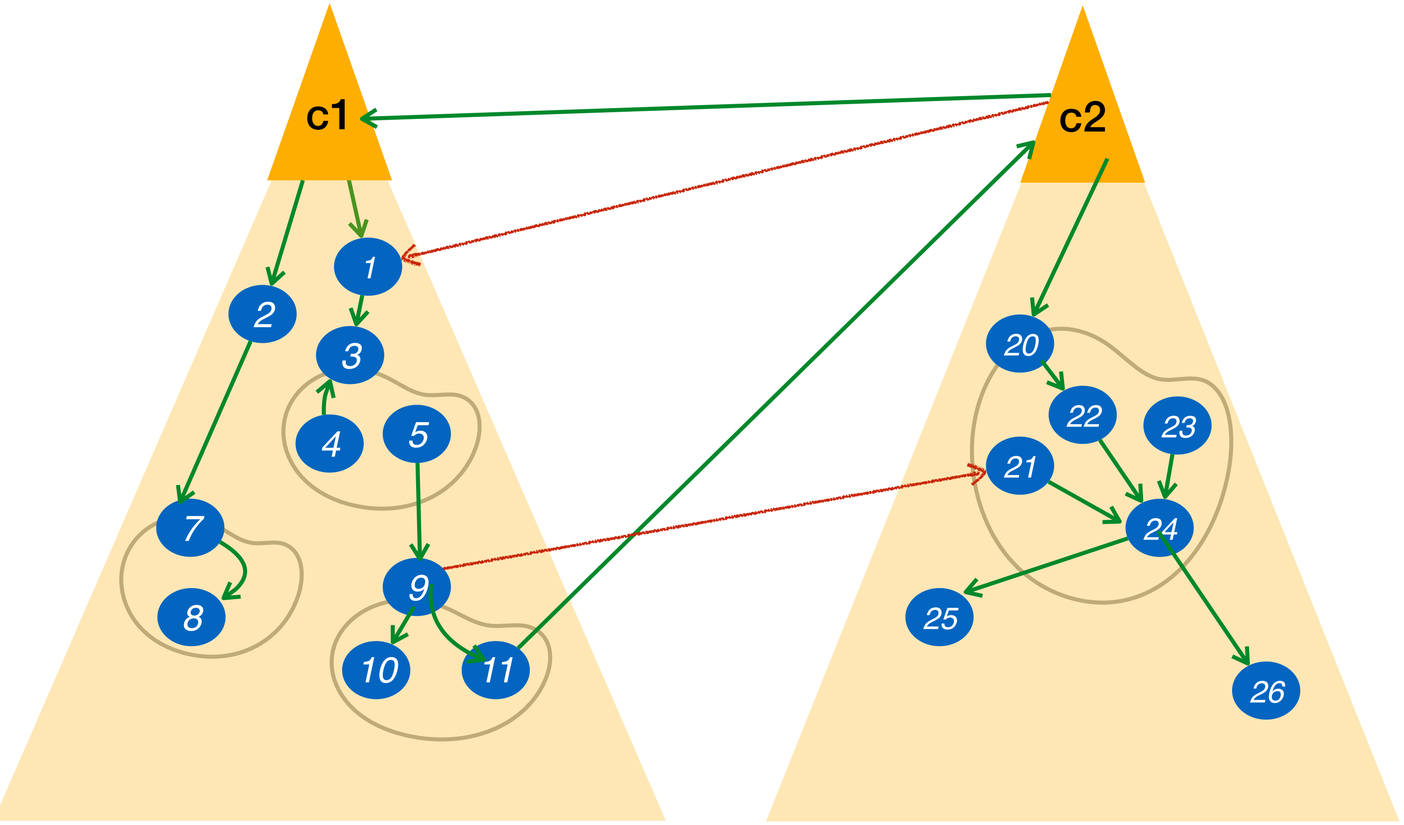
Separation ... and its remit



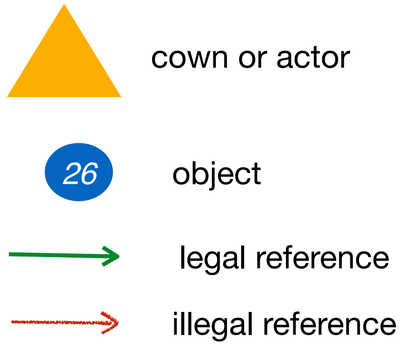
- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



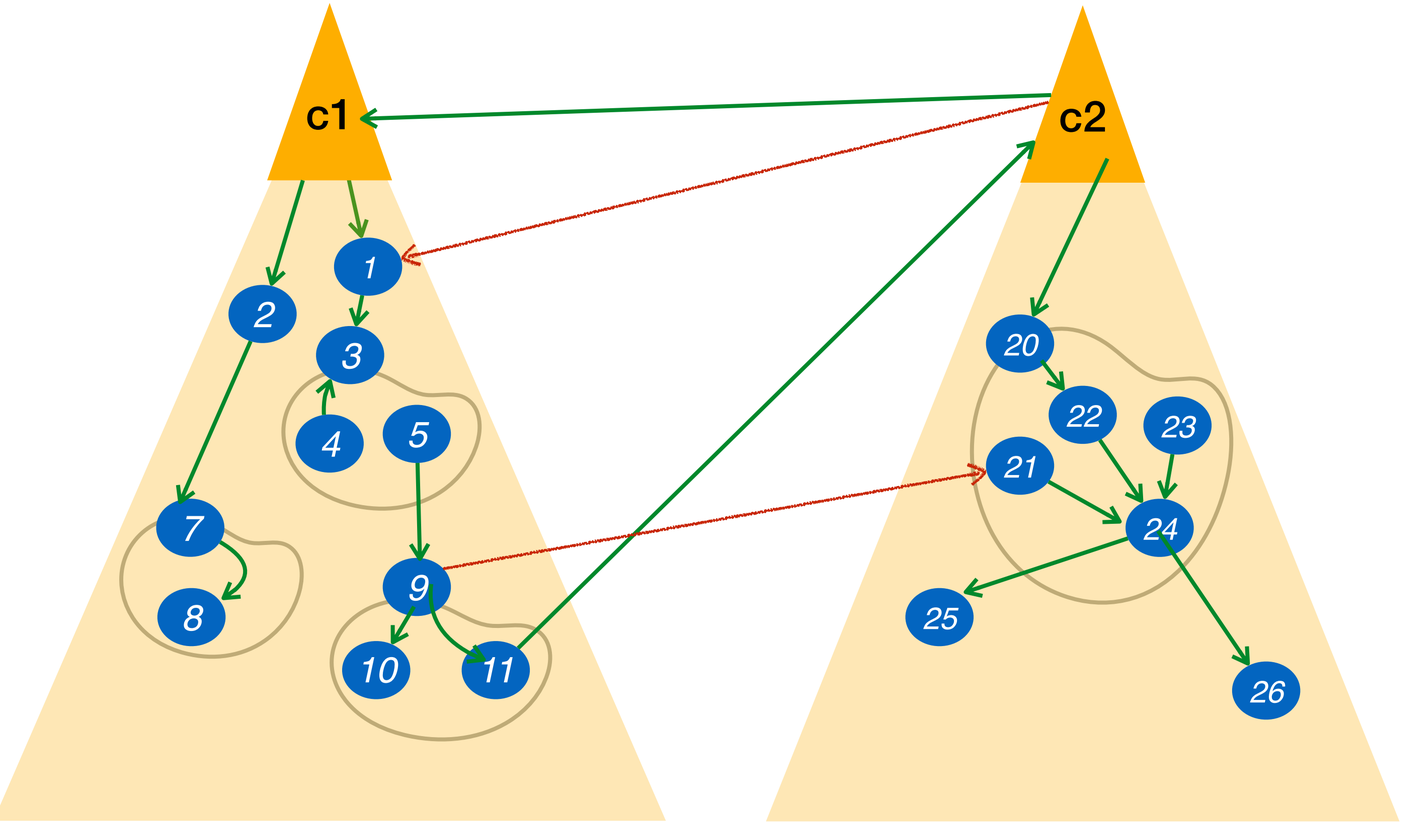
Finer-grained Separation — with Regions



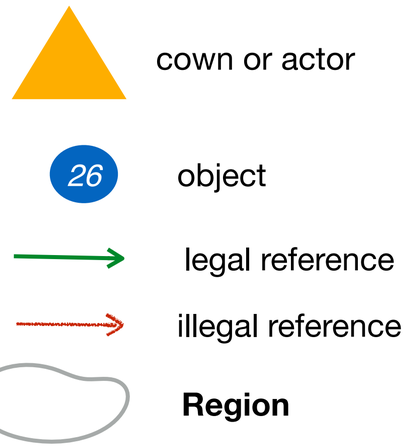
- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



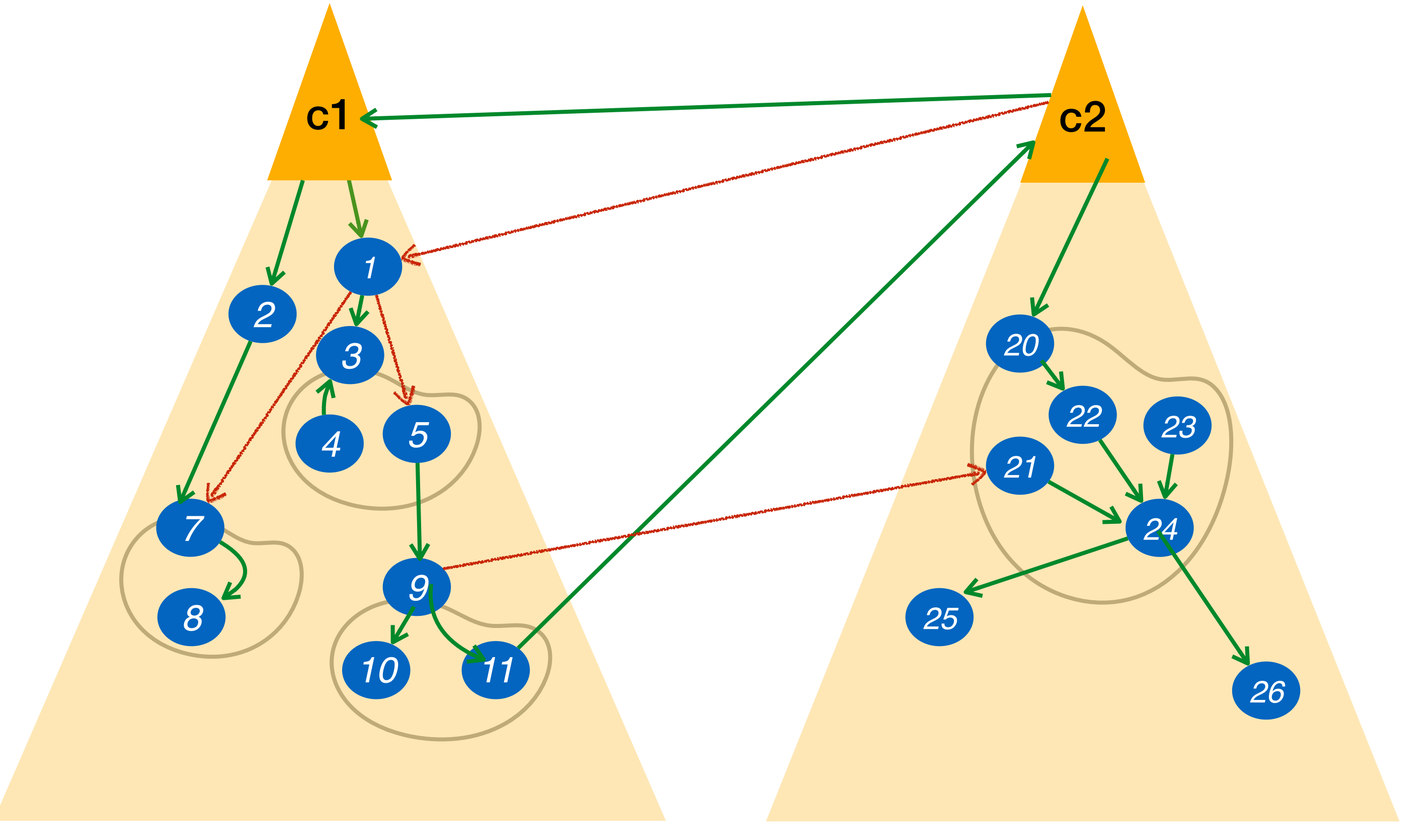
Finer-grained Separation — with Regions



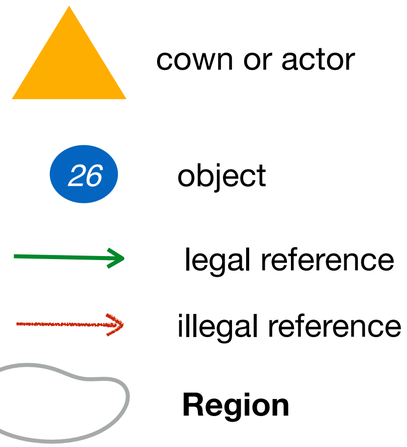
- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



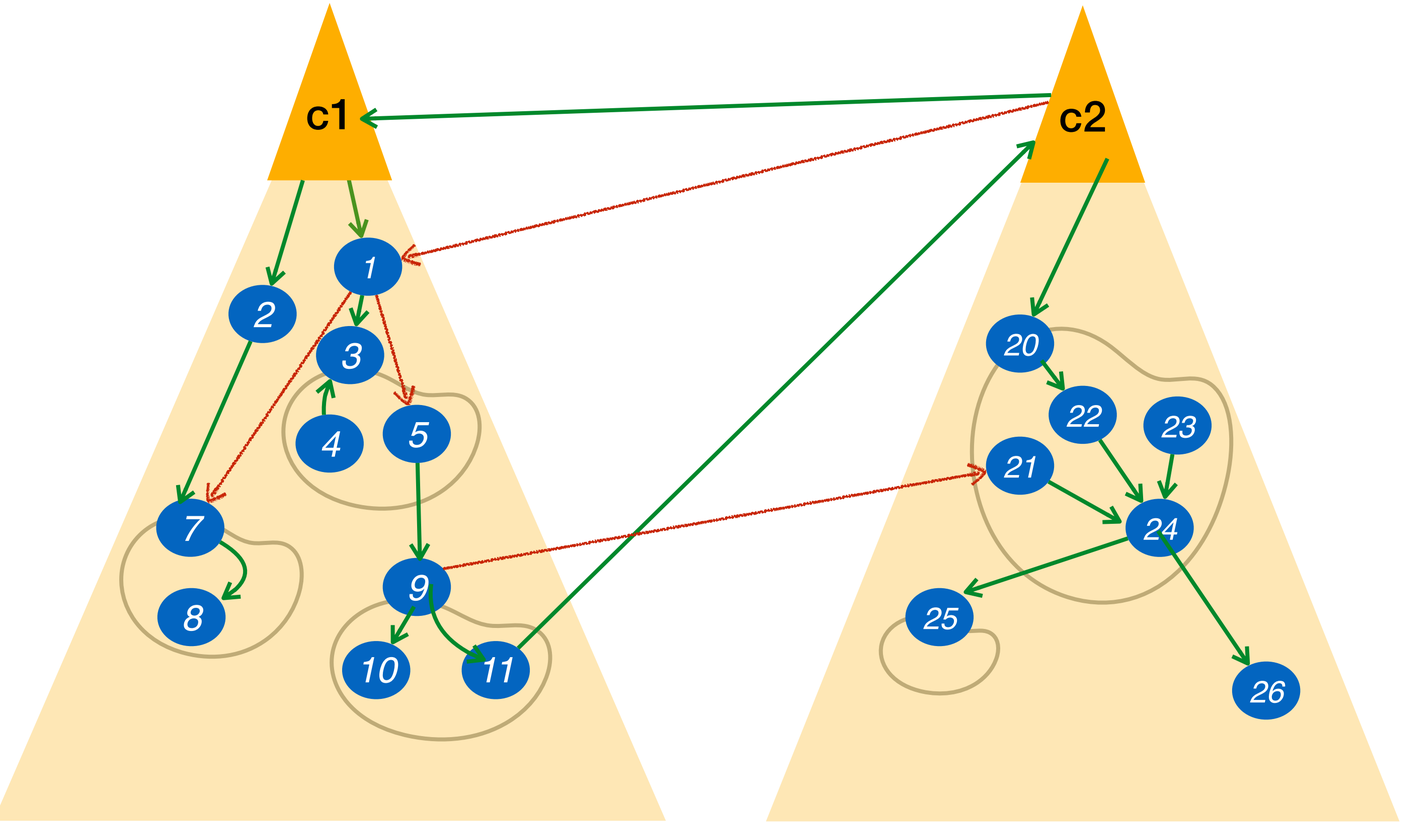
Finer-grained Separation — with Regions



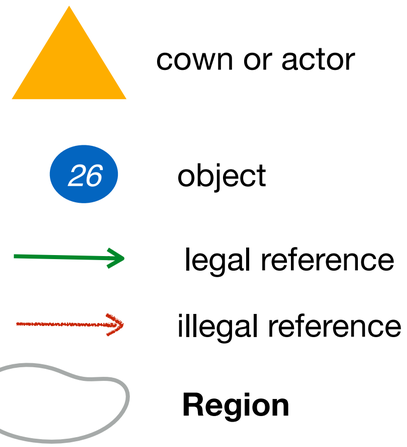
- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



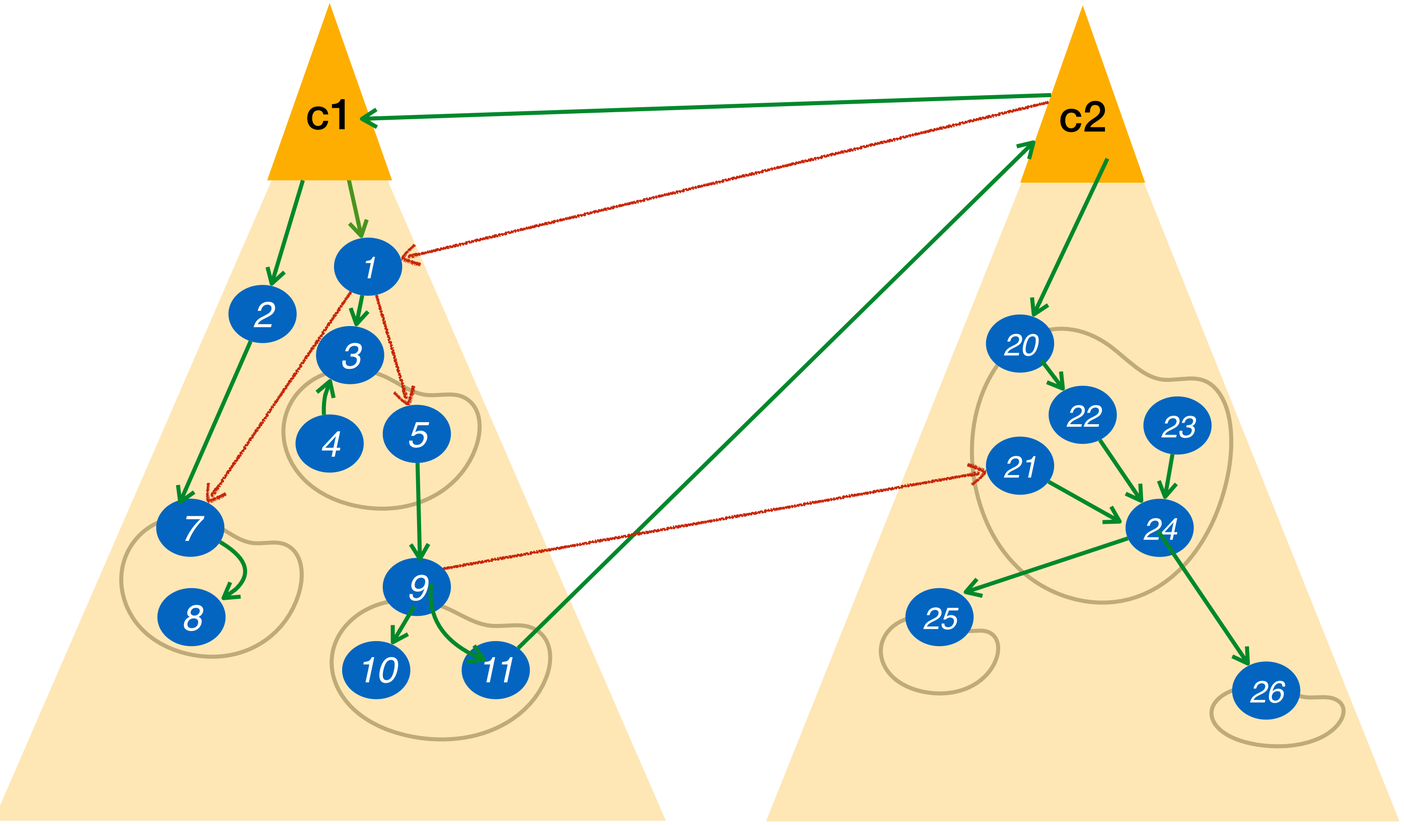
Finer-grained Separation — with Regions



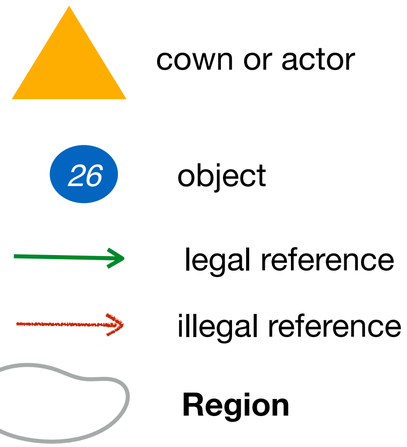
- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



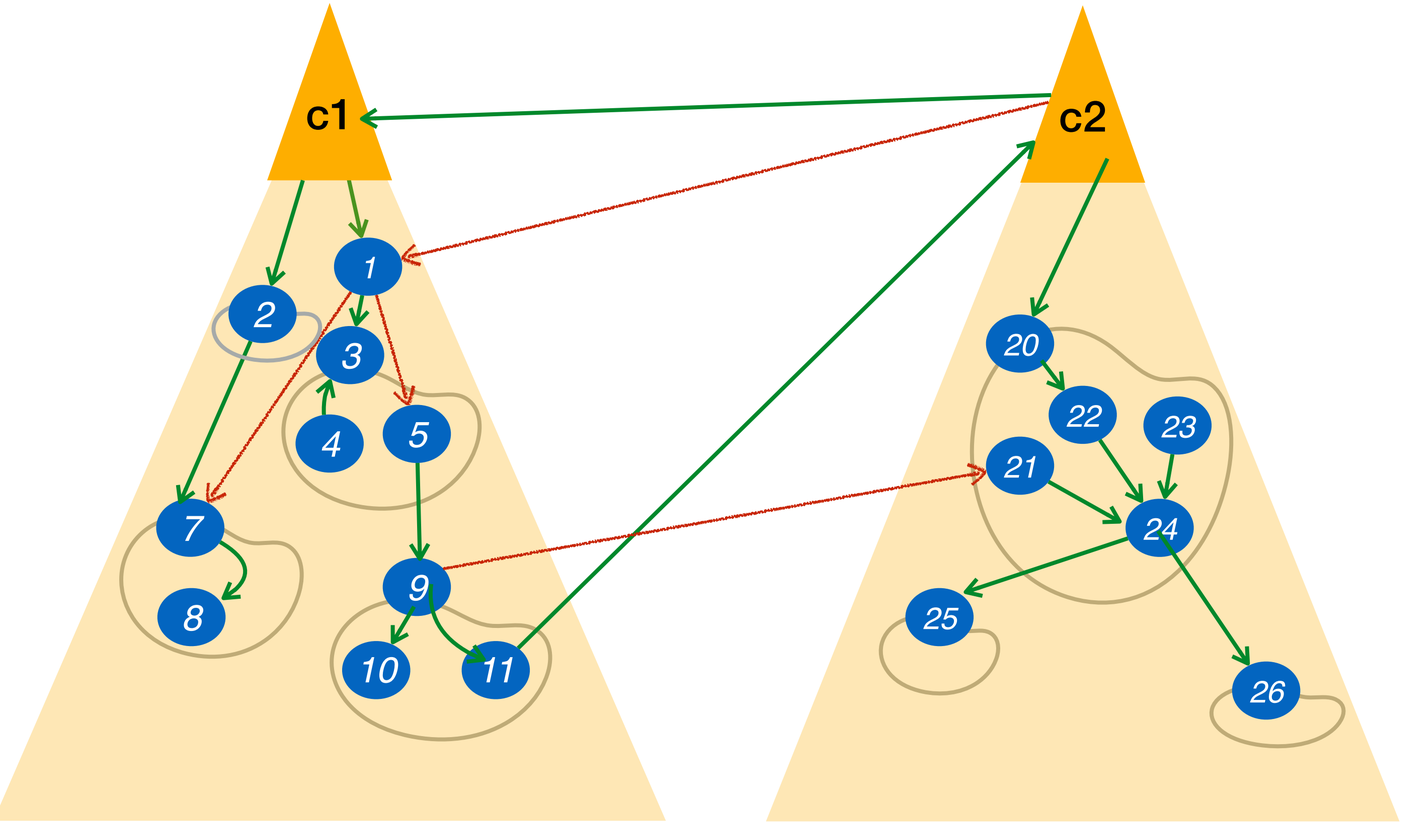
Finer-grained Separation — with Regions



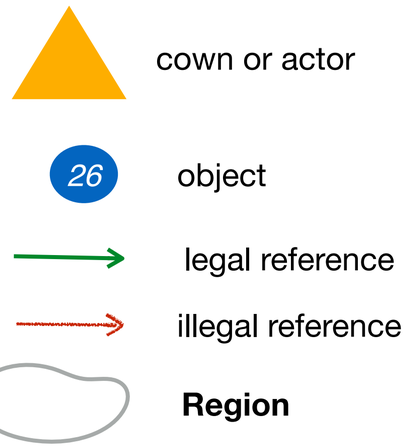
- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



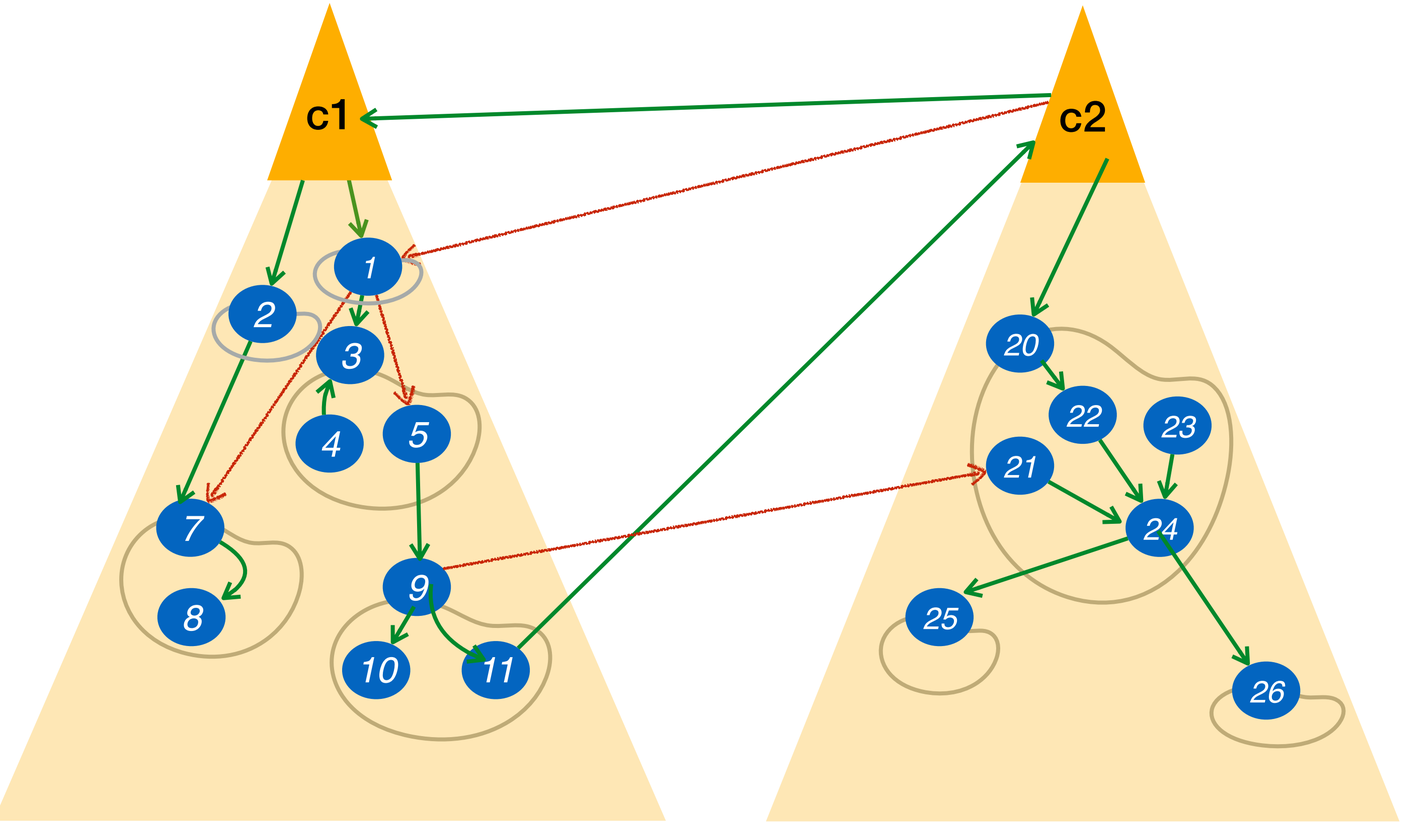
Finer-grained Separation — with Regions



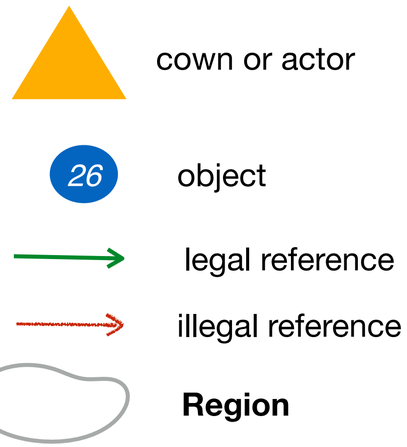
- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



Finer-grained Separation — with Regions



- Data-race freedom
- Share state safely without copy
- Garbage collection
- Causality



Actor-based programming (Pony)

Summary

- actor ~ active object (state)
- actors send asynchronous messages to other actors (**be**haviours)
- messages stored in queues; when scheduled, actor executes first behaviour from its message queue
- actors send synchronous messages to objects, or to themselves (**fun**ctions)

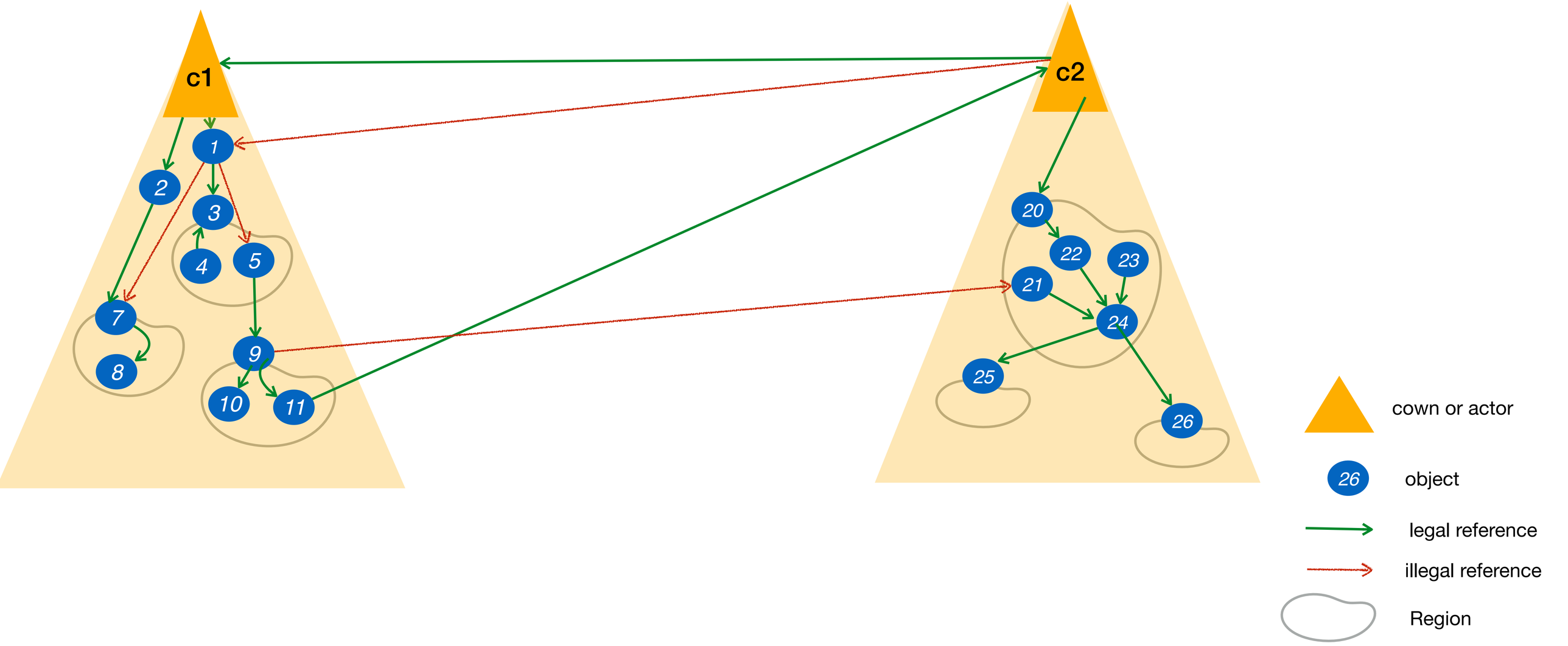
Behaviour oriented programming (Verona)

Summary

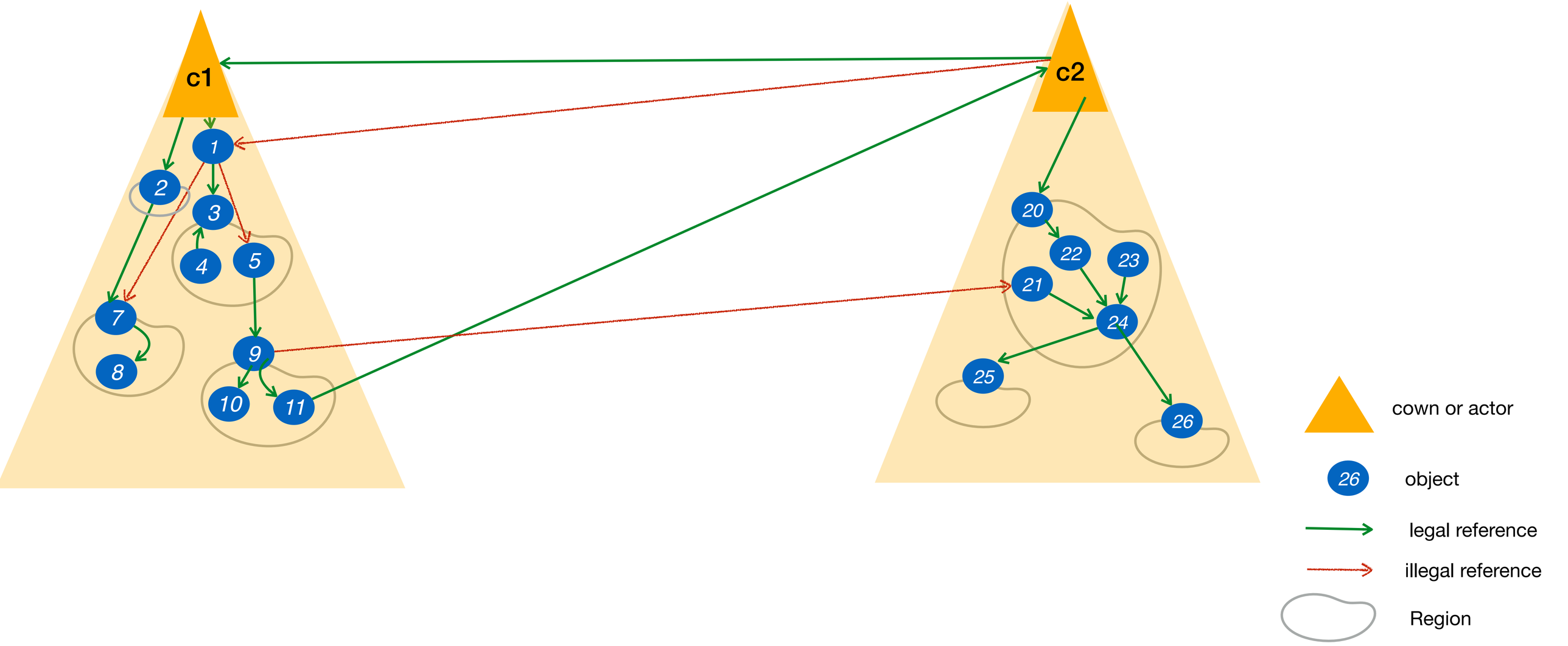
- Cowns protect separated state
- Behaviours require cowns;
  - acquire them atomically at the beginning of execution
  - release them atomically at the end of execution
- During its execution, behaviour has (unique) access to the state protected by the cowns it has acquired

Data race freedom

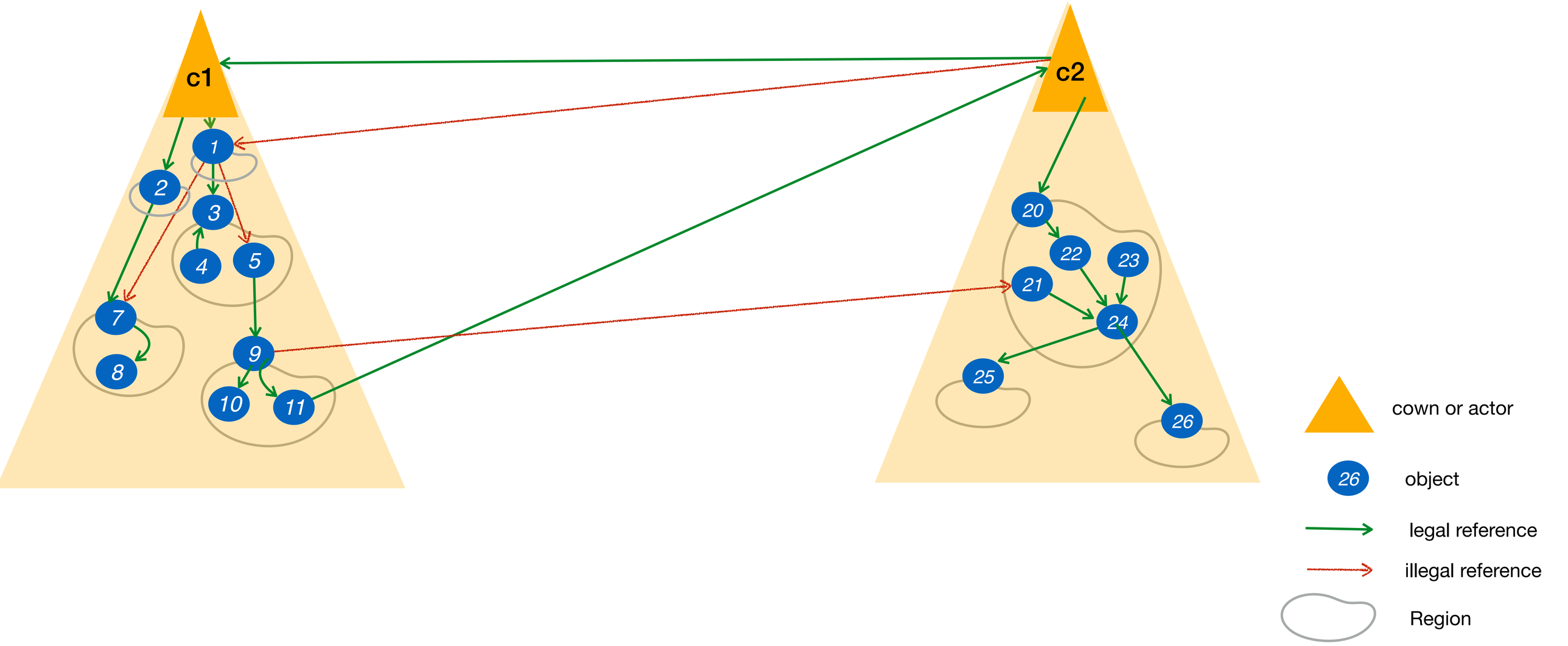
Data-race freedom Actors/Behaviours (and more later)



Data-race freedom Actors/Behaviours (and more later)

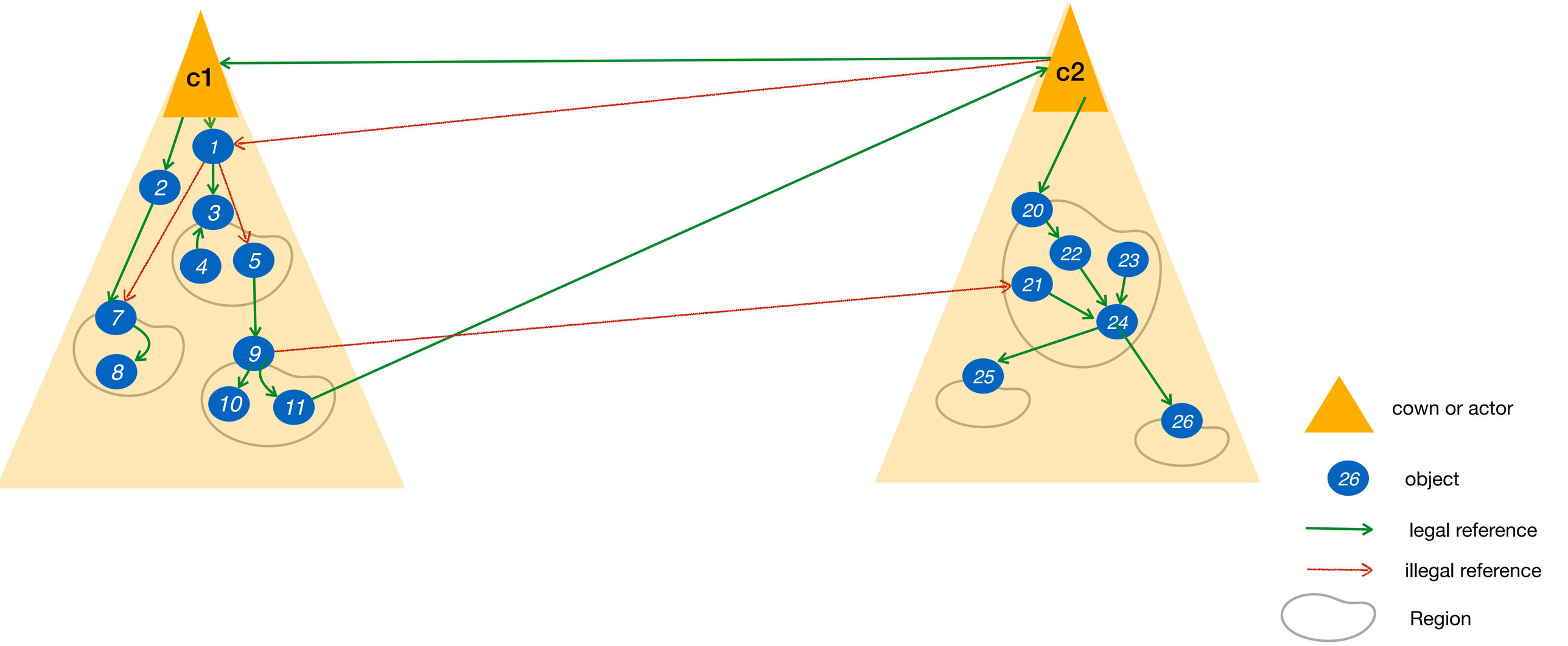


Data-race freedom Actors/Behaviours (and more later)

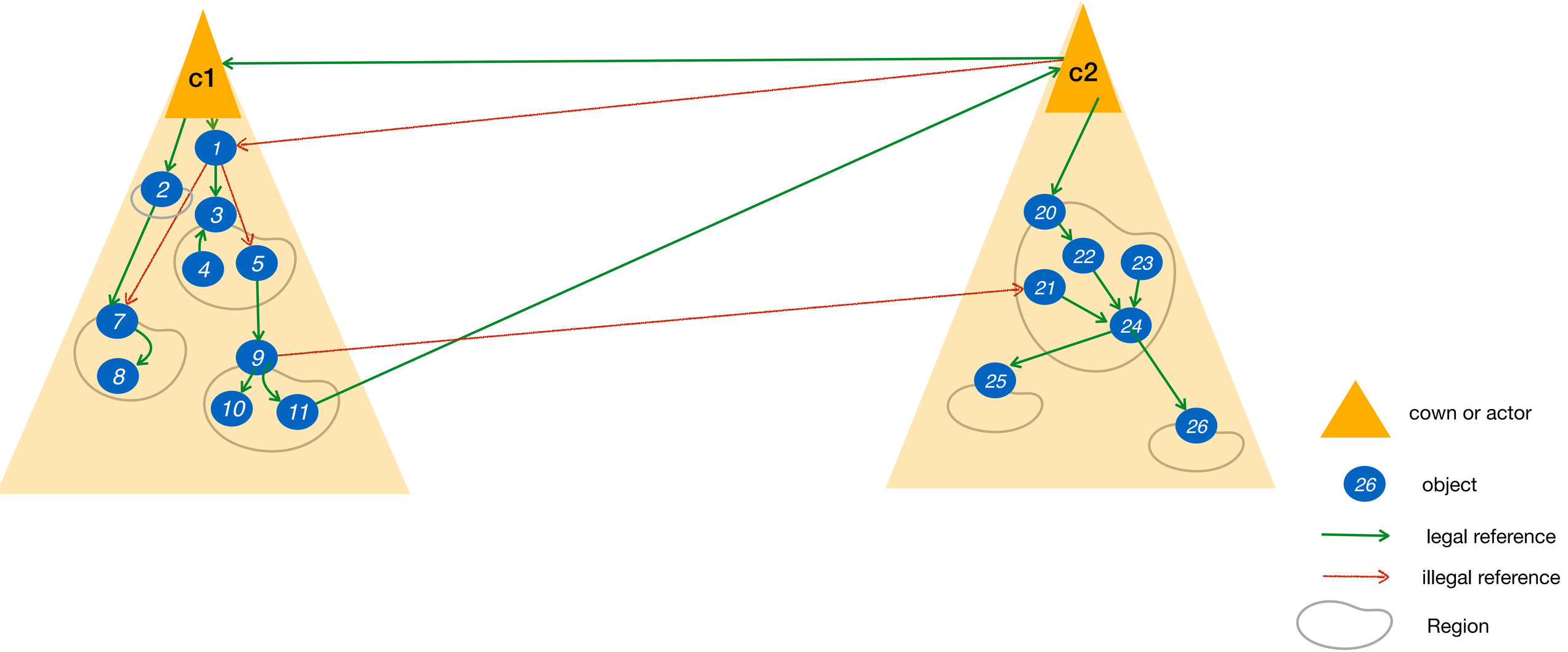


Safe share of state without copy

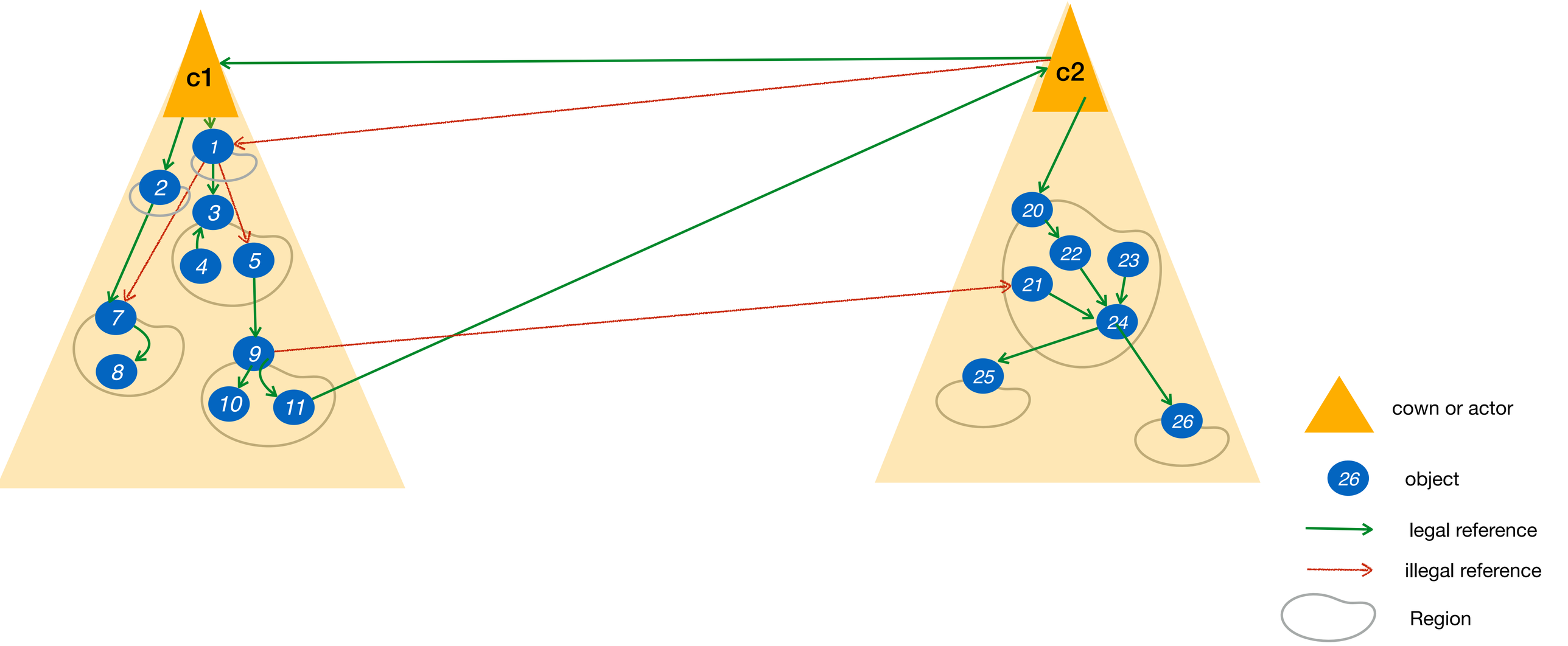
Pony actors share state safely - 1



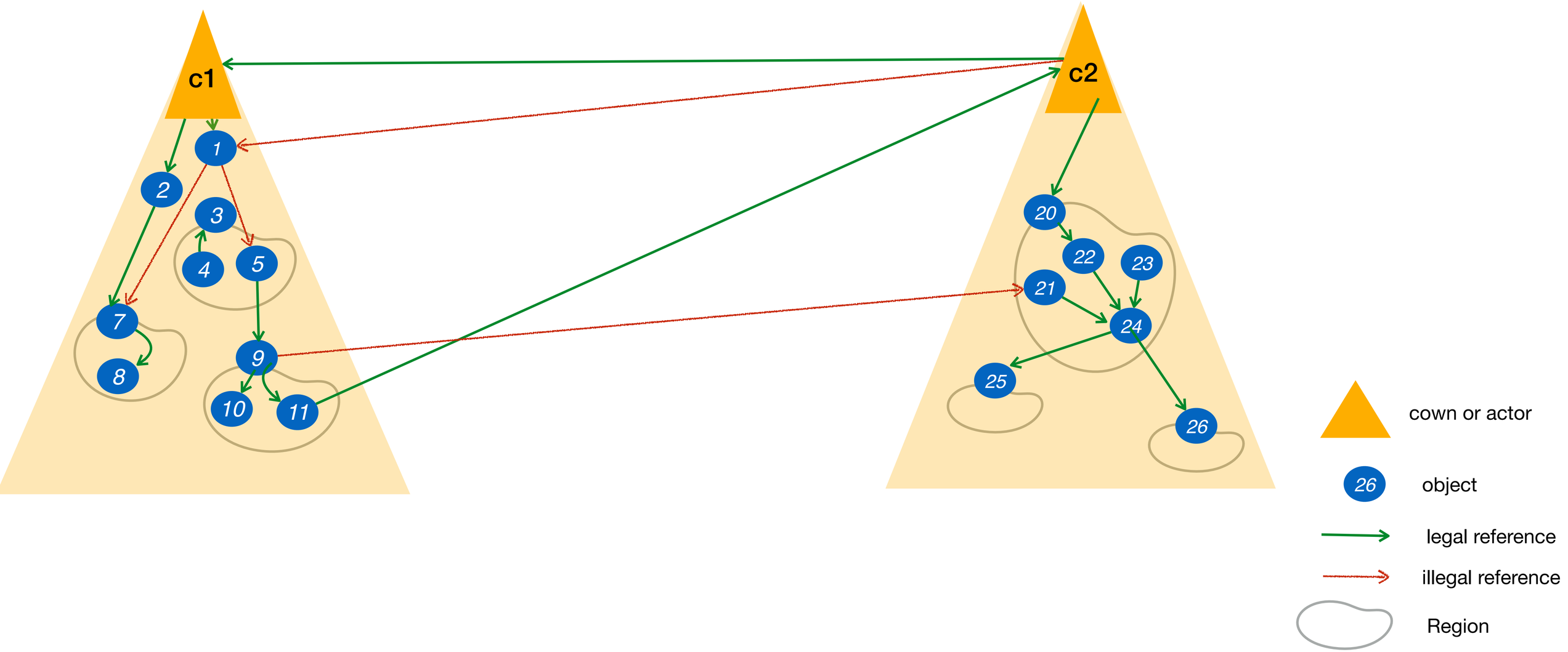
Pony actors share state safely - 1



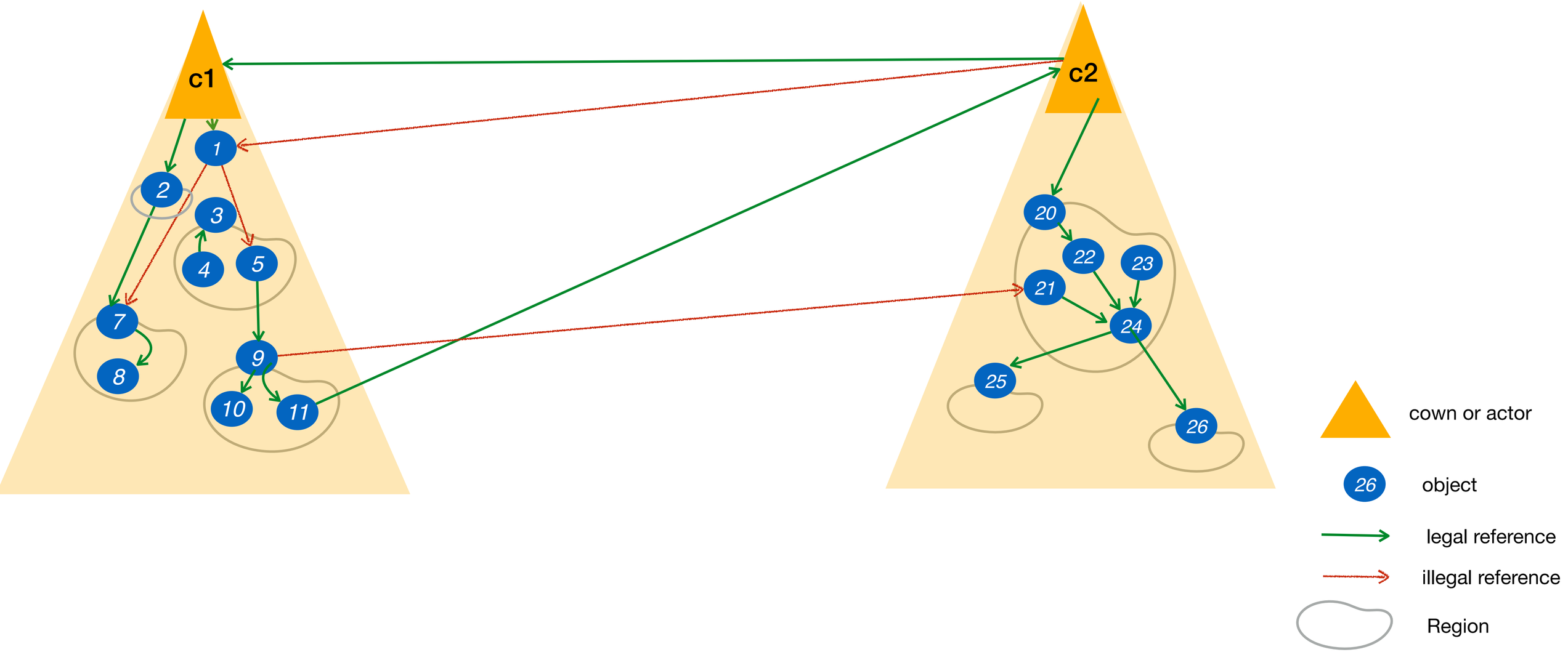
Pony actors share state safely - 1



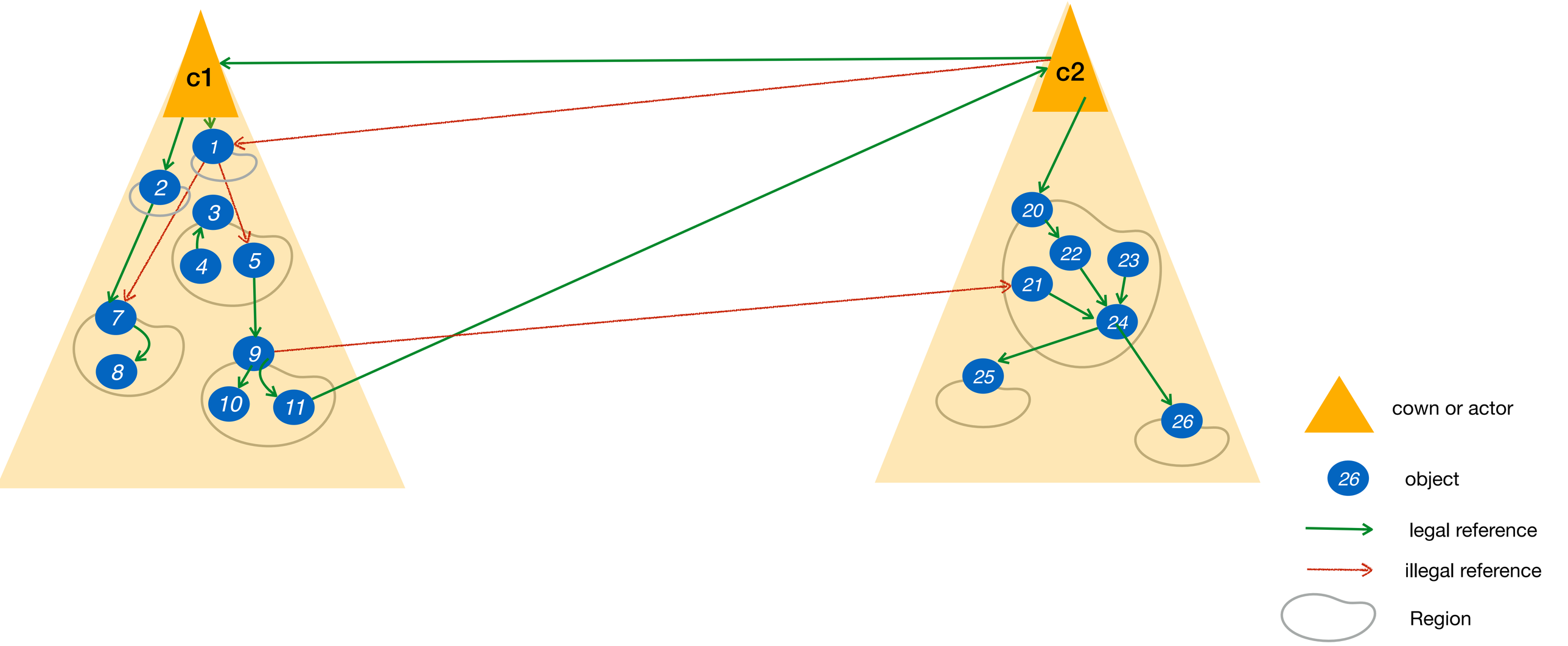
Pony actors share state safely - 2



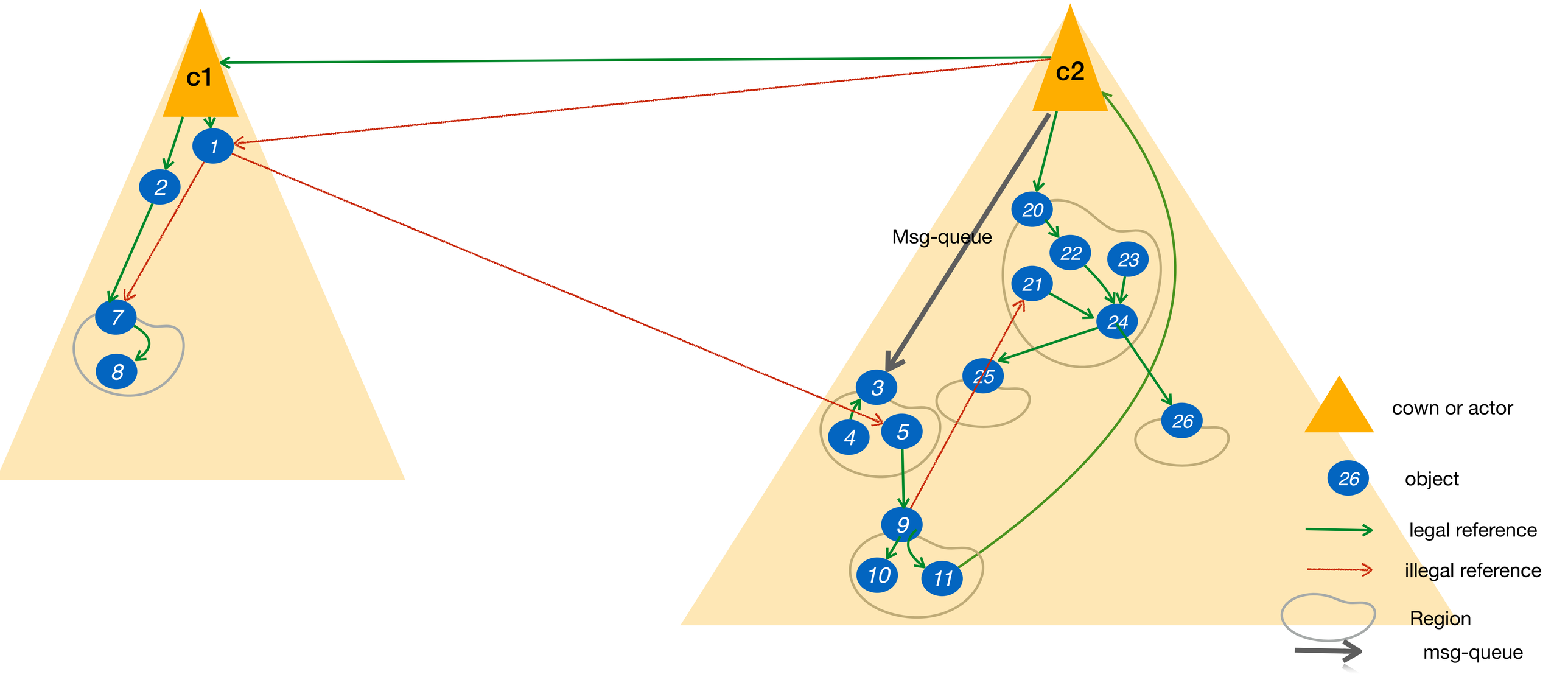
Pony actors share state safely - 2



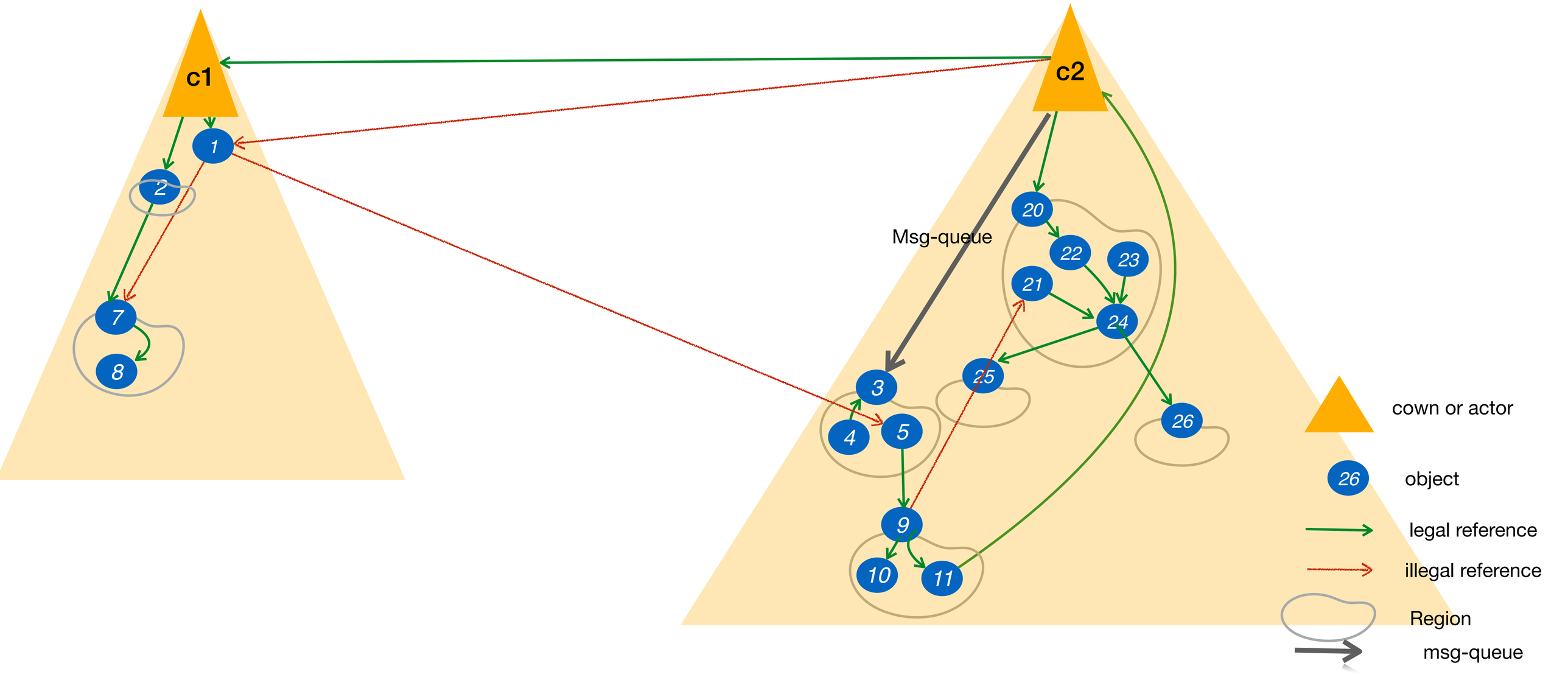
Pony actors share state safely - 2



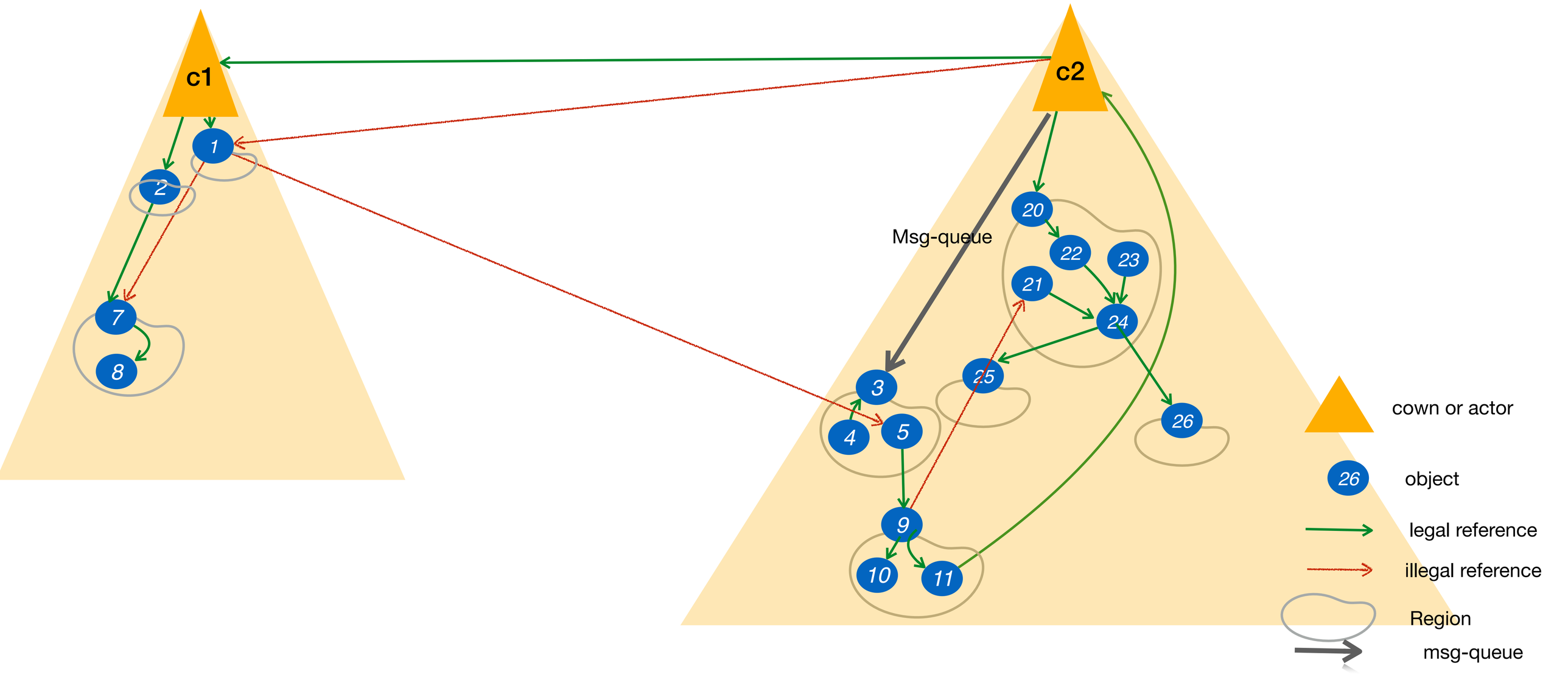
Pony actors share state safely - 3



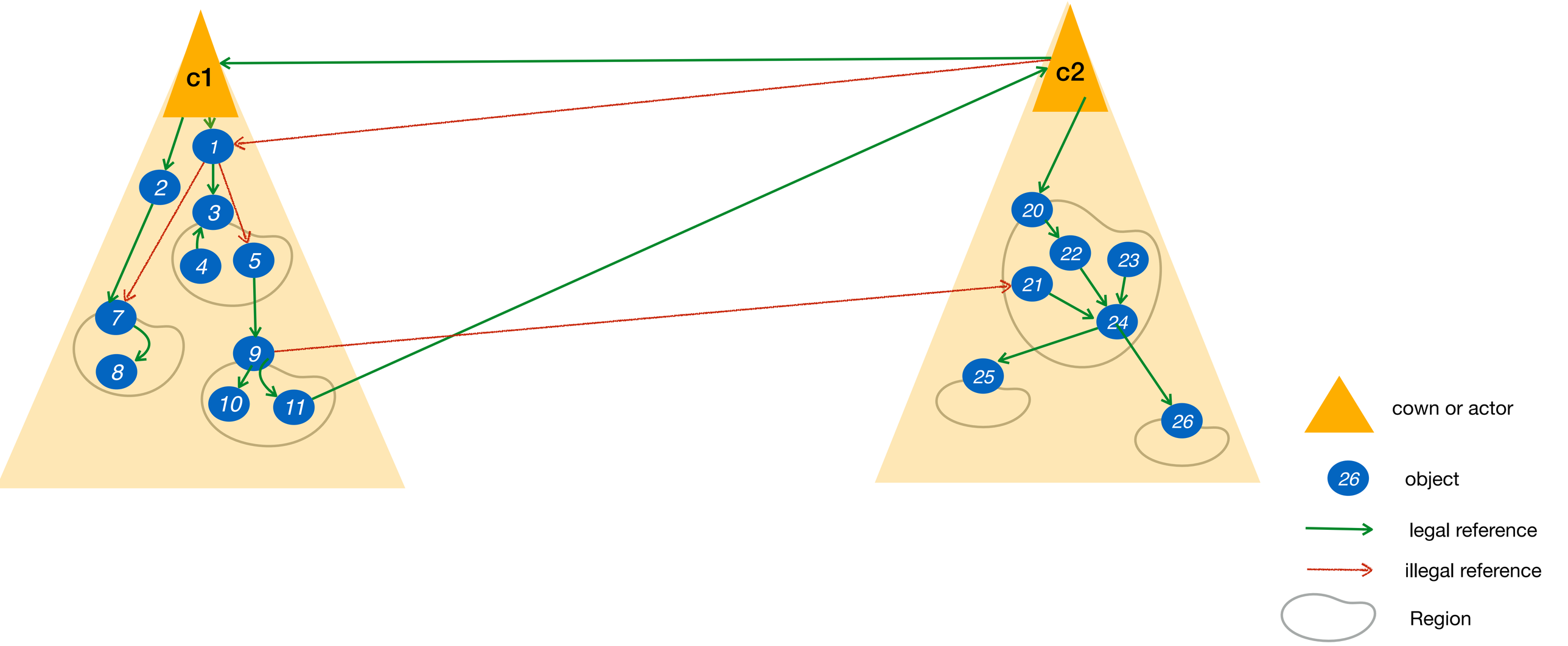
Pony actors share state safely - 3



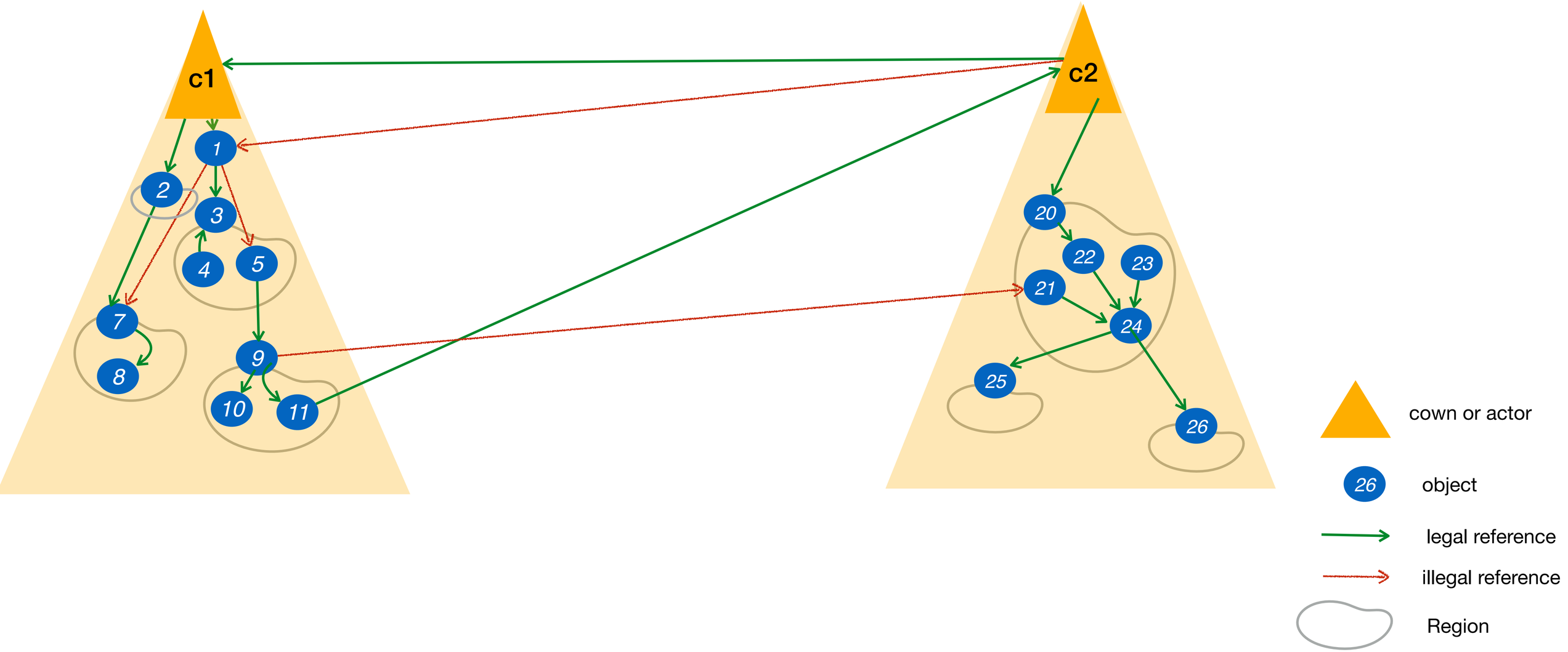
Pony actors share state safely - 3



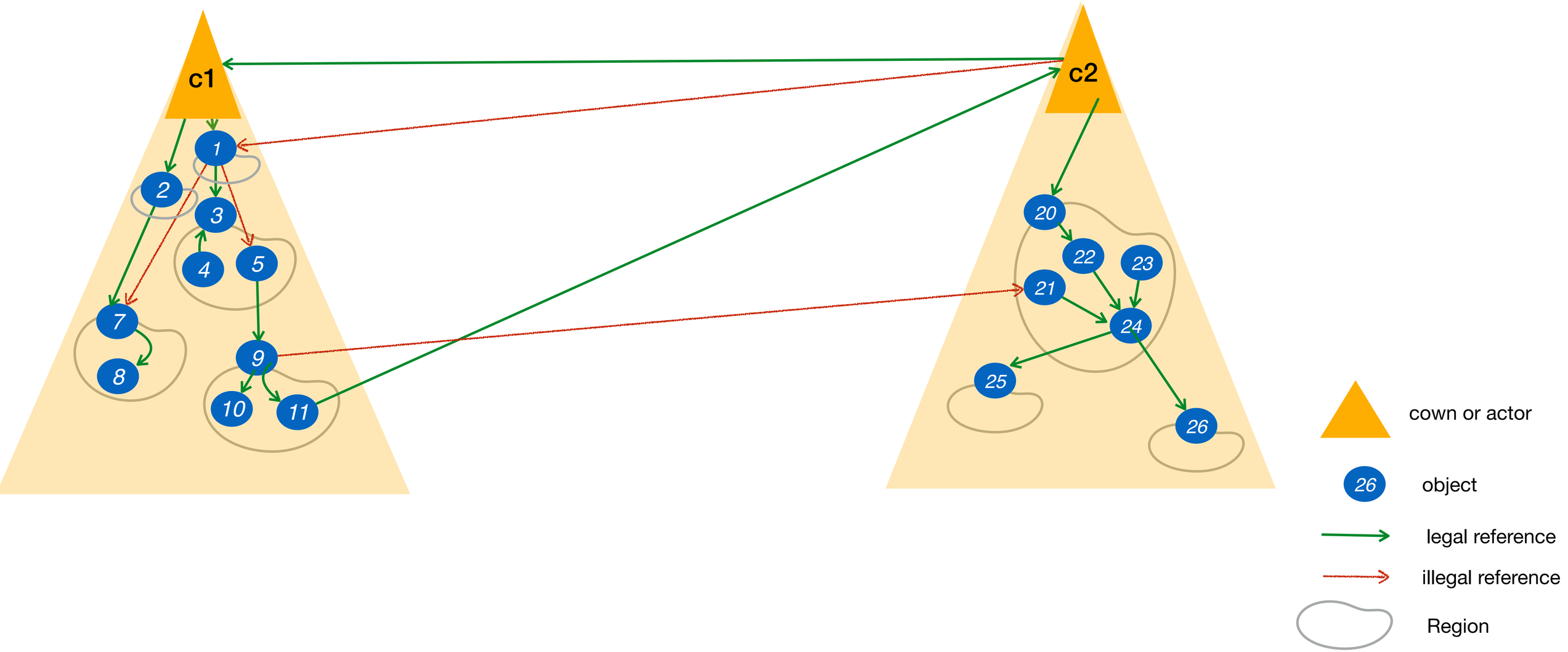
Cowns share state safely - 1



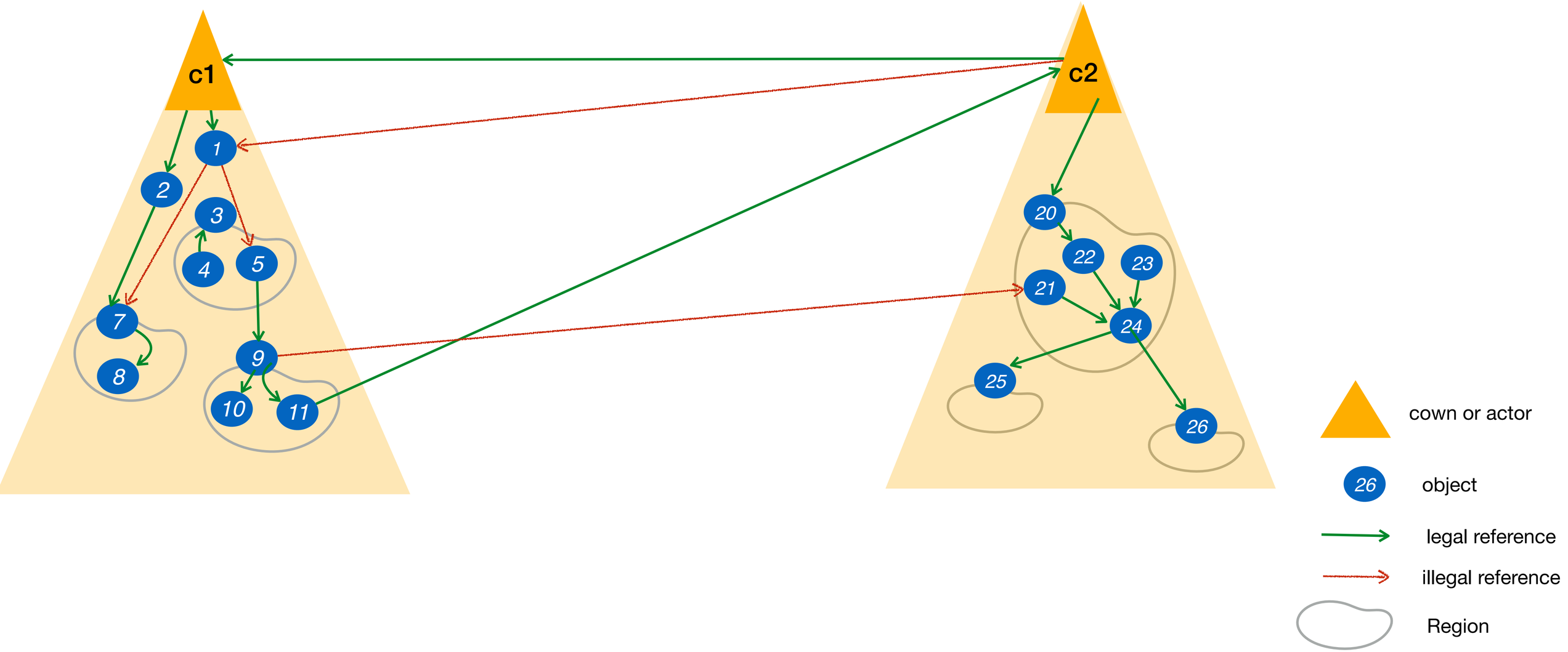
Cowns share state safely - 1



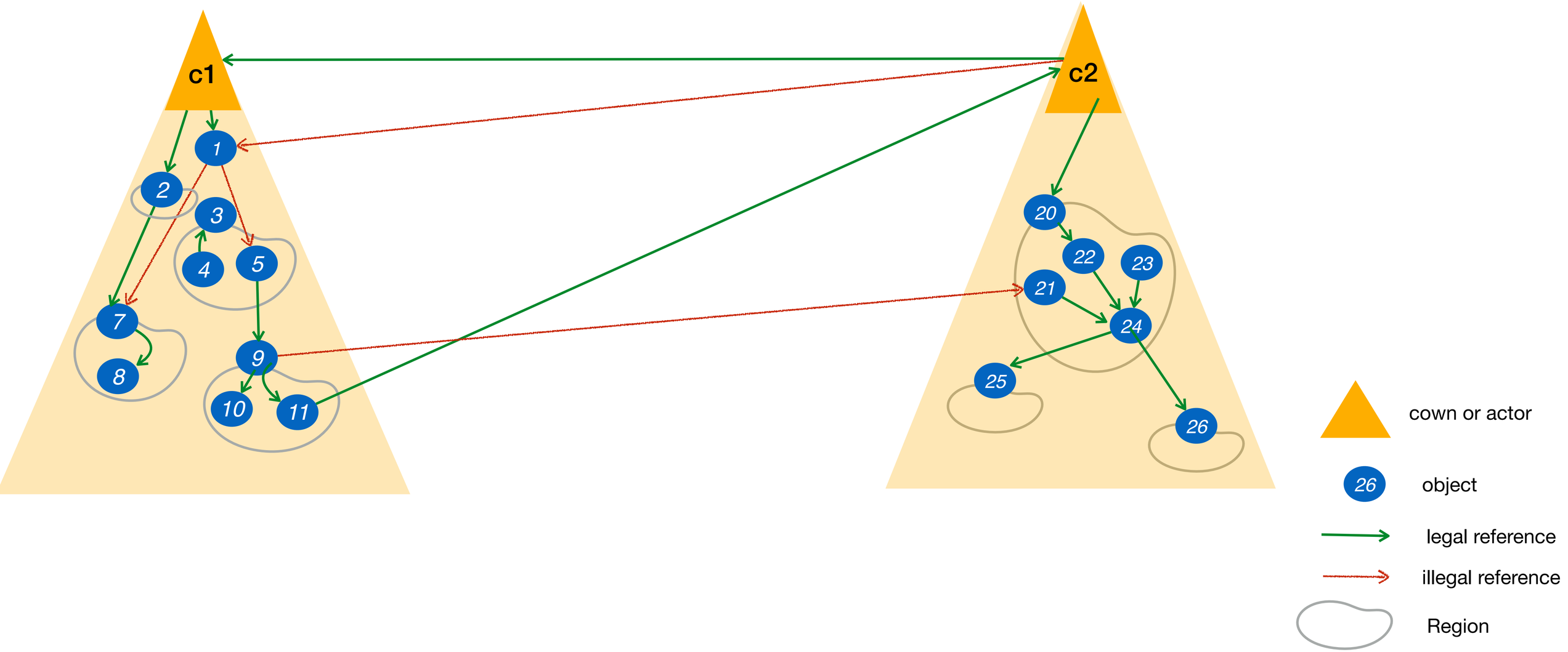
Cowns share state safely - 1



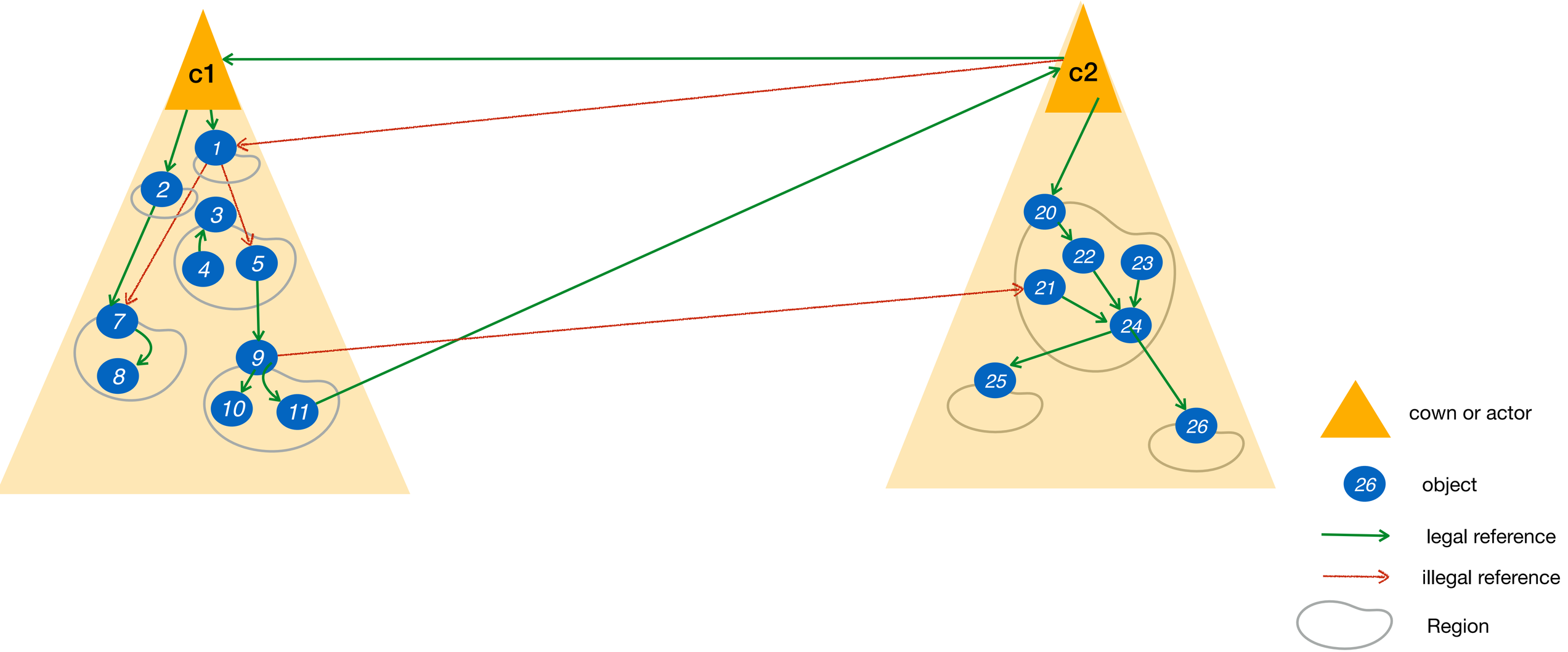
Cowns share state safely - 2



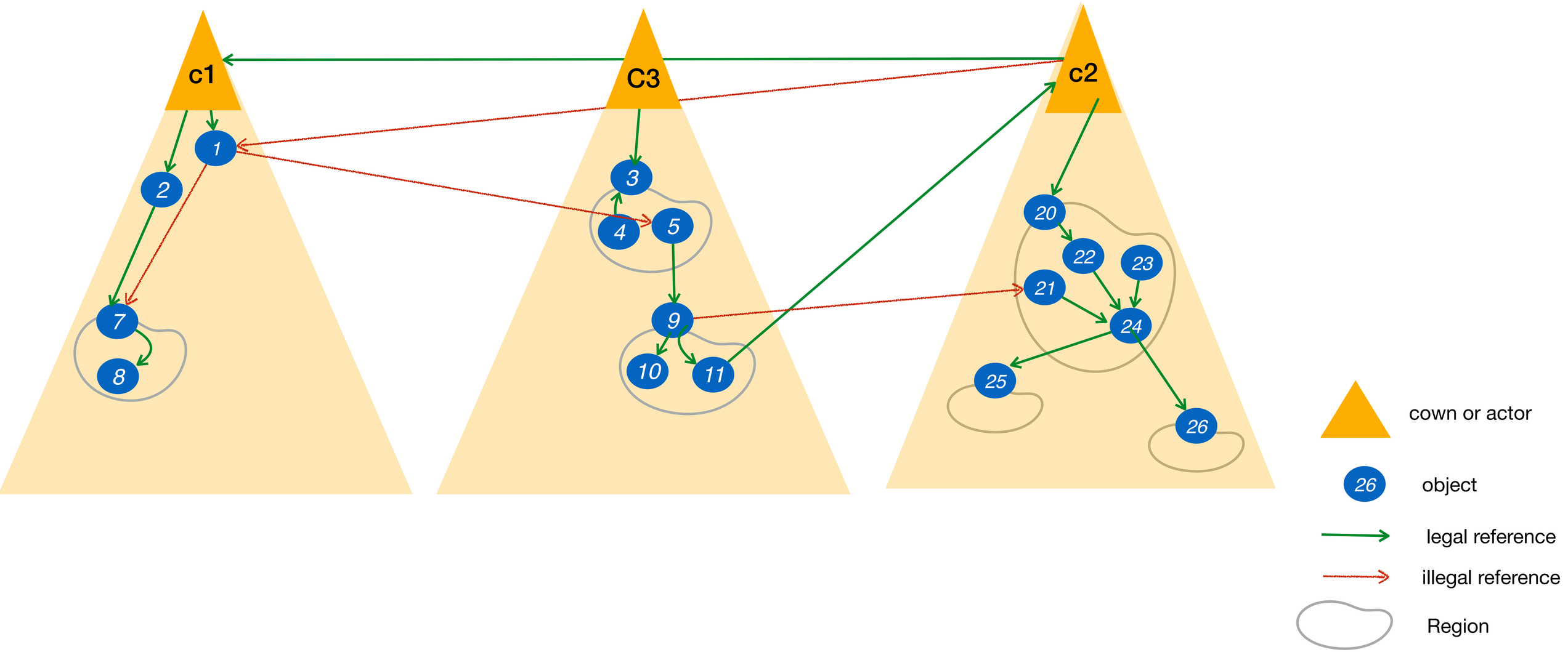
Cowns share state safely - 2



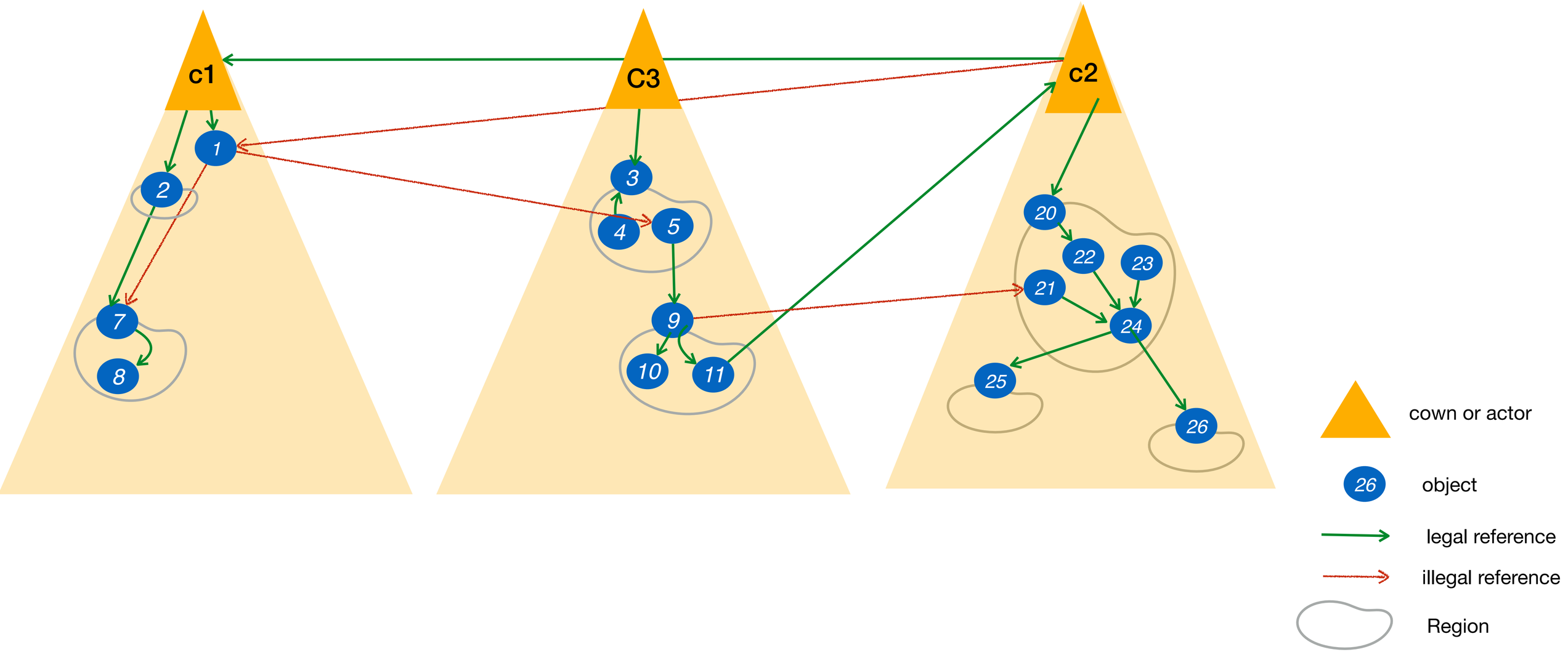
Cowns share state safely - 2



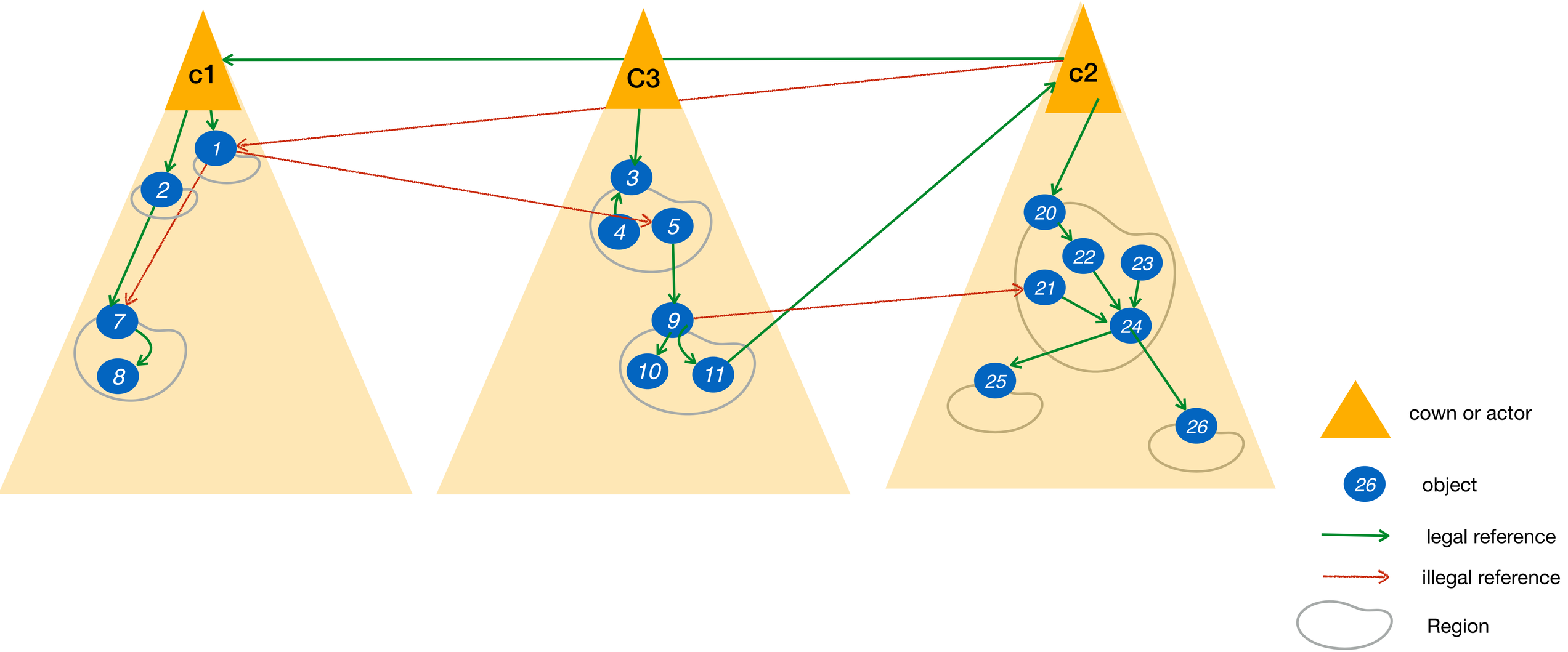
Cowns share state safely - 3



Cowns share state safely - 3



Cowns share state safely - 3



Garbage Collection

MAC: Multicore Actor Collector

GC for actors (Pony)

Fully Concurrent Garbage Collection of Actors on Many-Core Machines

Sylvan Clebsch and Sophia Drossopoulou
Department of Computing, Imperial College, London
{sc5511, scd}@doc.ic.ac.uk

What is a dead actor?

$$\text{Idle}(a) \quad \text{iff} \quad (\text{a is not executing a behaviour}) \ \wedge \ (\text{a.MsgQueue is empty})$$

$$\text{Dead}(a) \quad \text{iff} \quad \text{Idle}(a) \ \wedge \ \forall a'.[\, a \in a'.\text{heap} \implies \text{Dead}(a')\,] \ \wedge \ \forall a'.\,[\, a \notin a'.\text{MsgQueue}\,]$$

What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,]$ $\wedge$ $\forall a'.\,[\, a \notin a'.\text{MsgQueue}\,]$

not idle

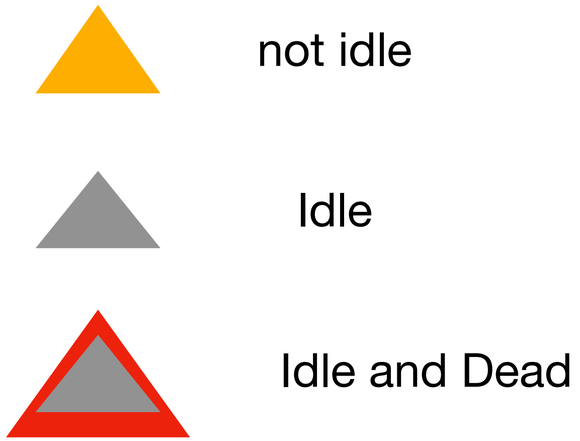
Idle

Idle and Dead

What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,]$ $\wedge$ $\forall a'.\,[\, a \notin a'.\text{MsgQueue}\,]$



What is a dead actor?

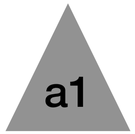
Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'.[a$\in$a'.heap $\Longrightarrow$ Dead(a')] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

not idle

Idle

Idle and Dead



What is a dead actor?

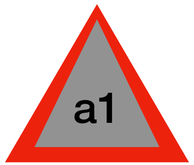
Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,]$ $\wedge$ $\forall a'.\,[\, a \notin a'.\text{MsgQueue}\,]$

not idle

Idle

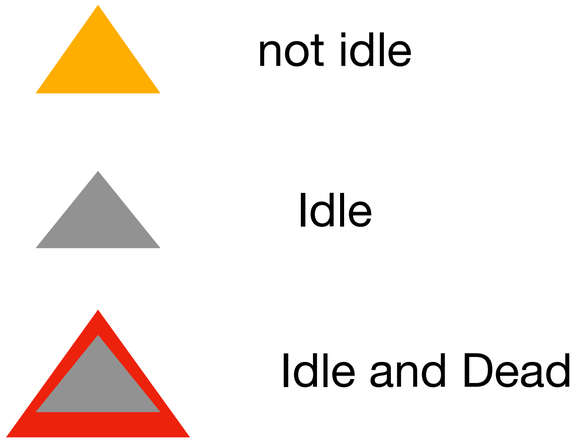
Idle and Dead

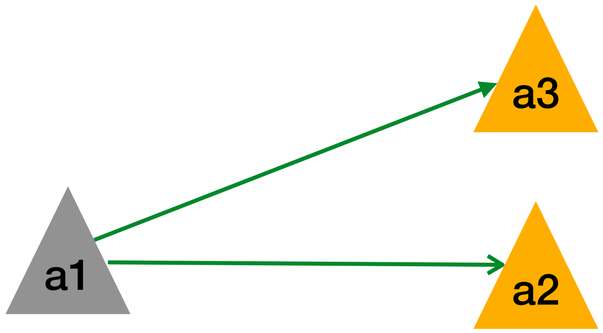


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'.[a$\in$a'.heap $\Longrightarrow$ Dead(a')] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

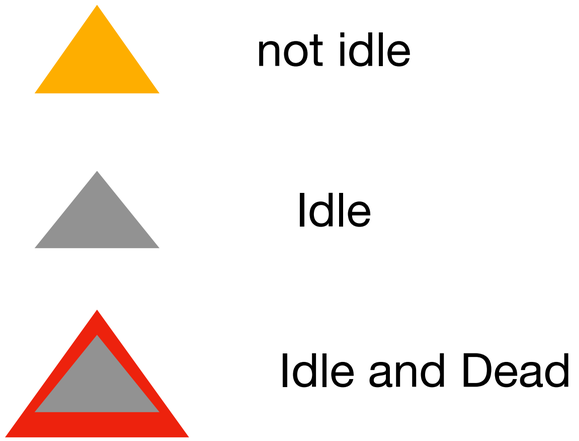


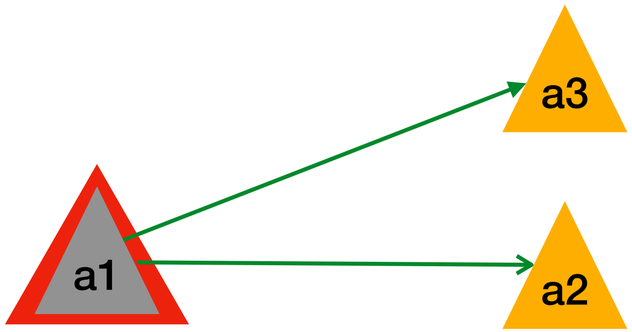


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,] \wedge \forall a'.[\, a \notin a'.\text{MsgQueue}\,]$

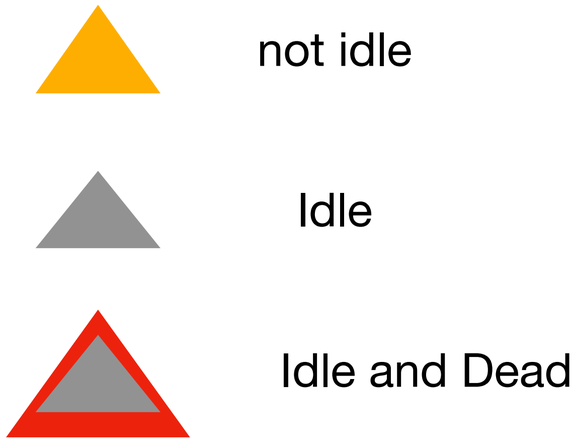




What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

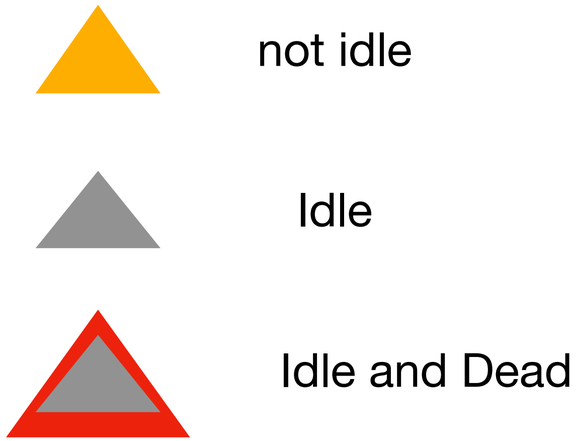
Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,] \wedge \forall a'.[\, a \notin a'.\text{MsgQueue}\,]$

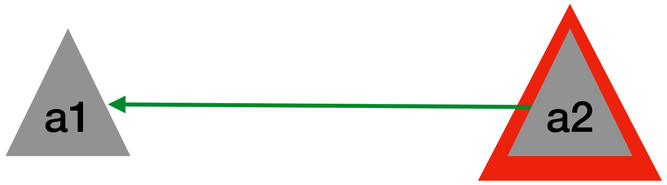


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'.[a$\in$a'.heap $\Longrightarrow$ Dead(a')] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

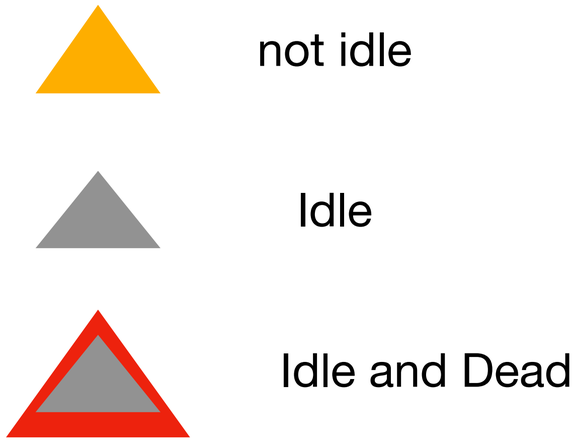


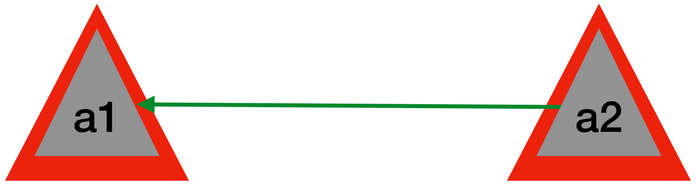


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,] \wedge \forall a'.[\, a \notin a'.\text{MsgQueue}\,]$

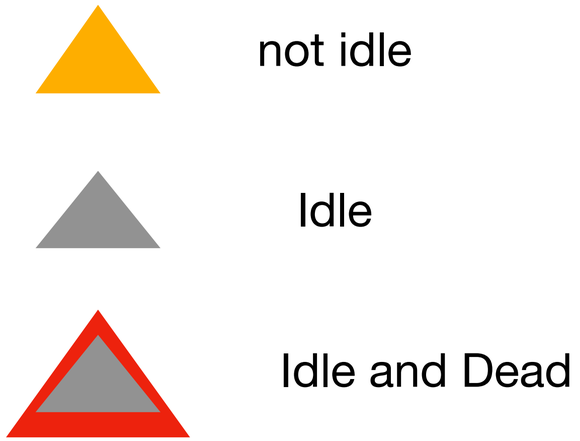




What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

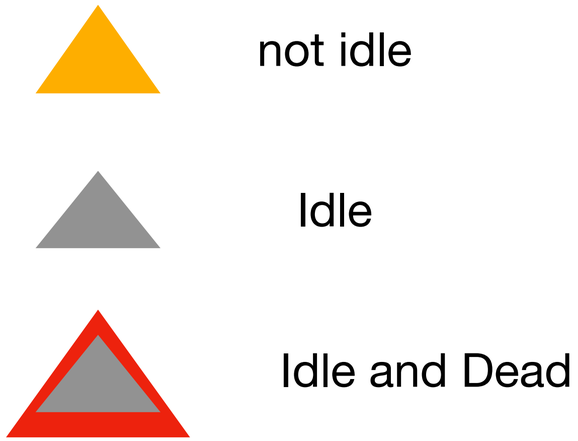
Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,] \wedge \forall a'.[\, a \notin a'.\text{MsgQueue}\,]$



What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

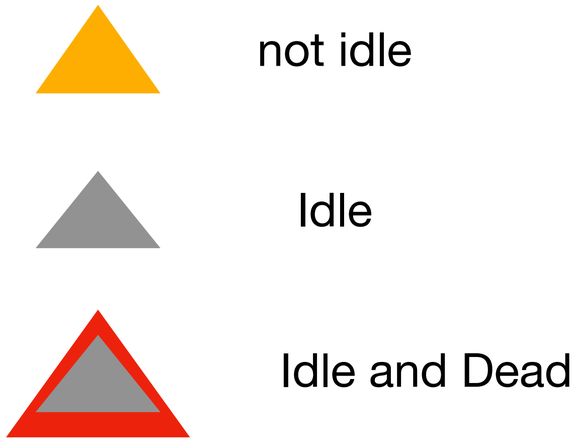
Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,] \wedge \forall a'.[\, a \notin a'.\text{MsgQueue}\,]$



What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

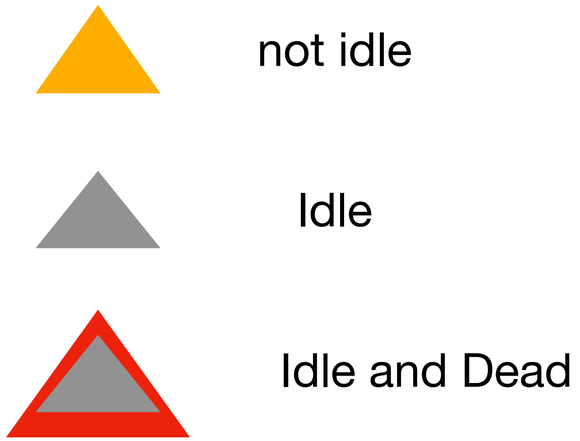
Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,] \wedge \forall a'.[\, a \notin a'.\text{MsgQueue}\,]$

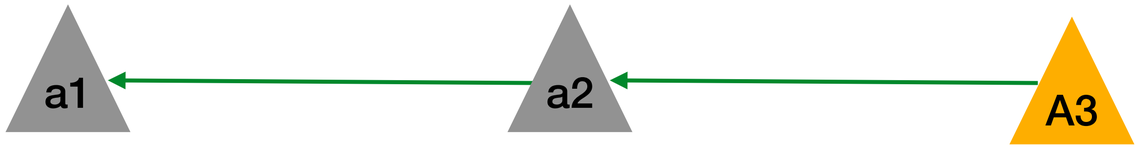


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'.[a$\in$a'.heap $\Longrightarrow$ Dead(a')] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

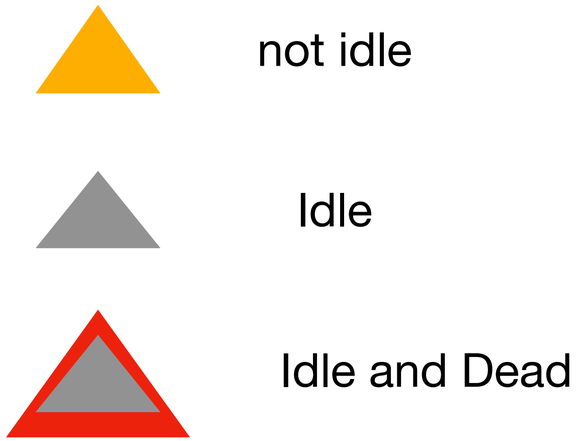


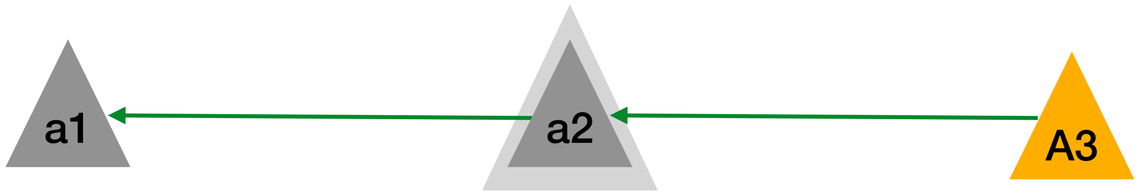


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,] \wedge \forall a'.[\, a \notin a'.\text{MsgQueue}\,]$

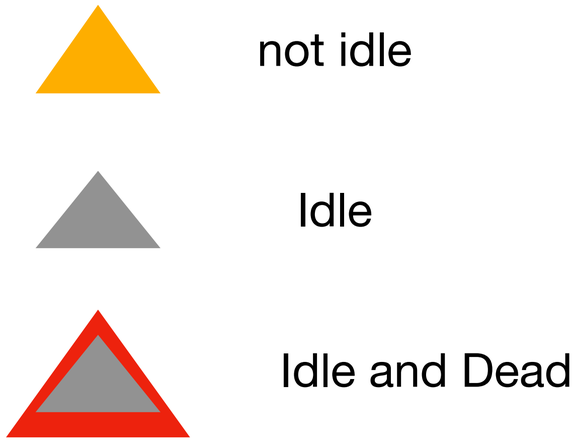


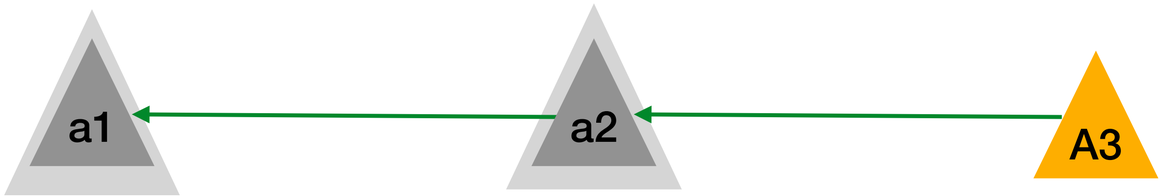


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'.[a$\in$a'.heap $\Longrightarrow$ Dead(a')] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

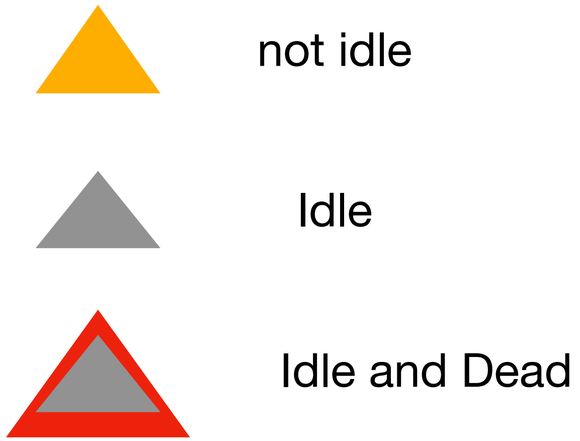




What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

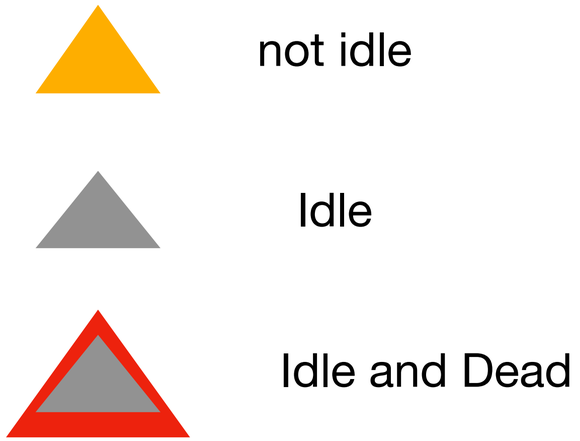
Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,]$ $\wedge$ $\forall a'.\,[\, a \notin a'.\text{MsgQueue}\,]$

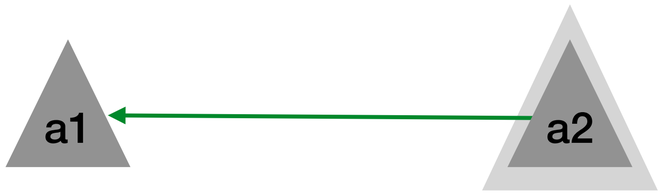


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'.[a$\in$a'.heap $\Longrightarrow$ Dead(a')] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

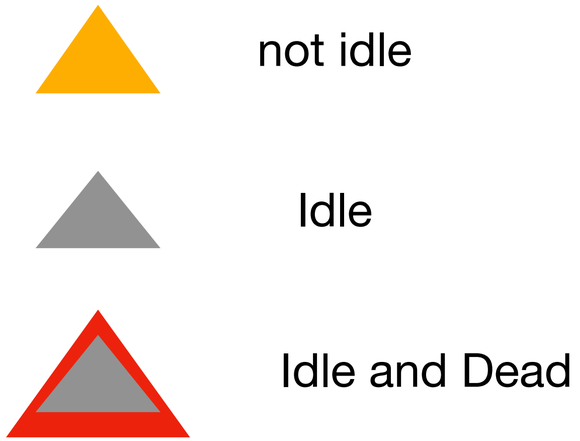


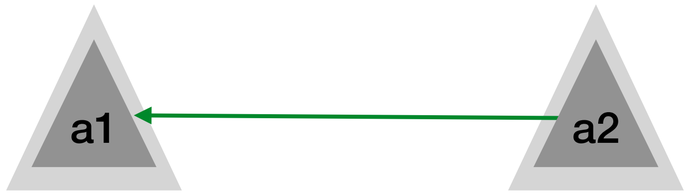


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'.[a$\in$a'.heap $\Longrightarrow$ Dead(a')] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]





What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.[\, a \in a'.\text{heap} \Longrightarrow \text{Dead}(a')\,]$ $\wedge$ $\forall a'.\,[\, a \notin a'.\text{MsgQueue}\,]$

not idle

Idle

Idle and Dead

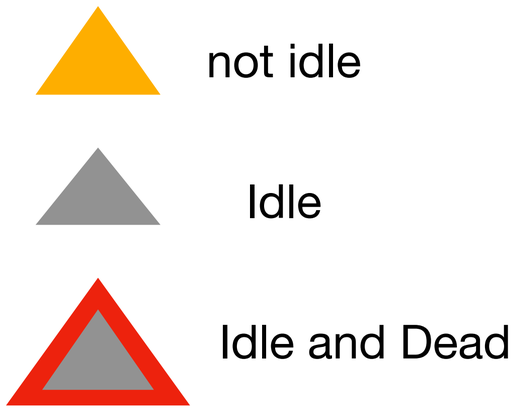
What is the challenge?

What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.$ [$a' \in a.heap \implies$ Dead(a)] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $\forall a'.$ [$a_j \in a'.heap \implies \exists k \in [1..n].a' = a_k$] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]]

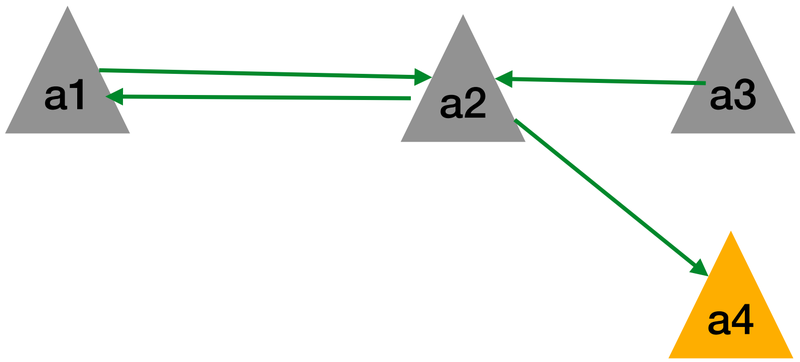


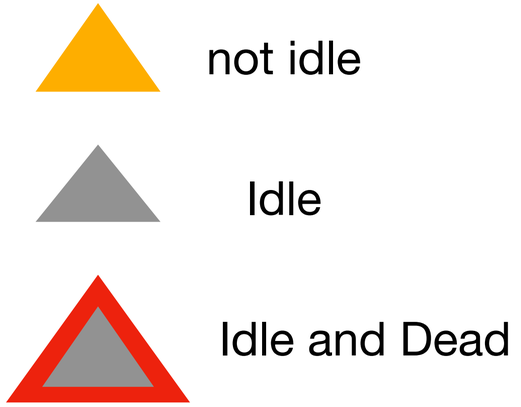
What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j \in$ a'.heap $\Longrightarrow$ $\exists k \in [1..n].a' = a_k$] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]



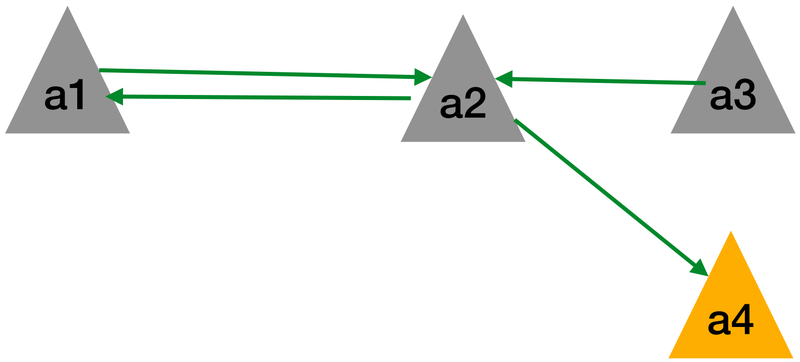


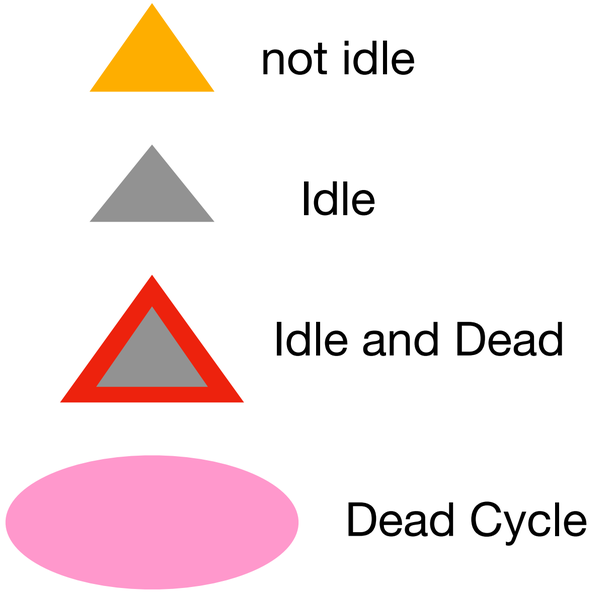
What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j \in$a'.heap $\Longrightarrow$ $\exists k \in [1..n].a' = a_k$] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]



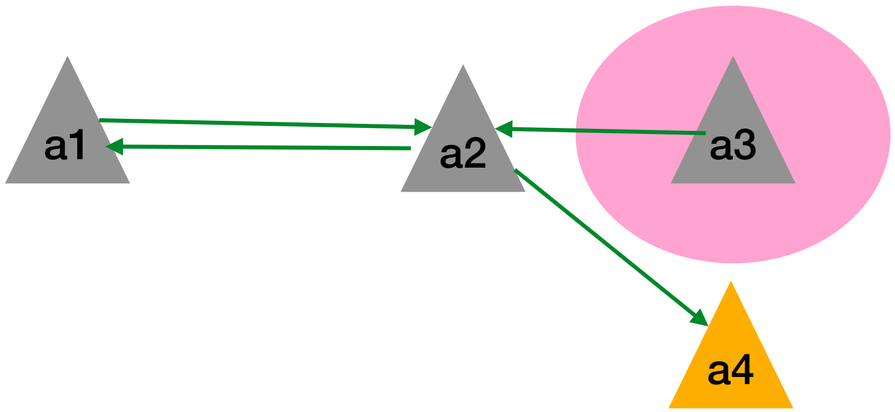


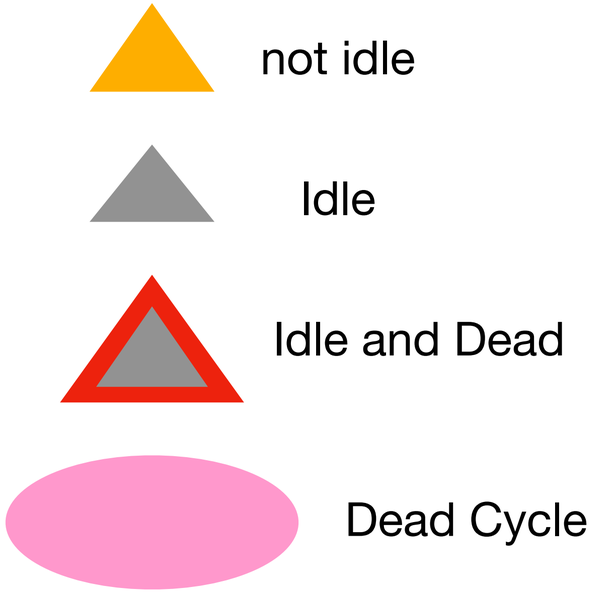
What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j \in$a'.heap $\Longrightarrow$ $\exists k \in [1..n].a'=a_k$] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]



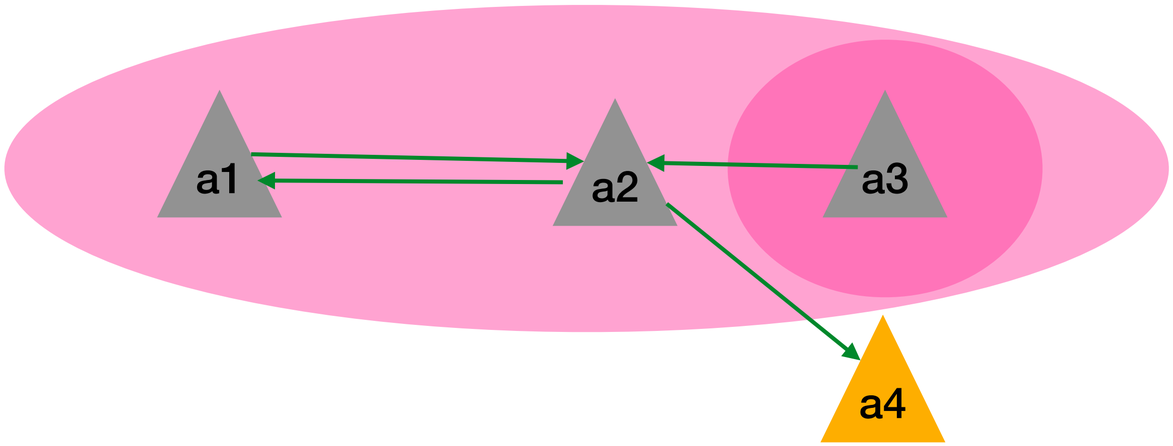


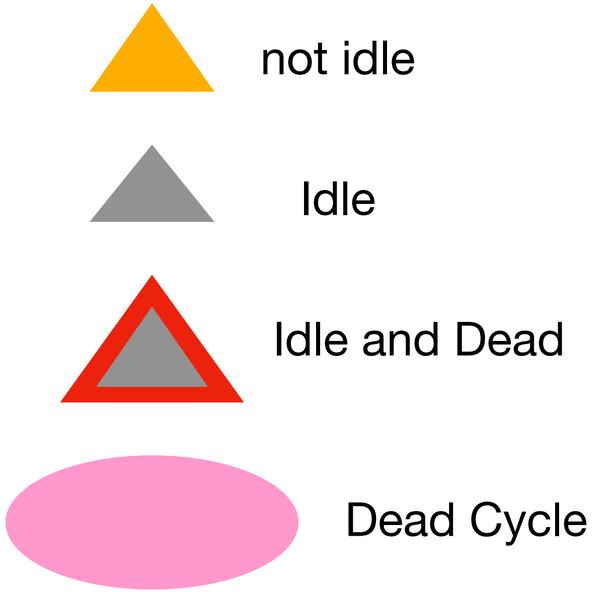
What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j \in$a'.heap $\Longrightarrow$ $\exists k \in [1..n].a'=a_k$] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]



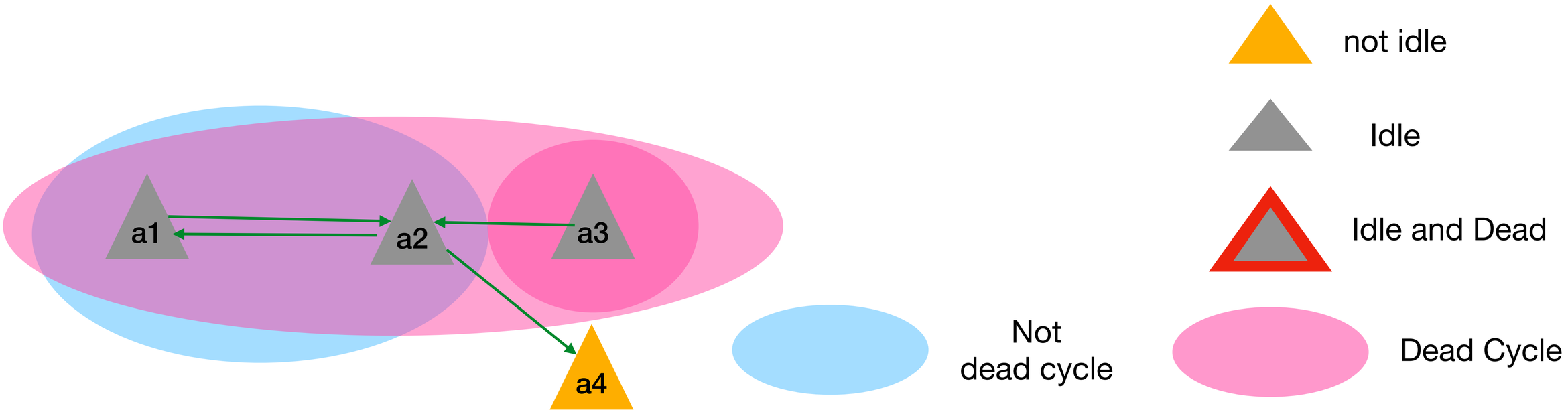


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j \in$a'.heap $\Longrightarrow$ $\exists k \in [1..n].a'=a_k$] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]

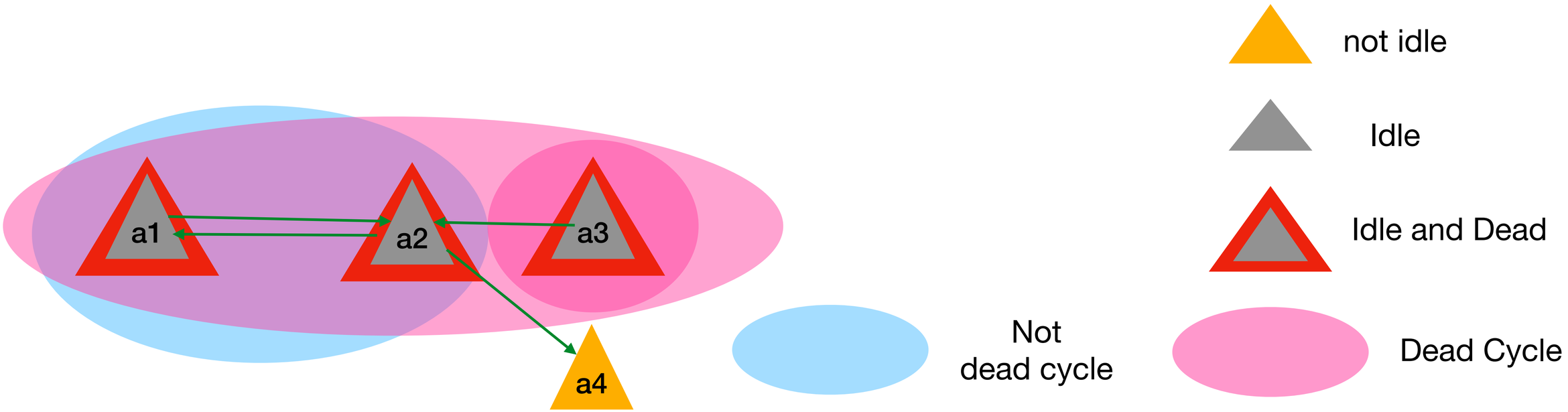


What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j \in$a'.heap $\Longrightarrow$ $\exists k \in [1..n].a'=a_k$] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]



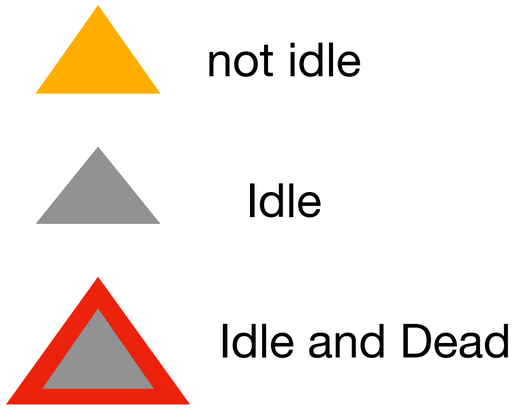
What is a dead actor?

$Idle(a)$ iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

$Dead(a)$ iff $Idle(a) \wedge \forall a'.\ [\ a' \in a.heap \implies Dead(a)\ ] \wedge \forall a'.\ [\ a \notin a'.MsgQueue\ ]$

$DeadCycle(a_1,.. a_n)$ iff $\forall j \in [1..n].$
$[\ Idle(a_j) \wedge \forall a'.\ [\ a_j \in a'.heap \implies \exists j \in [1..n].a' = a_j\ ] \wedge \forall a'.\ [\ a \notin a'.MsgQueue\ ]\ ]$

Lemma: $Dead(a) \implies \exists\ a_2,.. a_n.\ DeadCycle(a,a_2,.. a_n)$



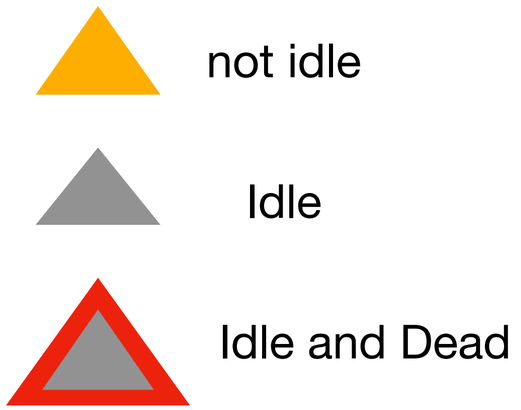
What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j$$\in$a'.heap $\Longrightarrow$ $\exists j \in [1..n]$.a'=a_j] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]

Lemma: Dead(a) $\Longrightarrow$ $\exists$ a_2,.. a_n. DeadCycle(a,a_2,.. a_n)



Therefore, we will be searching for dead cycles!

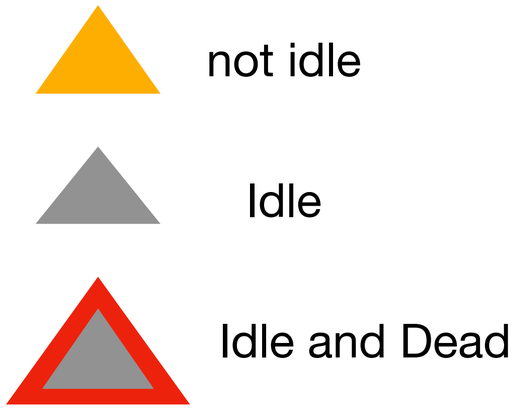
What is a dead actor?

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j$$\in$a'.heap $\Longrightarrow$ $\exists j \in [1..n]$.a'=a_j] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]

Lemma: Dead(a) $\Longrightarrow$ $\exists$ a_2,.. a_n. DeadCycle(a,a_2,.. a_n)



Therefore, we will be searching for dead cycles!

What is the challenge?

Searching for DeadCycle

$\text{Idle}(a)$ iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

$\text{Dead}(a)$ iff $\text{Idle}(a) \wedge \forall a'.\ [\ a' \in a.\text{heap} \Longrightarrow \text{Dead}(a)\ ] \ \wedge \ \forall a'.\ [\ a \notin a'.\text{MsgQueue}\ ]$

$\text{DeadCycle}(a_1,.. a_n)$ iff $\forall j \in [1..n].$

$[\ \text{Idle}(a_j) \ \wedge \forall a'.\ [\ a_j \in a'.\text{heap} \Longrightarrow \exists k \in [1..n].a' = a_k\ ] \ \wedge \ \forall a'.\ [\ a \notin a'.\text{MsgQueue}\ ]\ ]$

Searching for DeadCycle

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall$a'. [a'$\in$a.heap $\Longrightarrow$ Dead(a)] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ $\forall$a'. [$a_j \in$a'.heap $\Longrightarrow$ $\exists k \in [1..n]$.a'=a_k] $\wedge$ $\forall$a'. [a$\notin$a'.MsgQueue]]

DeadCycle'(a_1,.. a_n) iff $\forall j \in [1..n]$.
[Idle(a_j) $\wedge$ | { a' | $a_j \in$a'.heap } | = | { k | $k \in [1..n] \wedge$ $a_j \in a_k$.heap } | $\wedge$
| { (a',i) | $a_j \in$a'.MsgQueue[i] } | =0]

Searching for DeadCycle

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.$ [$a' \in a.heap \Longrightarrow$ Dead(a)] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $\forall a'.$ [$a_j \in a'.heap \Longrightarrow \exists k \in [1..n].a' = a_k$] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]]

DeadCycle'(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $|\{ a' \mid a_j \in a'.heap \}| = |\{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \}|$ $\wedge$
$|\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}| = 0$]

Searching for DeadCycle

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.$ [$a' \in a.heap \Longrightarrow$ Dead(a)] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $\forall a'.$ [$a_j \in a'.heap \Longrightarrow \exists k \in [1..n].a' = a_k$] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]]

DeadCycle'(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $| \{ a' \mid a_j \in a'.heap \} | = | \{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \} |$ $\wedge$
$| \{ (a',i) \mid a_j \in a'.MsgQueue[i] \} | = 0$]

Searching for DeadCycle

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.$ [$a' \in a.heap \Longrightarrow$ Dead(a)] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $\forall a'.$ [$a_j \in a'.heap \Longrightarrow \exists k \in [1..n].a' = a_k$] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]]

DeadCycle'(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $| \{ a' \mid a_j \in a'.heap \} | = | \{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \} |$ $\wedge$
$| \{ (a',i) \mid a_j \in a'.MsgQueue[i] \} | = 0$]

Searching for DeadCycle

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.$ [$a' \in a.heap \Longrightarrow$ Dead(a)] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $\forall a'.$ [$a_j \in a'.heap \Longrightarrow \exists k \in [1..n].a' = a_k$] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]]

DeadCycle'(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $| \{ a' \mid a_j \in a'.heap \} | = | \{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \} |$ $\wedge$
$| \{ (a',i) \mid a_j \in a'.MsgQueue[i] \} | = 0$]

Searching for DeadCycle

Idle(a) iff (a is not executing a behaviour) $\wedge$ (a.MsgQueue is empty)

Dead(a) iff Idle(a) $\wedge$ $\forall a'.$ [$a' \in a.heap \Longrightarrow$ Dead(a)] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]

DeadCycle(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $\forall a'.$ [$a_j \in a'.heap \Longrightarrow \exists k \in [1..n].a' = a_k$] $\wedge$ $\forall a'.$ [$a \notin a'.MsgQueue$]]

DeadCycle'(a_1,.. a_n) iff $\forall j \in [1..n].$
[Idle(a_j) $\wedge$ $| \{ a' \mid a_j \in a'.heap \} | = | \{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \} |$ $\wedge$
$| \{ (a',i) \mid a_j \in a'.MsgQueue[i] \} | = 0$]

Lemma: DeadCycle(a_1,a_2,.. a_n) $\Longleftrightarrow$ DeadCycle'(a_1,a_2,.. a_n)

Searching for DeadCycle’

DeadCycle’$(a_1,.. a_n)$ iff $\forall j \in [1..n].$

$[\ Idle(a_j)\ \wedge |\{ a' \mid a_j \in a'.heap \}| = |\{ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap \}| \wedge$

$|\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}| = 0\ ]$

Searching for DeadCycle’

DeadCycle’(a_1,.. a_n) iff $\forall j \in [1..n].$

$[\ Idle(a_j) \wedge |\{ a' \mid a_j \in a'.heap \}| = |\{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \}| \wedge$
$|\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}| = 0\ ]$

Actors carry a RC-field of type INT such that

(INV) $a.RC + \ldots\ldots = |\{ a' \mid a \in a'.heap \}| + |\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}|$

Then, we have that

Lemma

$\forall j \in [1..n].\ [\ Idle(a_j) \wedge a_j.RC = |\{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \}| \implies$ DeadCycle’(a_1,.. a_n)]

Searching for DeadCycle’

DeadCycle’(a_1,.. a_n) iff $\forall j \in [1..n].$

$[\ Idle(a_j) \wedge |\{ a' \mid a_j \in a'.heap \}| = |\{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \}| \wedge$
$|\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}| = 0\ ]$

Actors carry a RC-field of type INT such that

(INV) $a.RC + \ldots\ldots = |\{ a' \mid a \in a'.heap \}| + |\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}|$

Then, we have that

Lemma

$\forall j \in [1..n].\ [\ Idle(a_j) \wedge a_j.RC = |\{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \}| \implies DeadCycle'(a_1,.. a_n)\ ]$

Is it easier now?

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j) \ \wedge \ a_j.RC = |\ \{\ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap\ \}\ | \quad \Longrightarrow \quad DeadCycle'(a_1,..\ a_n) \quad]$

And where

(INV) $\qquad a.RC + \quad ..... \qquad = |\ \{\ a' \mid a \in a'.heap\ \}\ | + |\ \{\ (a',i) \mid a_j \in a'.MsgQueue[i]\ \}\ |$

1) Complete … in INV
2) Preserve INV
3) Complete the protocol, using Lemma and INV
4) Take into account that topology / idleness changes concurrently with execution of protocol

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j) \ \wedge\ a_j.RC = |\ \{\ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap\ \}\ | \quad \Longrightarrow \quad DeadCycle'(a_1,..\ a_n)\ \ ]$

And where

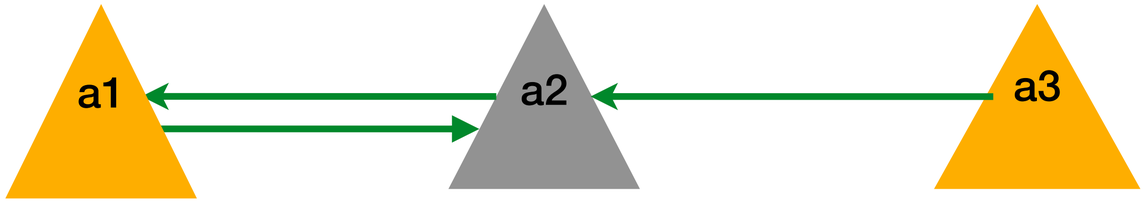
(INV) $\qquad a.RC + \quad ..... \qquad = |\ \{\ a' \mid a \in a'.heap\ \}\ | + |\ \{\ (a',i) \mid a_j \in a'.MsgQueue[i]\ \}\ |$

What remains to do?

1) Complete … in INV
2) Preserve INV
3) Complete the protocol, using Lemma and INV
4) Take into account that topology / idleness changes concurrently with execution of protocol

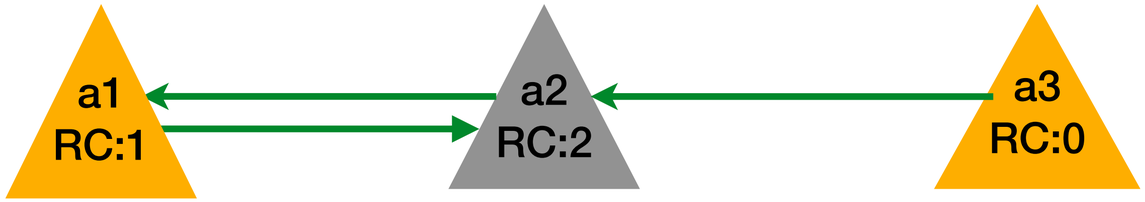
Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |



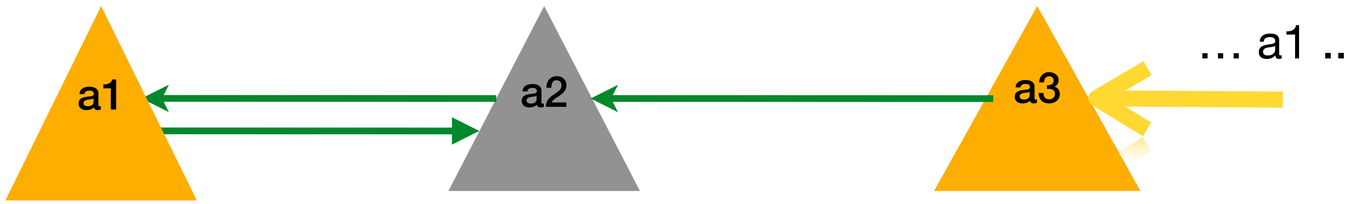
Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |



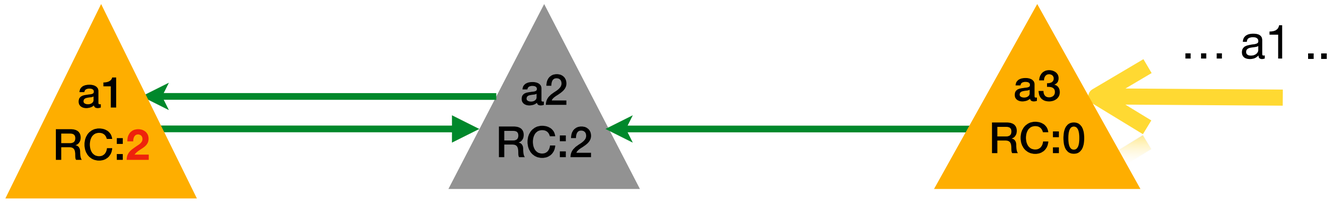
Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |



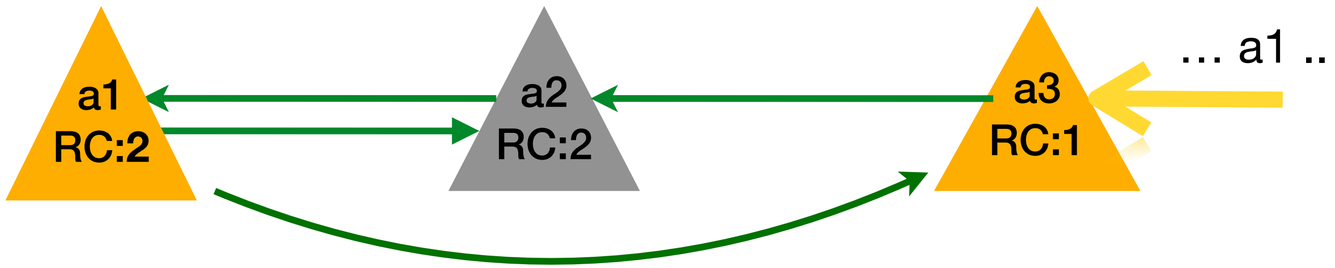
Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |



Preserve and Complete INV

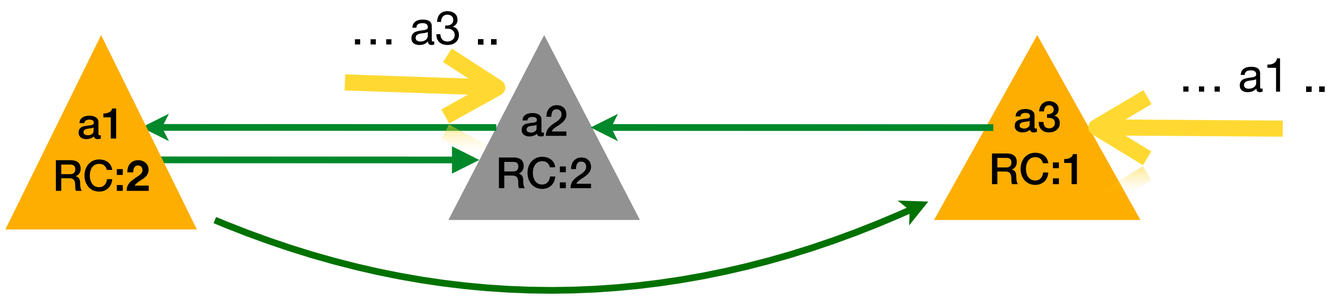
(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |



Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |

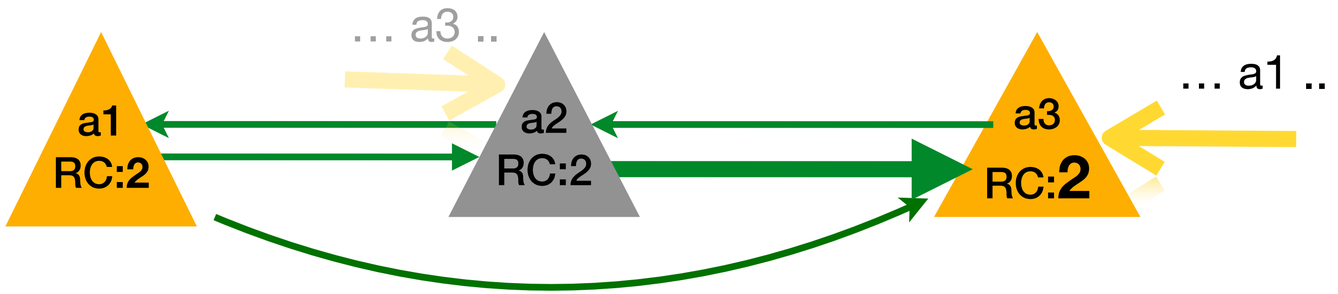
Imagine that a1 sends to a2 a message …a3…



Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |

Imagine that a1 sends to a2 a message …a3…

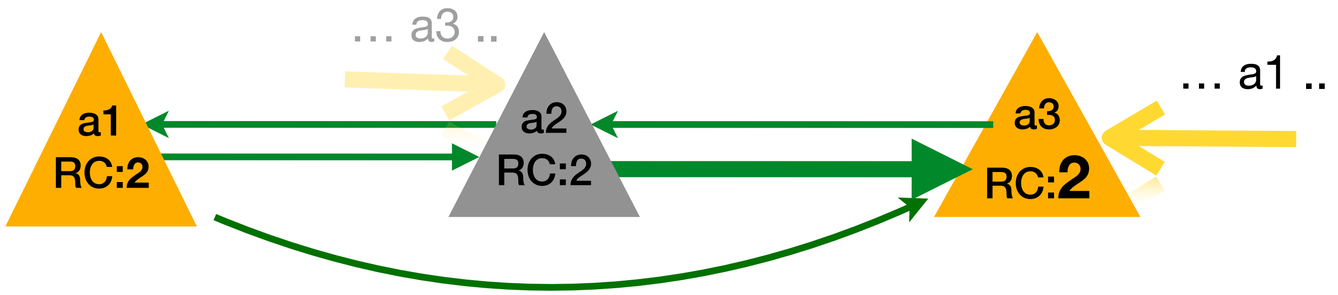


Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |

Imagine that a1 sends to a2 a message …a3…

when a2 reads the message, then a2 **will have a reference to a3**, the RC of a3 should become **2**

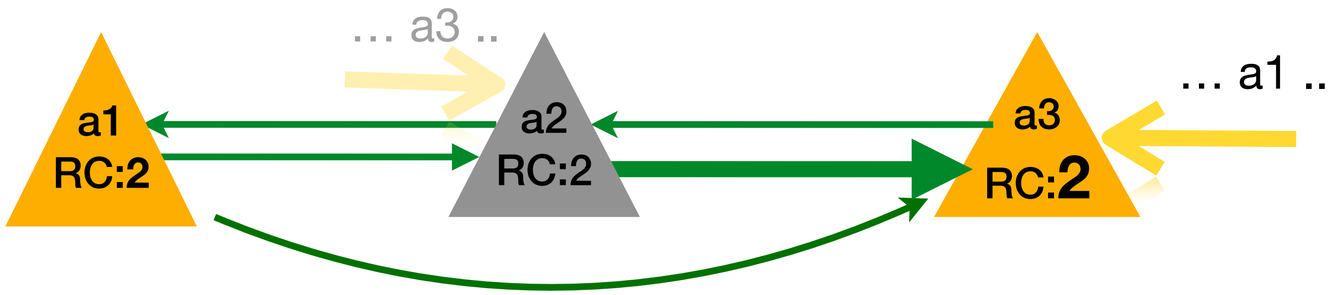


Preserve and Complete INV

(INV) a.RC + ... = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |

Imagine that a1 sends to a2 a message …a3…

when a2 reads the message, then a2 **will have a reference to a3**, the RC of a3 should become **2**



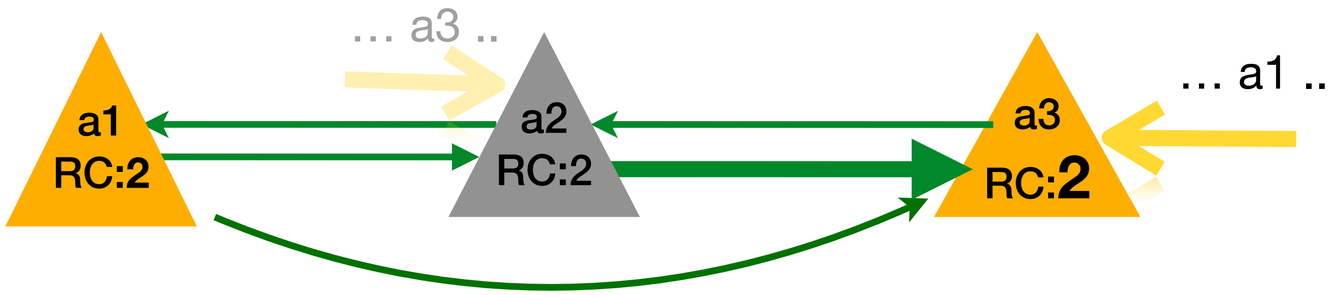
How will a3 know to increment its RC?

Preserve and Complete INV

(INV) a.RC + ….. = | { a’ | a∈a’.heap } | + | { (a’,i) | a_j∈a’.MsgQueue[i] } |

Imagine that a1 sends to a2 a message …a3…

when a2 reads the message, then a2 **will have a reference to a3**, the RC of a3 should become **2**



How will a3 know to increment its RC?

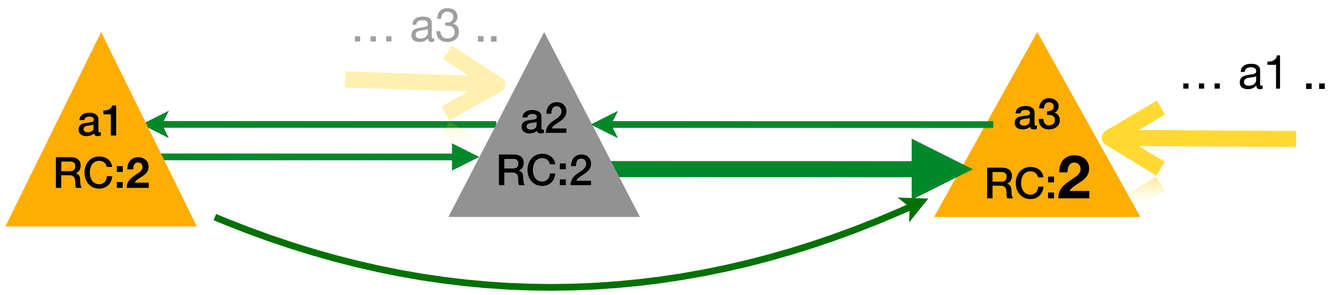
How is INV preserved?

Preserve and Complete INV

(INV) a.RC + ….. = | { a’ | a∈a’.heap } | + | { (a’,i) | a_j∈a’.MsgQueue[i] } |

Imagine that a1 sends to a2 a message …a3…

when a2 reads the message, then a2 **will have a reference to a3**, the RC of a3 should become **2**



How will a3 know to increment its RC? a1 will send an INC message to a3

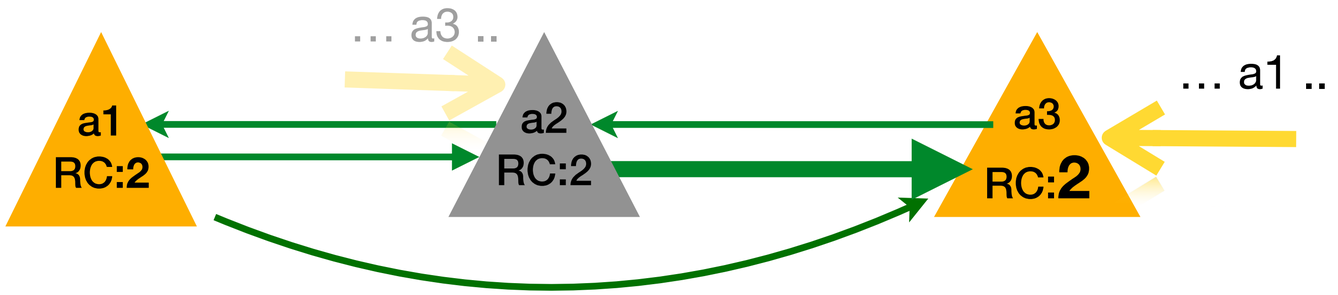
How is INV preserved?

Preserve and Complete INV

(INV) a.RC + ….. = | { a’ | a∈a’.heap } | + | { (a’,i) | a_j∈a’.MsgQueue[i] } |

Imagine that a1 sends to a2 a message …a3…

when a2 reads the message, then a2 **will have a reference to a3**, the RC of a3 should become **2**



How will a3 know to increment its RC?

a1 will send an INC message to a3

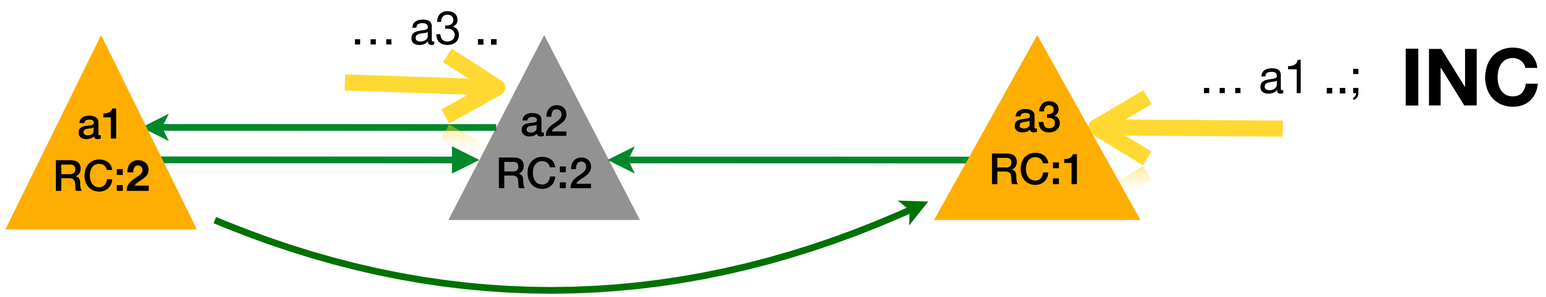
How is INV preserved?

INV takes the INC (and DEC messages) into account

Preserve and Complete INV

(INV) a.RC + **a.INCDEC** = | { a’ | a∈a’.heap } | + | { (a’,i) | a_j∈a’.MsgQueue[i] } |

Imagine that a1 sends to a2 a message …a3…

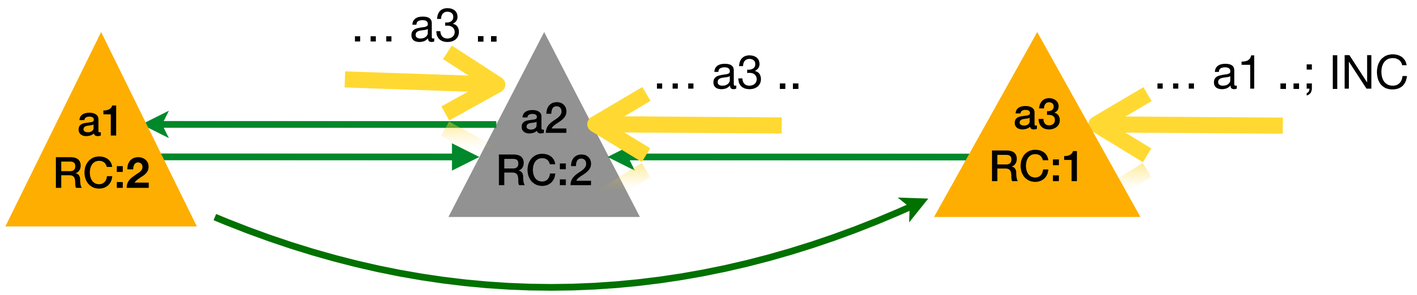


a.INCDEC = | { k | a.MsgQueue[k] = INC } | - { k | a.MsgQueue[k] = DEC } |

Preserving INV — sending message

(INV) $a.RC + a.INCDEC = |\{ a' \mid a \in a'.heap \}| + |\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}|$

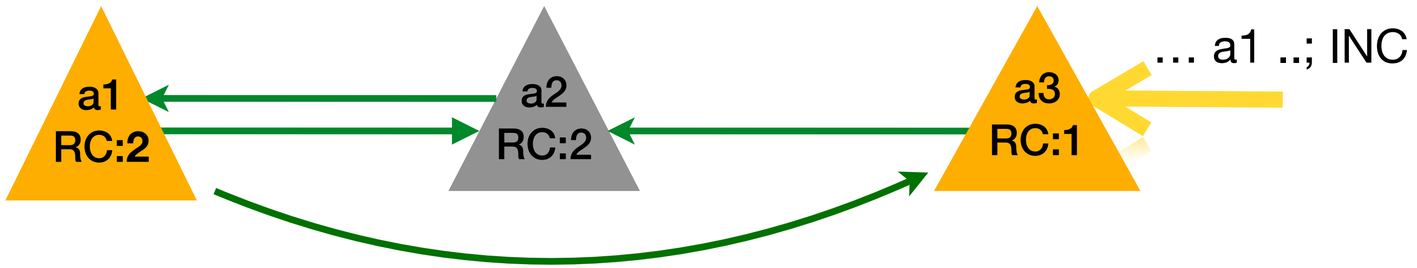
If a wants to send msg(a') to a"
then Append(a'.MsqQueue,INC)
Append(a".MsqQueue,msg(a"))



Preserving INV — receiving message

(INV) a.RC + a.INCDEC = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |

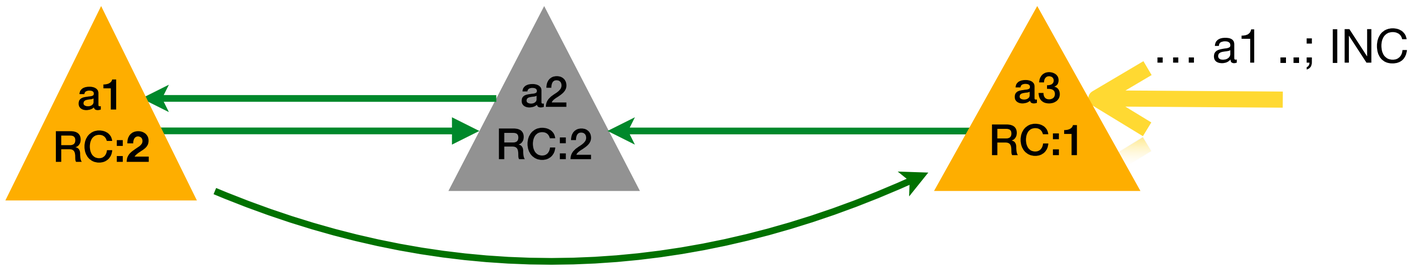
If Top(a.MsqQueue)=msg(a')
then Pop(a.MsgQueue);
if a'∈a.heap
then Append(a'.MsgQueue,DEC)



Preserving INV — receiving INC

(INV) a.RC + a.INCDEC = | { a' | a∈a'.heap } | + | { (a',i) | a_j∈a'.MsgQueue[i] } |

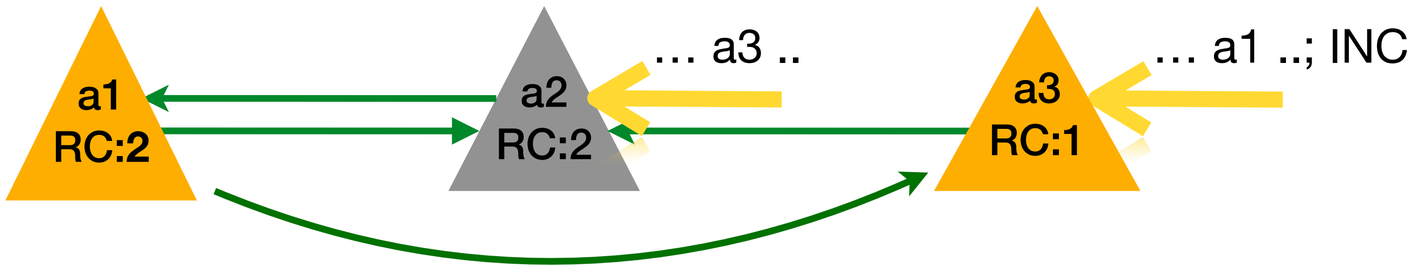
If Top(a.MsqQueue)=INC
then Pop(a.MsqQueue);
a.RC:=a.RC+1



Preserving INV — receiving DEC

(INV) $a.RC + a.INCDEC = |\{ a' \mid a \in a'.heap \}| + |\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}|$

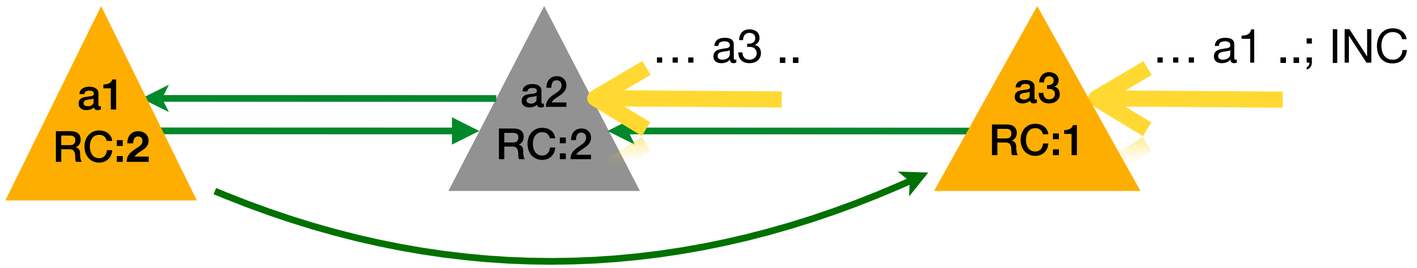
If Top(a.MsqQueue)=DEC
then Pop(a.MsqQueue);
a.RC:=a.RC-1



Preserving INV — local GC

(INV) a.RC + a.INCDEC = | { a’ | a∈a’.heap } | + | { (a’,i) | a_j∈a’.MsgQueue[i] } |

If a.heap drops reference to a’
then Append(a’.MsgQueue,DEC)



Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j) \ \wedge\ a_j.RC = |\{ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap \}| \quad \Longrightarrow\quad DeadCycle'(a_1,.. a_n)\quad]$

And where

(INV) $a.RC + a.INCDEC = |\{ a' \mid a \in a'.heap \}| + |\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}|$

1) ~~Complete … in INV~~
2) ~~Preserve INV~~
3) Complete the protocol using Lemma
4) Take into account that topology / idleness changes concurrently with execution of protocol

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j) \ \wedge \ a_j.RC = |\{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \}| \quad \Longrightarrow \quad DeadCycle'(a_1,.. a_n) \quad]$

And where

(INV) $a.RC + a.INCDEC = |\{ a' \mid a \in a'.heap \}| + |\{ (a',i) \mid a_j \in a'.MsgQueue[i] \}|$

What remains to do?

1) ~~Complete … in INV~~
2) ~~Preserve INV~~
3) Complete the protocol using Lemma
4) Take into account that topology / idleness changes concurrently with execution of protocol

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j)\ \wedge\ a_j.RC = |\{\, k \mid k \in [1..n] \wedge\ a_j \in a_k.heap \,\}|\quad \Longrightarrow\quad DeadCycle'(a_1,..\ a_n)\quad]$

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j) \ \wedge \ a_j.RC = |\{ k \mid k \in [1..n] \wedge a_j \in a_k.heap \}| \quad \Longrightarrow \quad DeadCycle'(a_1,.. a_n) \quad]$

We introduce a global cycle detector (CD). When an actor becomes idle, it sends to CD a BLK-message with

a) a.RC

b) { a’ | a’ ∈a.heap }

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j)\ \wedge\ a_j.RC = |\{\ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap\ \}|\quad \Longrightarrow\quad DeadCycle'(a_1,..\ a_n)\quad]$

We introduce a global cycle detector (CD). When an actor becomes idle, it sends to CD a BLK-message with

a) a.RC

b) { a’ | a’ ∈a.heap }

Based on the collected information, the CD can determine whether a set forms a DeadCycle’.

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j)\ \wedge\ a_j.RC = |\{\ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap\ \}|\quad \Longrightarrow\quad DeadCycle'(a_1,..\ a_n)\quad]$

We introduce a global cycle detector (CD). When an actor becomes idle, it sends to CD a BLK-message with

a) a.RC

b) { a’ | a’ ∈a.heap }

Based on the collected information, the CD can determine whether a set forms a DeadCycle’.

What if the topology/idleness changed after the actor sent BLK-msg to CD?

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j)\ \wedge\ a_j.RC = |\{\ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap\ \}|\quad \Longrightarrow\quad DeadCycle'(a_1,..\ a_n)\quad]$

We introduce a global cycle detector (CD). When an actor becomes idle, it sends to CD a BLK-message with

a) a.RC

b) { a’ | a’ ∈a.heap }

Based on the collected information, the CD can determine whether a set forms a DeadCycle’.

What if the topology/idleness changed after the actor sent BLK-msg to CD?

If an actor receives a INC/DEC/application message after having sent BLK, it sends UNBLK to CD, who discards this actor’s entry.

Searching for DeadCycle’

Lemma

$\forall j \in [1..n].\ [\quad Idle(a_j)\ \wedge\ a_j.RC = |\{\ k \mid k \in [1..n] \wedge\ a_j \in a_k.heap\ \}|\quad \Longrightarrow\quad DeadCycle'(a_1,..\ a_n)\quad]$

We introduce a global cycle detector (CD). When an actor becomes idle, it sends to CD a BLK-message with

a) a.RC

b) { a’ | a’ ∈a.heap }

Based on the collected information, the CD can determine whether a set forms a DeadCycle’.

What if the topology/idleness changed after the actor sent BLK-msg to CD?

If an actor receives a INC/DEC/application message after having sent BLK, it sends UNBLK to CD, who discards this actor’s entry.

When the CD has found a DeadCycle’(a_1,.. a_n), then it creates a fresh token t, and sends a CONF(t) request to all wh respond with ACK(t). When it receives ACK from all the actors, it collects a_1,.. a_n

Reflections on MAC

Is that the whole story?

Is it useful?

Is it sound?

Is it complete?

Does separation play any role?

What are its drawbacks?

Reflections on MAC

Is that the whole story?

Is it useful? Yes

Is it sound?

Is it complete?

Does separation play any role?

What are its drawbacks?

Reflections on MAC

Is that the whole story?

Is it useful? Yes

Is it sound? **YES** — and we have two latex proofs.

Is it complete?

Does separation play any role?

What are its drawbacks?

Reflections on MAC

Is that the whole story?

Is it useful? Yes

Is it sound? **YES** — and we have two latex proofs.

Is it complete? **YES** — and we have one latex proof.

Does separation play any role?

What are its drawbacks?

Reflections on MAC

Is that the whole story?

Is it useful? Yes

Is it sound? **YES** — and we have two latex proofs.

Is it complete? **YES** — and we have one latex proof.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

Reflections on MAC

Is that the whole story?

Is it useful? Yes

Is it sound? **YES** — and we have two latex proofs.

Is it complete? **YES** — and we have one latex proof.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

CD is a bottleneck.

Reflections on MAC

Is that the whole story?

Is it useful? Yes

Is it sound? **YES** — and we have two latex proofs.

Is it complete? **YES** — and we have one latex proof.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

CD is a bottleneck.

CD performs a complex search on a large data structure.

Reflections on MAC

Is that the whole story?

Is it useful? Yes

Is it sound? **YES** — and we have two latex proofs.

Is it complete? **YES** — and we have one latex proof.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

CD is a bottleneck.

CD performs a complex search on a large data structure.

We need to “scan” every application message upon send/receive.

Reflections on MAC

Is that the whole story? **No**, some more clever ideas used.

Is it useful? Yes

Is it sound? **YES** — and we have two latex proofs.

Is it complete? **YES** — and we have one latex proof.

Does separation play any role? **YES** — when scanning messages, and own heap.

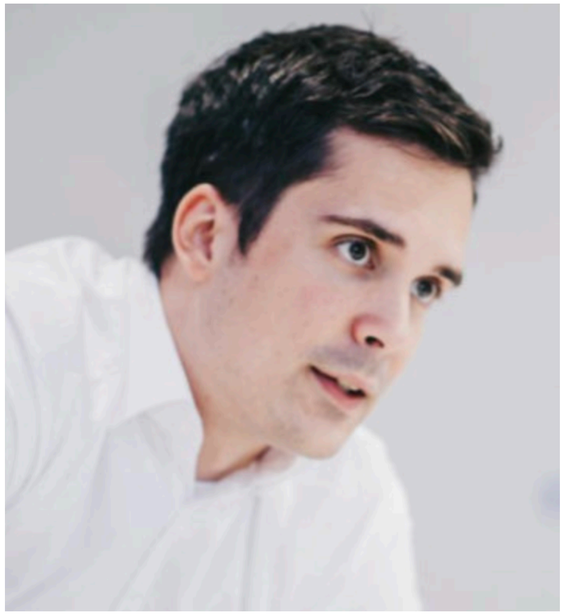
What are its drawbacks?

CD is a bottleneck.

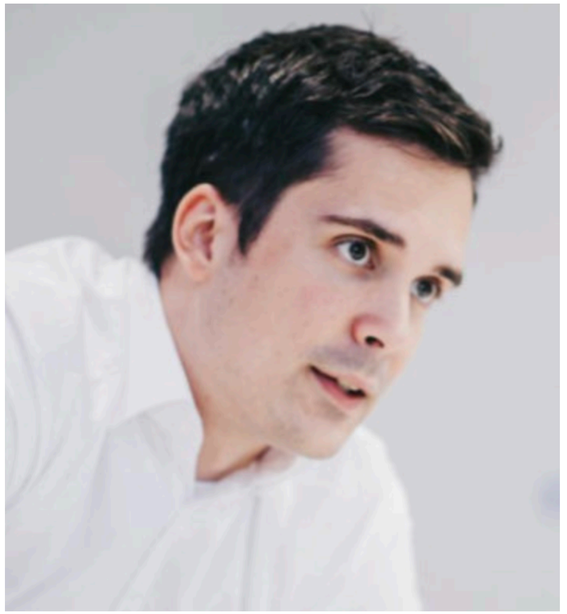
CD performs a complex search on a large data structure.

We need to “scan” every application message upon send/receive.

Enter eMAC - Sebastian Blessing



Enter eMAC - Sebastian Blessing



As of last Friday:

- MAC outperforms eMAC when there are a few very large cylces, and
- eMAC outperforms MAC when there are a lot of small cycles.

eMAC - basic idea

Perception: (ActorId, Number)
(A,k) : k out of A's direct predecessors are of unknown status

Perceptions: Perception*

Perceptions: Each actor keeps perception in its state
Actors notify their successors when their perception "improves"

eMAC - basic idea

Notice: ActorId × Number

A : k actor A has notified that a.RC-k of its immediate predecessors have reported idle

Notification: ActorId × Notice*

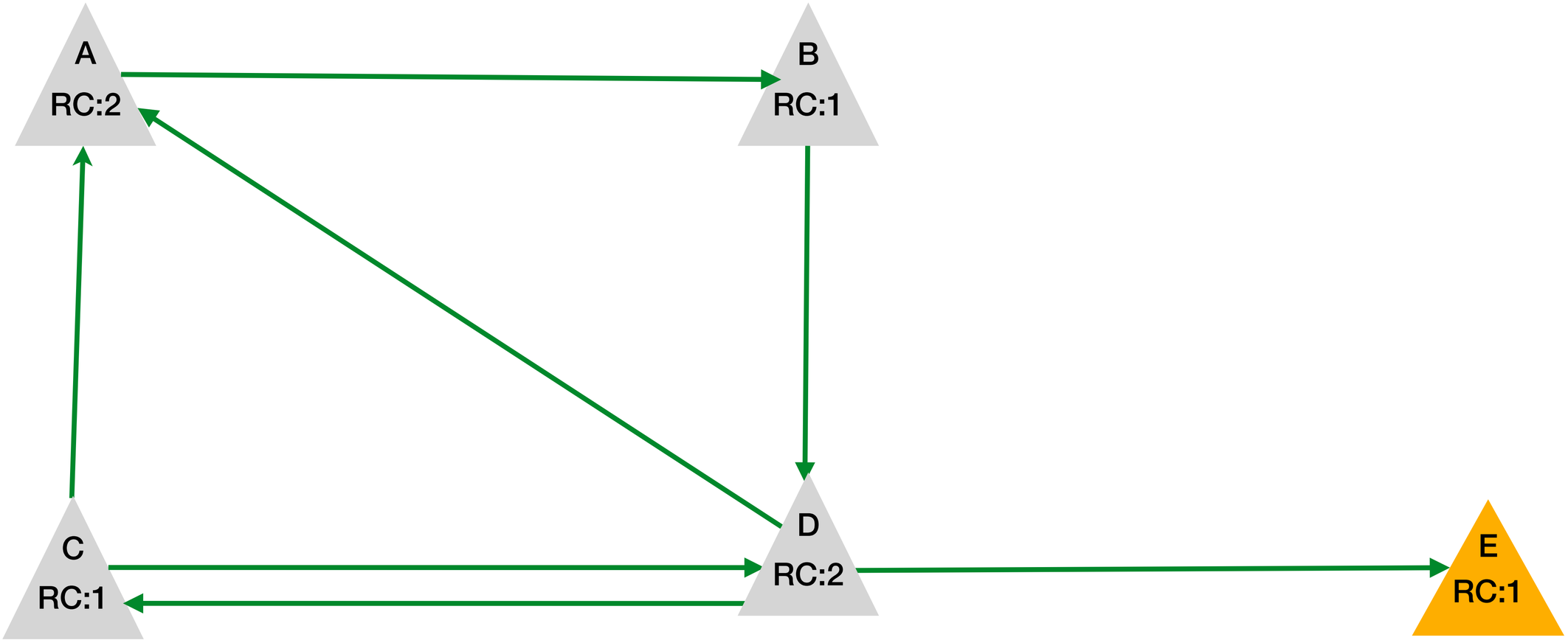
A, A_1:k_1…. A_n:k_n : A sends notices A_1:k_1 …. A_n:k_n

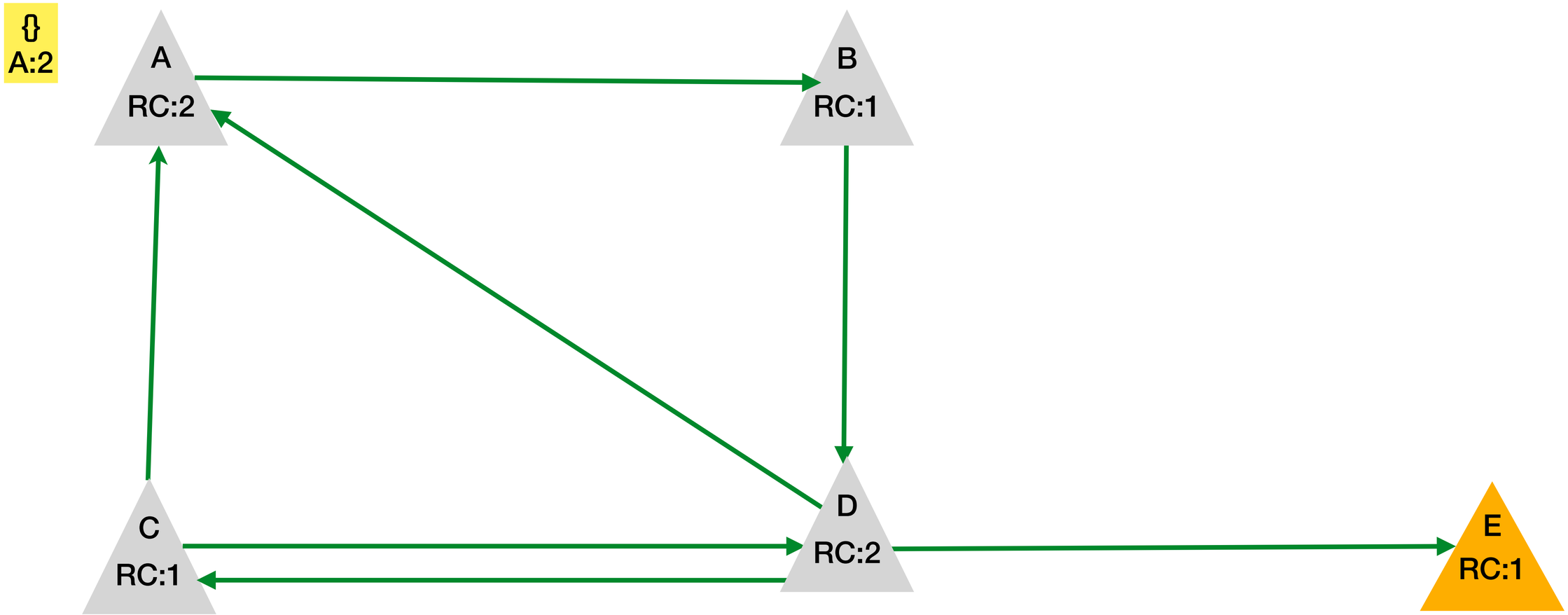
Perception: ActorId* × Notice*

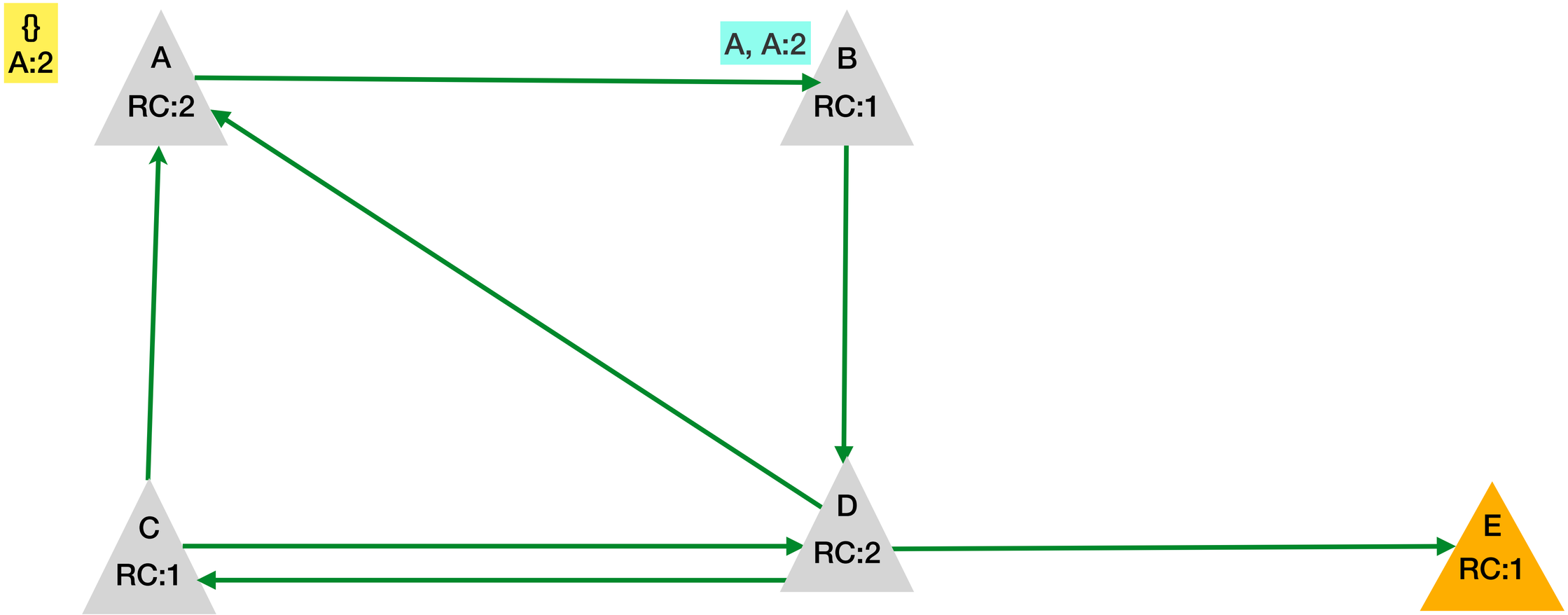
{ A_1, … A_n }, A'_1:k_1…. A'_m:k_m

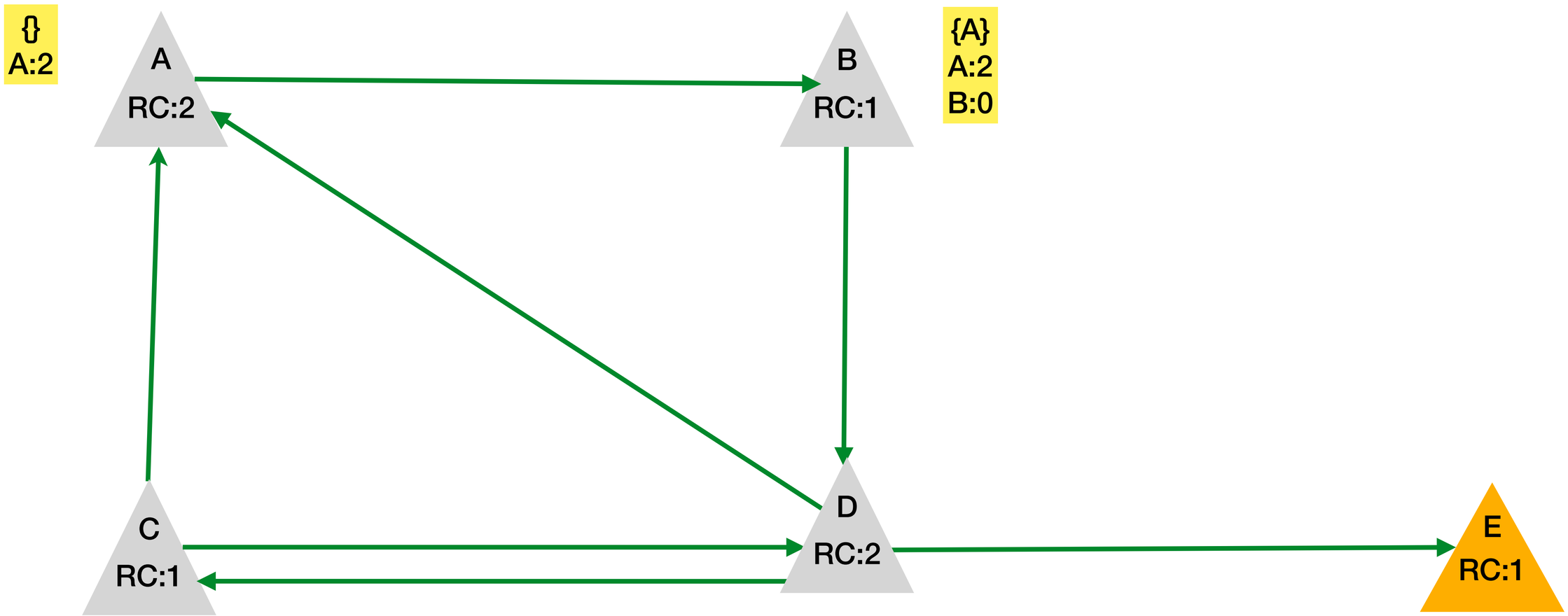
A_1, … A_n are some of the receiver's direct predecessors, and have sent notices to receiver
A'_1, … A'_m are some of the receiver's indirect predecessors, and have each sent notice A'_j:k_j

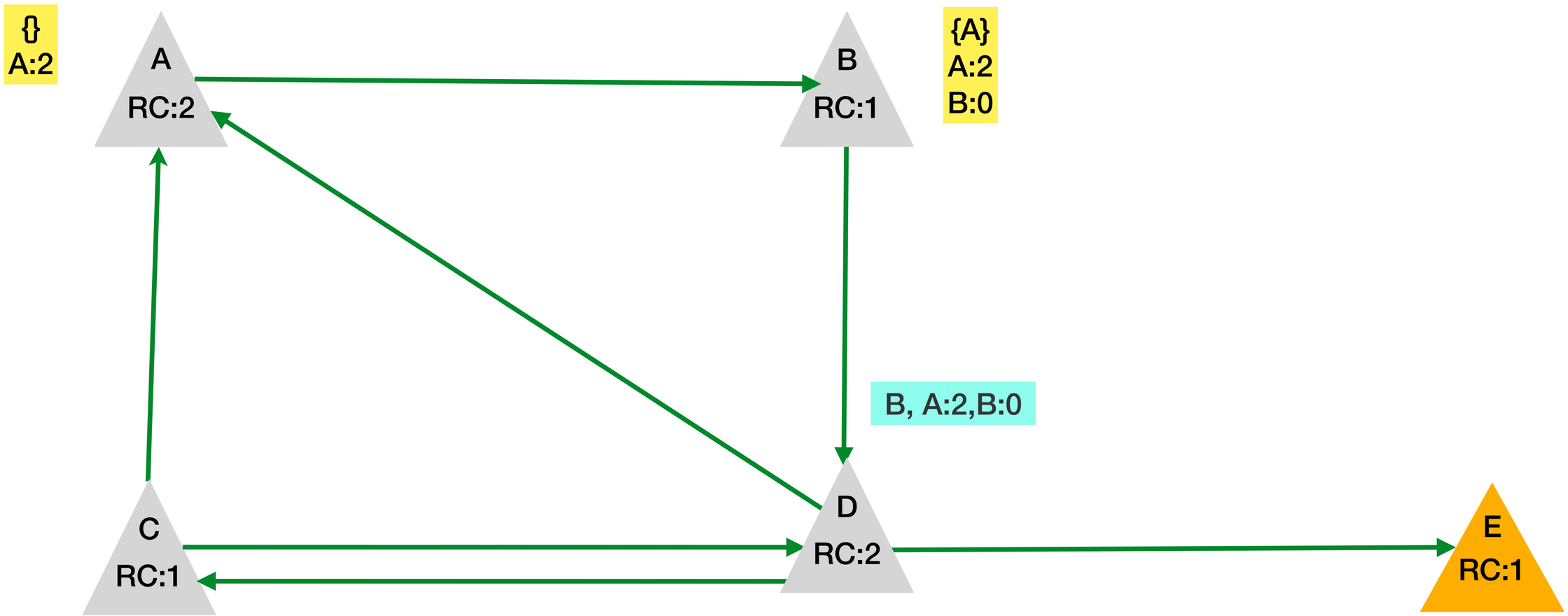
Protocol: Actors keep perception in their state
Actors notifies their immediate successors when they block, or their perception "improves"
Upon receipt of a notification, actors improve their perception
…..
…

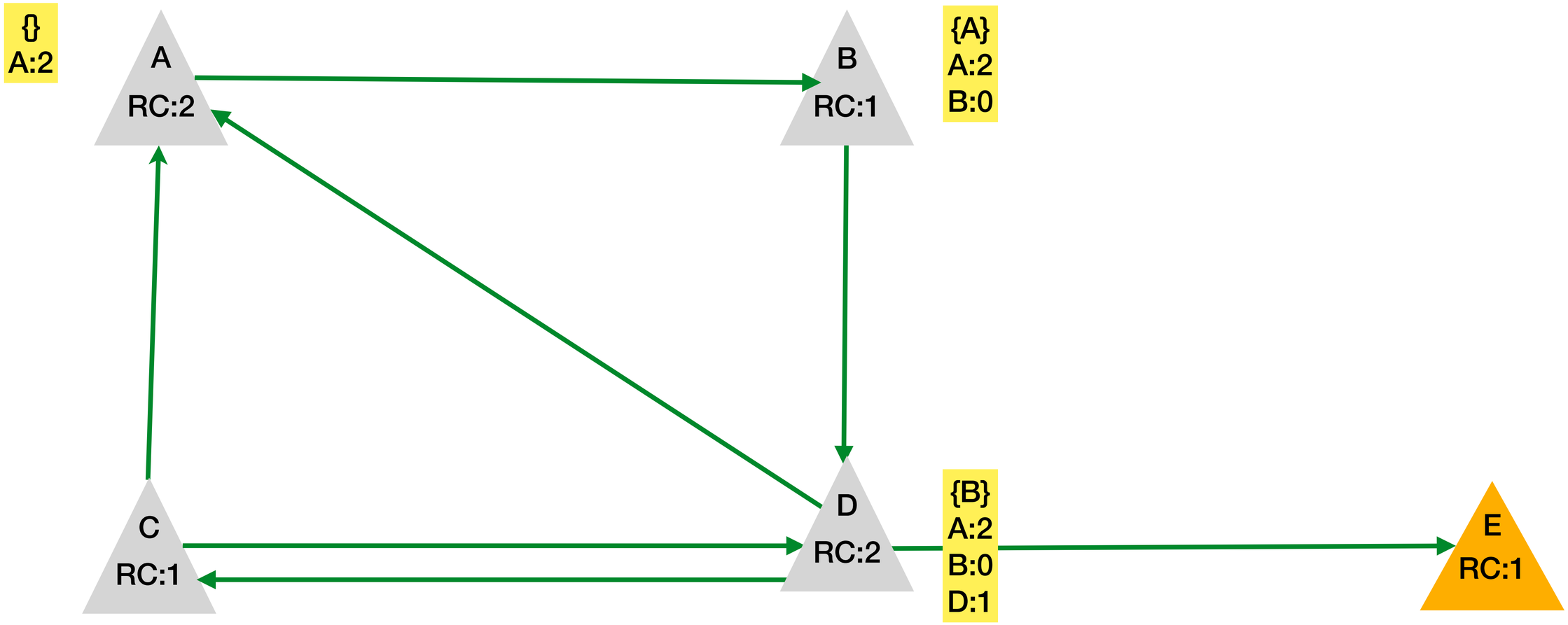

A
RC:2
B
RC:1
C
RC:1
D
RC:2
E
RC:1

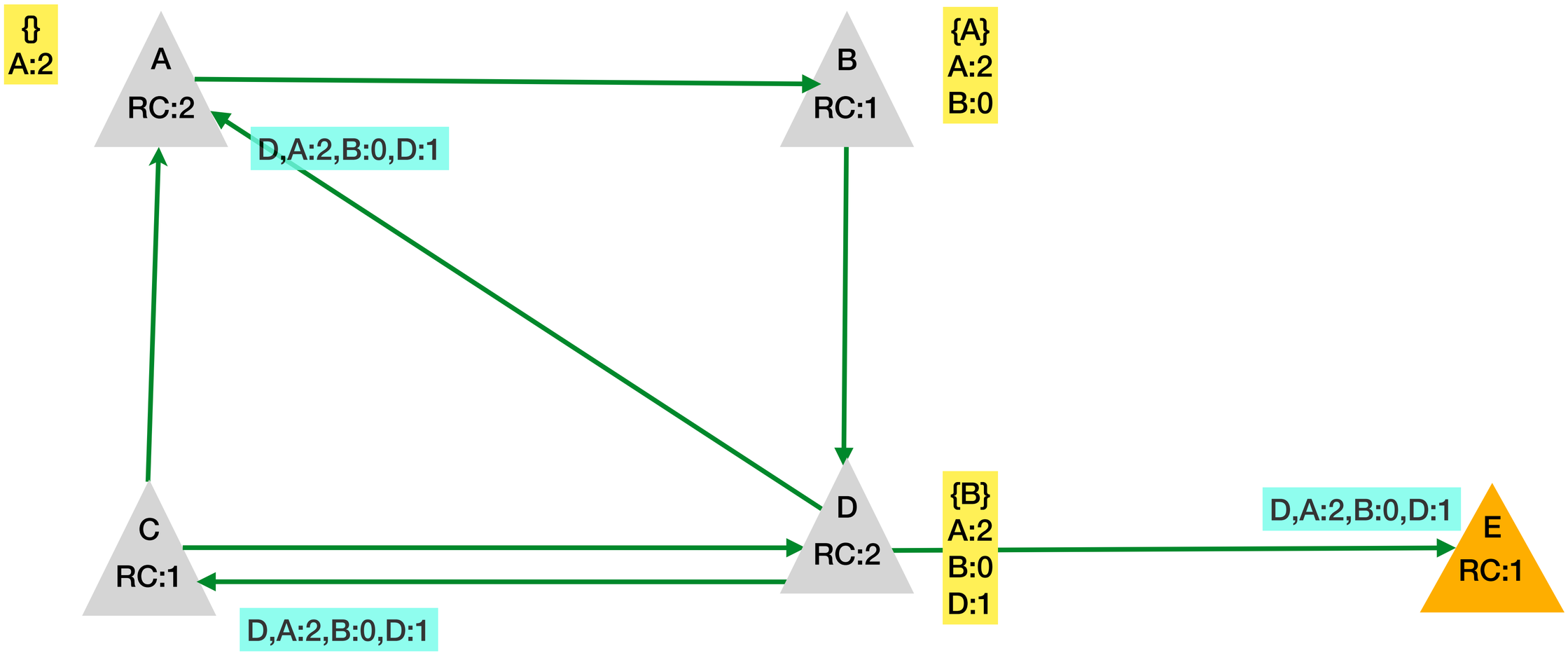

{}
A:2
A
RC:2
B
RC:1
C
RC:1
D
RC:2
E
RC:1

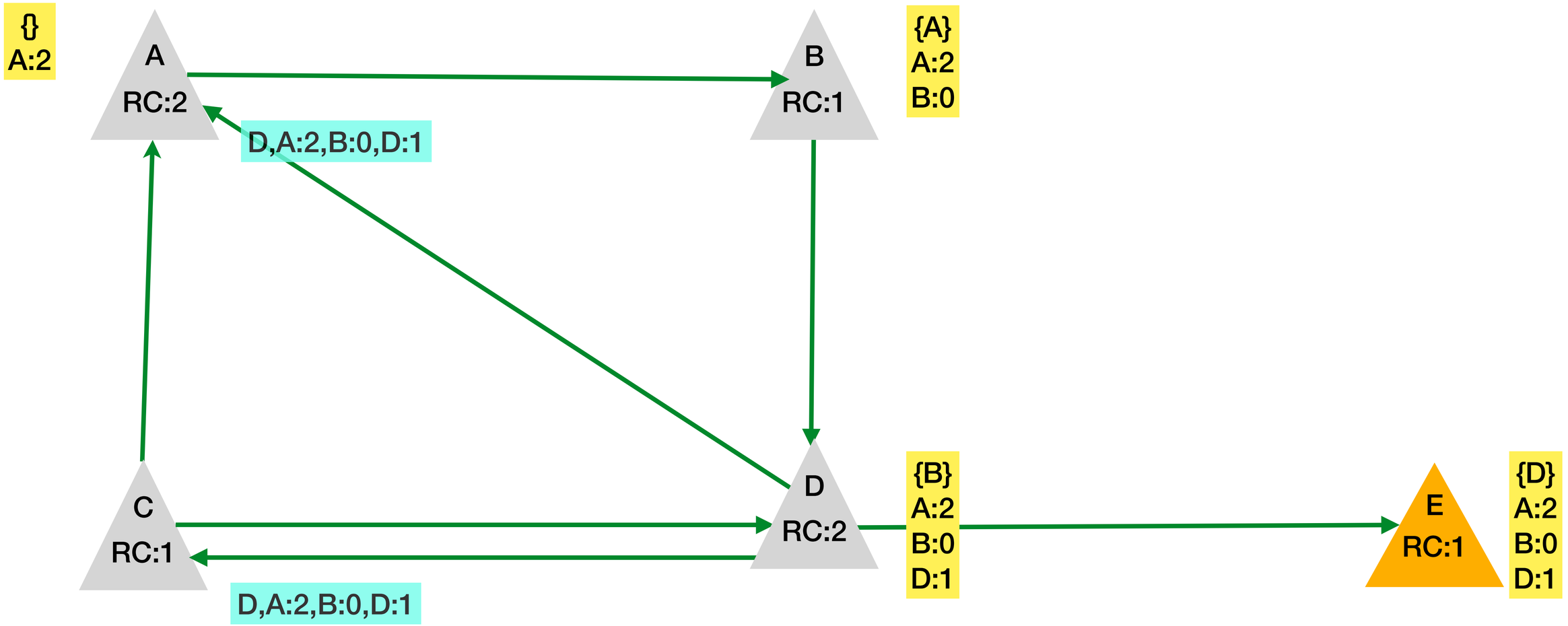

{}
A:2
A
RC:2
A, A:2
B
RC:1
C
RC:1
D
RC:2
E
RC:1

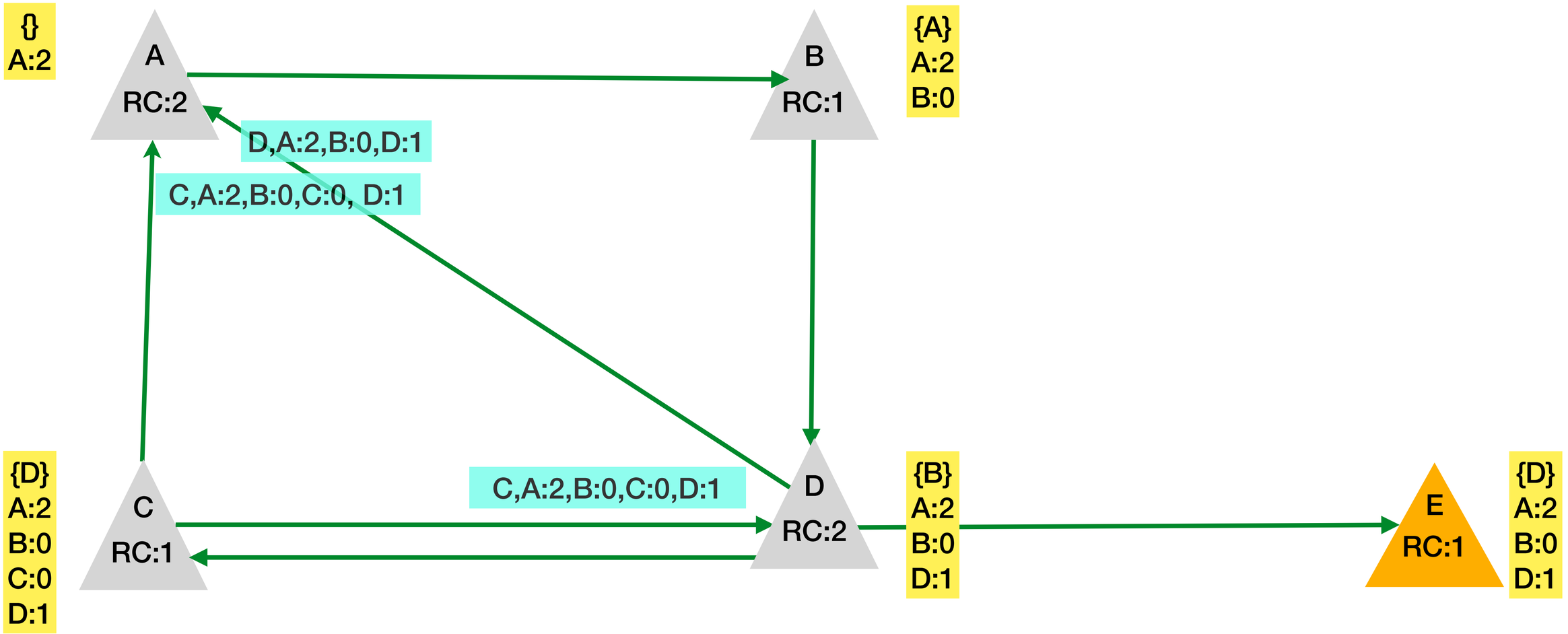

{}
A:2
A
RC:2
B
RC:1
{A}
A:2
B:0
C
RC:1
D
RC:2
E
RC:1

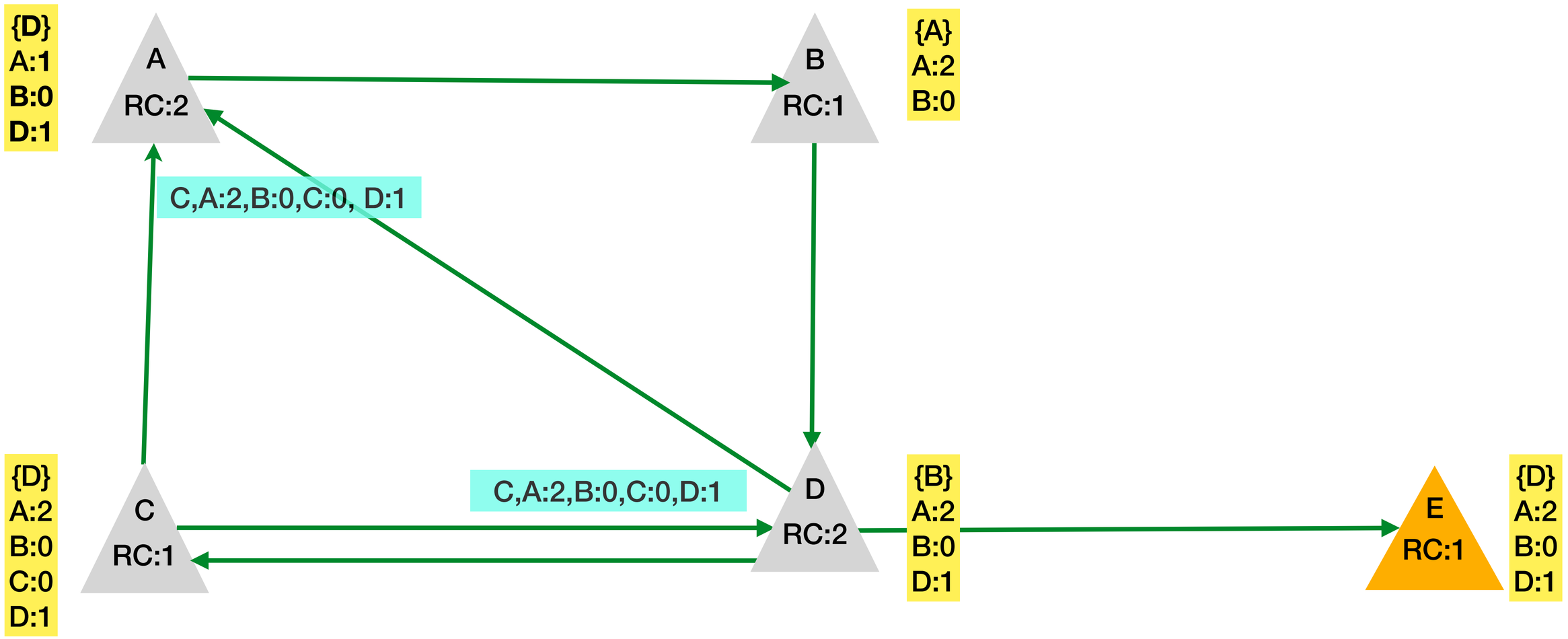

{}
A:2
A
RC:2
B
RC:1
{A}
A:2
B:0
B, A:2,B:0
C
RC:1
D
RC:2
E
RC:1

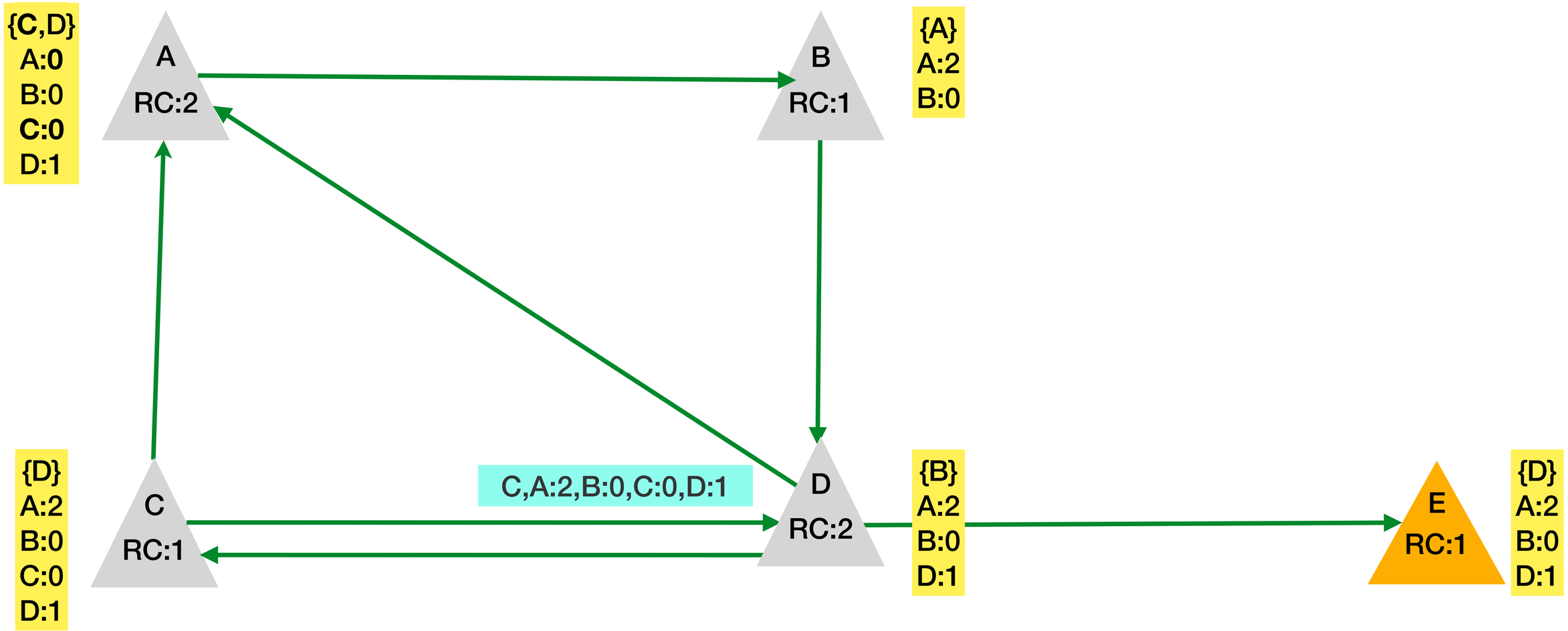

{}
A:2
A
RC:2
B
RC:1
{A}
A:2
B:0
C
RC:1
D
RC:2
{B}
A:2
B:0
D:1
E
RC:1

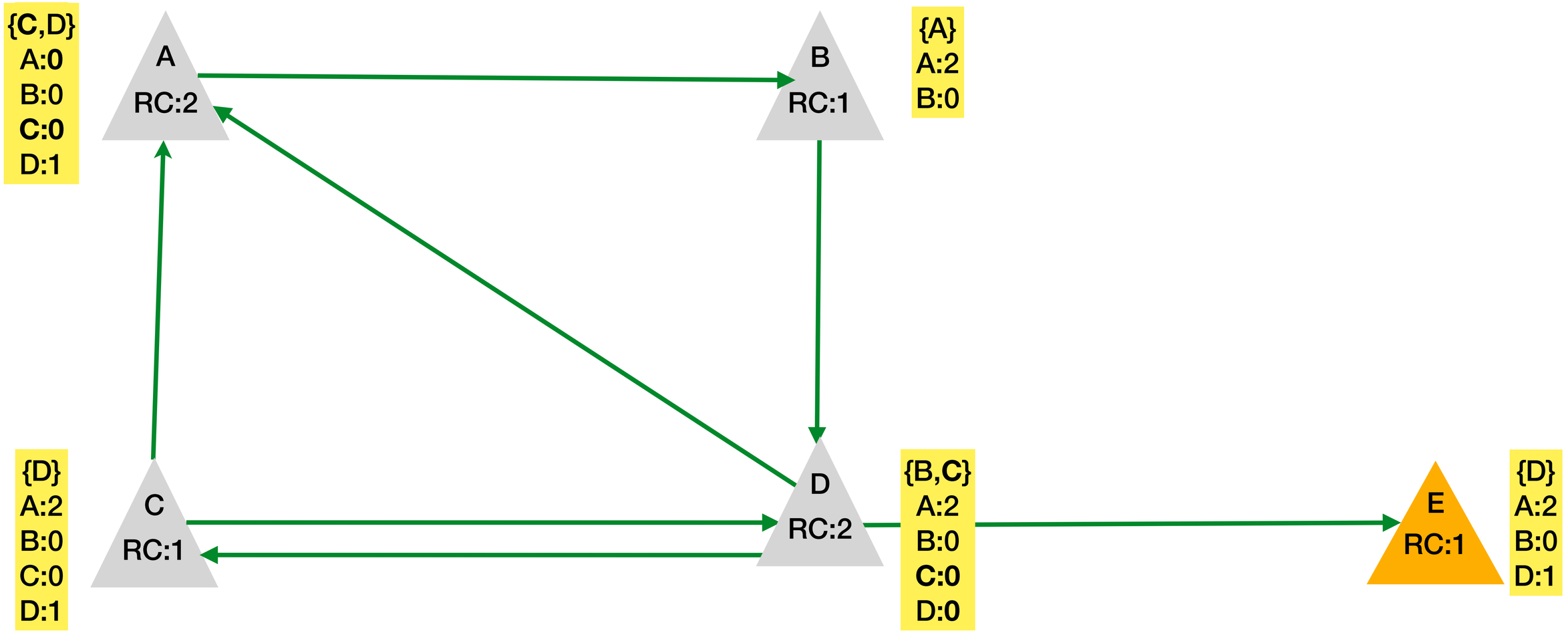

{}
A:2
A
RC:2
B
RC:1
{A}
A:2
B:0
D,A:2,B:0,D:1
C
RC:1
D
RC:2
{B}
A:2
B:0
D:1
D,A:2,B:0,D:1
E
RC:1
D,A:2,B:0,D:1


{}
A:2
A
RC:2
B
RC:1
{A}
A:2
B:0
D,A:2,B:0,D:1
C
RC:1
D
RC:2
{B}
A:2
B:0
D:1
E
RC:1
{D}
A:2
B:0
D:1
D,A:2,B:0,D:1

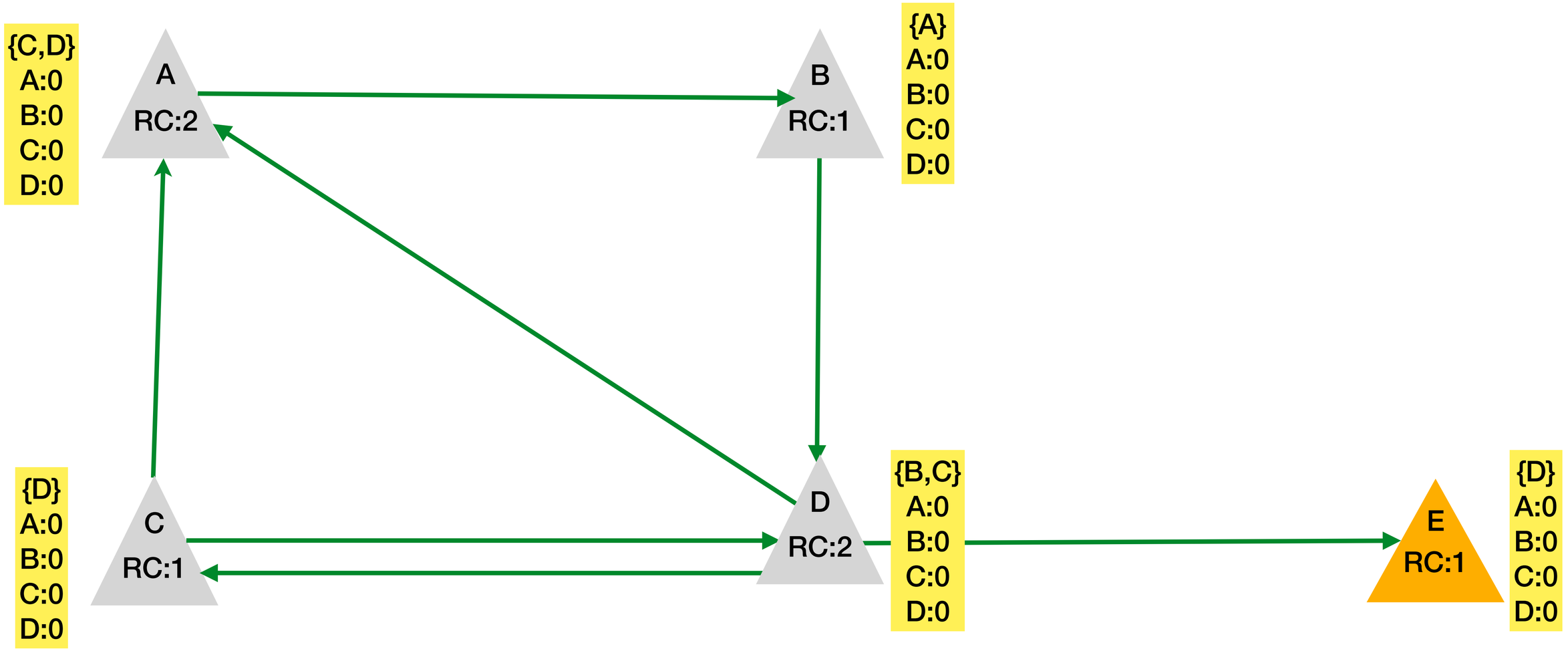

{}
A:2
A
RC:2
B
RC:1
{A}
A:2
B:0
D,A:2,B:0,D:1
C,A:2,B:0,C:0, D:1
{D}
A:2
B:0
C:0
D:1
C
RC:1
C,A:2,B:0,C:0,D:1
D
RC:2
{B}
A:2
B:0
D:1
E
RC:1
{D}
A:2
B:0
D:1


{D}
A:1
B:0
D:1
A
RC:2
B
RC:1
{A}
A:2
B:0
C,A:2,B:0,C:0, D:1
{D}
A:2
B:0
C:0
D:1
C
RC:1
C,A:2,B:0,C:0,D:1
D
RC:2
{B}
A:2
B:0
D:1
E
RC:1
{D}
A:2
B:0
D:1

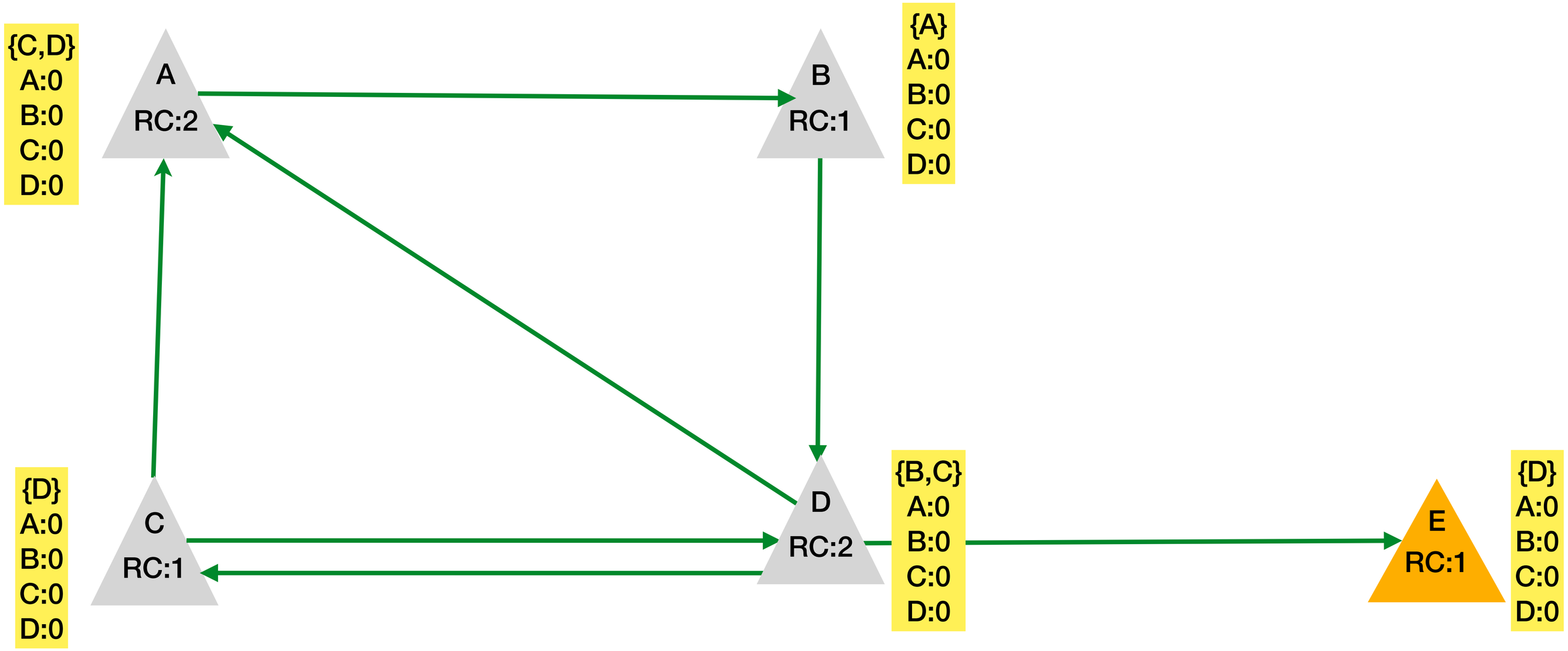

{C,D}
A:0
B:0
C:0
D:1
A
RC:2
B
RC:1
{A}
A:2
B:0
{D}
A:2
B:0
C:0
D:1
C
RC:1
C,A:2,B:0,C:0,D:1
D
RC:2
{B}
A:2
B:0
D:1
E
RC:1
{D}
A:2
B:0
D:1


{C,D}
A:0
B:0
C:0
D:1
A
RC:2
B
RC:1
{A}
A:2
B:0
{D}
A:2
B:0
C:0
D:1
C
RC:1
D
RC:2
{B,C}
A:2
B:0
C:0
D:0
E
RC:1
{D}
A:2
B:0
D:1

eventually ….

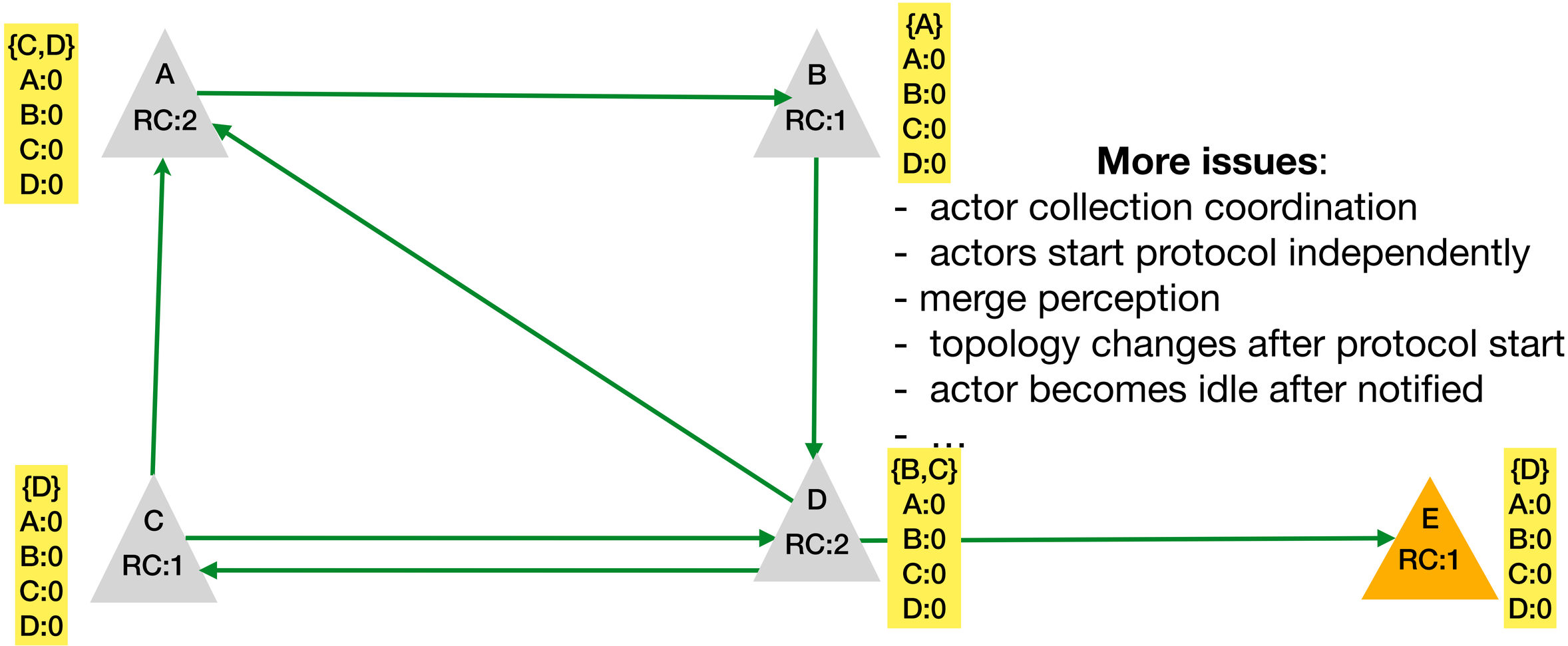

{C,D}
A:0
B:0
C:0
D:0
A
RC:2
B
RC:1
{A}
A:0
B:0
C:0
D:0
{D}
A:0
B:0
C:0
D:0
C
RC:1
D
RC:2
{B,C}
A:0
B:0
C:0
D:0
E
RC:1
{D}
A:0
B:0
C:0
D:0

eventually ….



… and now A, B, C, D can be collected

eventually ….



More issues:

- actor collection coordination
- actors start protocol independently
- merge perception
- topology changes after protocol start
- actor becomes idle after notified
- …

… and now A, B, C, D can be collected

Reflections on eMAC

Is that the whole story?

Is it useful?

Is it sound?

Is it complete?

Does separation play any role?

What are its drawbacks?

Reflections on eMAC

Is that the whole story?

Is it useful? Yes …

Is it sound?

Is it complete?

Does separation play any role?

What are its drawbacks?

Reflections on eMAC

Is that the whole story?

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete?

Does separation play any role?

What are its drawbacks?

Reflections on eMAC

Is that the whole story?

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete? **YES** — with proof sketch.

Does separation play any role?

What are its drawbacks?

Reflections on eMAC

Is that the whole story?

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete? **YES** — with proof sketch.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

Reflections on eMAC

Is that the whole story?

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete? **YES** — with proof sketch.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

~~CD is a bottleneck.~~

Reflections on eMAC

Is that the whole story?

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete? **YES** — with proof sketch.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

~~CD is a bottleneck.~~

~~CD performs a complex search on a large data structure.~~

Reflections on eMAC

Is that the whole story?

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete? **YES** — with proof sketch.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

~~CD is a bottleneck.~~

~~CD performs a complex search on a large data structure.~~

We need to “scan” every application message upon send/receive.

Reflections on eMAC

Is that the whole story? **No**, more problems needed solving.

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete? **YES** — with proof sketch.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

~~CD is a bottleneck.~~

~~CD performs a complex search on a large data structure.~~

We need to “scan” every application message upon send/receive.

Reflections on eMAC

Is that the whole story? **No**, more problems needed solving.

Is it useful? Yes …

Is it sound? **YES** — with proof sketch.

Is it complete? **YES** — with proof sketch.

Does separation play any role? **YES** — when scanning messages, and own heap.

What are its drawbacks?

~~CD is a bottleneck.~~

~~CD performs a complex search on a large data structure.~~

We need to “scan” every application message upon send/receive.

More and longer messages

Garbage Collection
for objects (Pony)

Soundness of a Concurrent Collector for Actors

Juliana Franco[1] Sylvan Clebsch[2]
Sophia Drossopoulou[1] Jan Vitek[3] Tobias Wrigstad[4]

[1] Imperial College, London [2] Microsoft Research Cambridge
[3] Northeastern University & CVUT [4] Uppsala University, Uppsala

Abstract ORCA is a garbage collection protocol for actor-based programs. Multiple actors may mutate the heap while the collector is run

ESOP'18

Orca: GC and Type System Co-Design for Actor Languages

SYLVAN CLEBSCH, Microsoft Research Cambridge, United Kingdom
JULIANA FRANCO, Imperial College London, United Kingdom
SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
ALBERT MINGKUN YANG, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
JAN VITEK, Northeastern University, United States of America

Orca is a concurrent and parallel garbage collector for actor programs, which does not require any stop-the-world steps, or synchronisation mechanisms, and which has been designed to support zero-copy message passing and sharing of mutable data. Orca is part of the runtime of the actor-based language Pony. Pony's runtime was co-designed with the Pony language. This co-design allowed us to exploit certain language

OOPSLA'17

Garbage Collection
for objects (Pony)

ORCA:
Ownership and
Reference Counting based
Garbage Collection in the
Actor World

Soundness of a Concurrent Collector for Actors

Juliana Franco[1] Sylvan Clebsch[2]
Sophia Drossopoulou[1] Jan Vitek[3] Tobias Wrigstad[4]

[1] Imperial College, London [2] Microsoft Research Cambridge
[3] Northeastern University & CVUT [4] Uppsala University, Uppsala

Abstract ORCA is a garbage collection protocol for actor-based pro-

ESOP'18

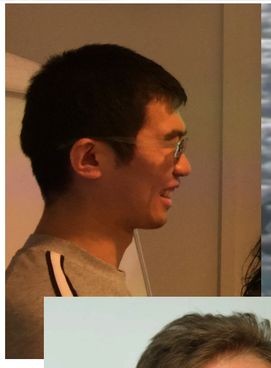
Orca: GC and Type System Co-Design for Actor Languages

SYLVAN CLEBSCH, Microsoft Research Cambridge, United Kingdom
JULIANA FRANCO, Imperial College London, United Kingdom
SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
ALBERT MINGKUN YANG, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
JAN VITEK, Northeastern University, United States of America

Orca is a concurrent and parallel garbage collector for actor programs, which does not require any stop-the-world steps, or synchronisation mechanisms, and which has been designed to support zero-copy message passing and sharing of mutable data. Orca is part of the runtime of the actor-based language Pony. Pony's runtime was co-designed with the Pony language. This co-design allowed us to exploit certain language

OOPSLA'17

Garbage Collection
for objects (Pony)



ORCA:
Ownership and
Reference Counting based
Garbage **C**ollection in the
Actor World

Soundness of a Concurrent Collector for Actors

Juliana Franco[1] Sylvan Clebsch[2]
Sophia Drossopoulou[1] Jan Vitek[3] Tobias Wrigstad[4]

[1] Imperial College, London [2] Microsoft Research Cambridge
[3] Northeastern University & CVUT [4] Uppsala University, Uppsala

Abstract ORCA is a garbage collection protocol for actor-based programs. Multiple actors may mutate the heap while the collector is run

ESOP'18

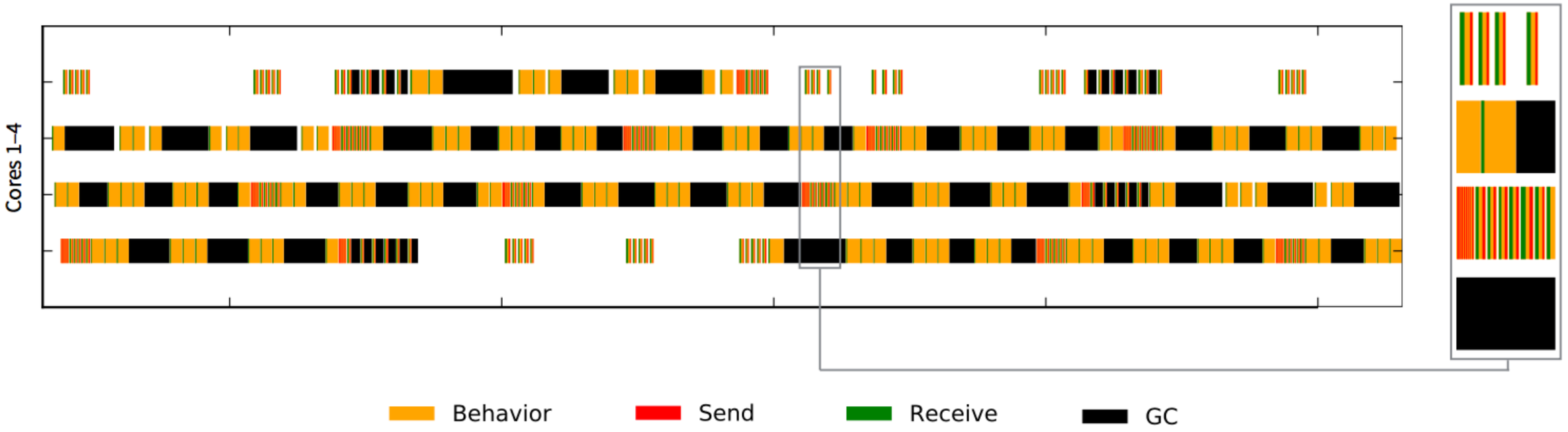
Orca: GC and Type System Co-Design for Actor Languages

SYLVAN CLEBSCH, Microsoft Research Cambridge, United Kingdom
JULIANA FRANCO, Imperial College London, United Kingdom
SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
ALBERT MINGKUN YANG, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
JAN VITEK, Northeastern University, United States of America

Orca is a concurrent and parallel garbage collector for actor programs, which does not require any stop-the-world steps, or synchronisation mechanisms, and which has been designed to support zero-copy message passing and sharing of mutable data. Orca is part of the runtime of the actor-based language Pony. Pony's runtime was co-designed with the Pony language. This co-design allowed us to exploit certain language

OOPSLA'17

ORCA is *fully concurrent*
ie no synchronization, lo locks, no barrier, no stop the world step.



GC & Concurrency Challenges

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

Challenge_2: How avoid data races between GC and mutators?

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

The allocating actor (owner)

Challenge_2: How avoid data races between GC and mutators?

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

The allocating actor (owner)

Challenge_2: How avoid data races between GC and mutators?

Type System

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

The allocating actor (owner)

Challenge_2: How avoid data races between GC and mutators?

Type System

Challenge_3: How does the “owner” know whether there are foreign references to its owned objects?

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

The allocating actor (owner)

Challenge_2: How avoid data races between GC and mutators?

Type System

Challenge_3: How does the “owner” know whether there are foreign references to its owned objects?

use INV; Deferred Reference Counts and Messaging Mechanism

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

The allocating actor (owner)

Challenge_2: How avoid data races between GC and mutators?

Type System

Challenge_3: How does the “owner” know whether there are foreign references to its owned objects?

use INV; Deferred Reference Counts and Messaging Mechanism

Challenge_4: How deal with uncertainty in message delivery

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

The allocating actor (owner)

Challenge_2: How avoid data races between GC and mutators?

Type System

Challenge_3: How does the “owner” know whether there are foreign references to its owned objects?

use INV; Deferred Reference Counts and Messaging Mechanism

Challenge_4: How deal with uncertainty in message delivery

rely on Causal Message Delivery

GC & Concurrency Challenges

Challenge_1: Who collects the objects?

The allocating actor (owner)

Challenge_2: How avoid data races between GC and mutators?

Type System

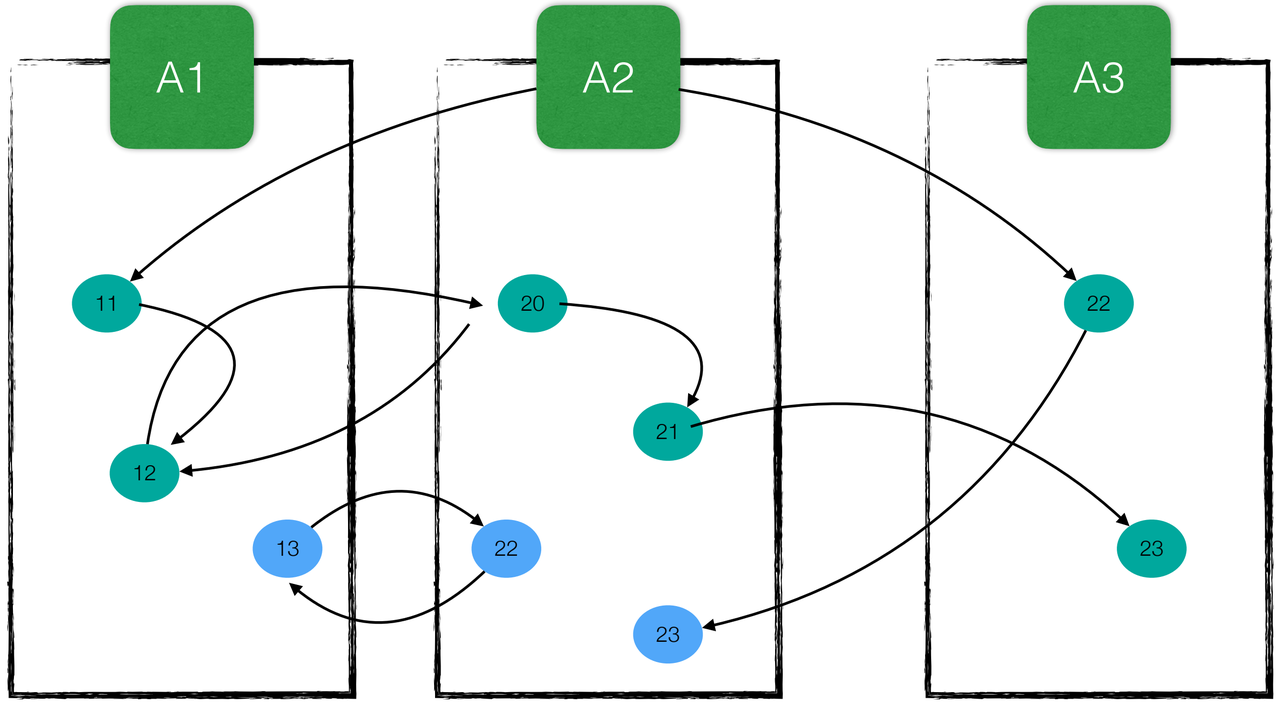
Challenge_3: How does the “owner” know whether there are foreign references to its owned objects?

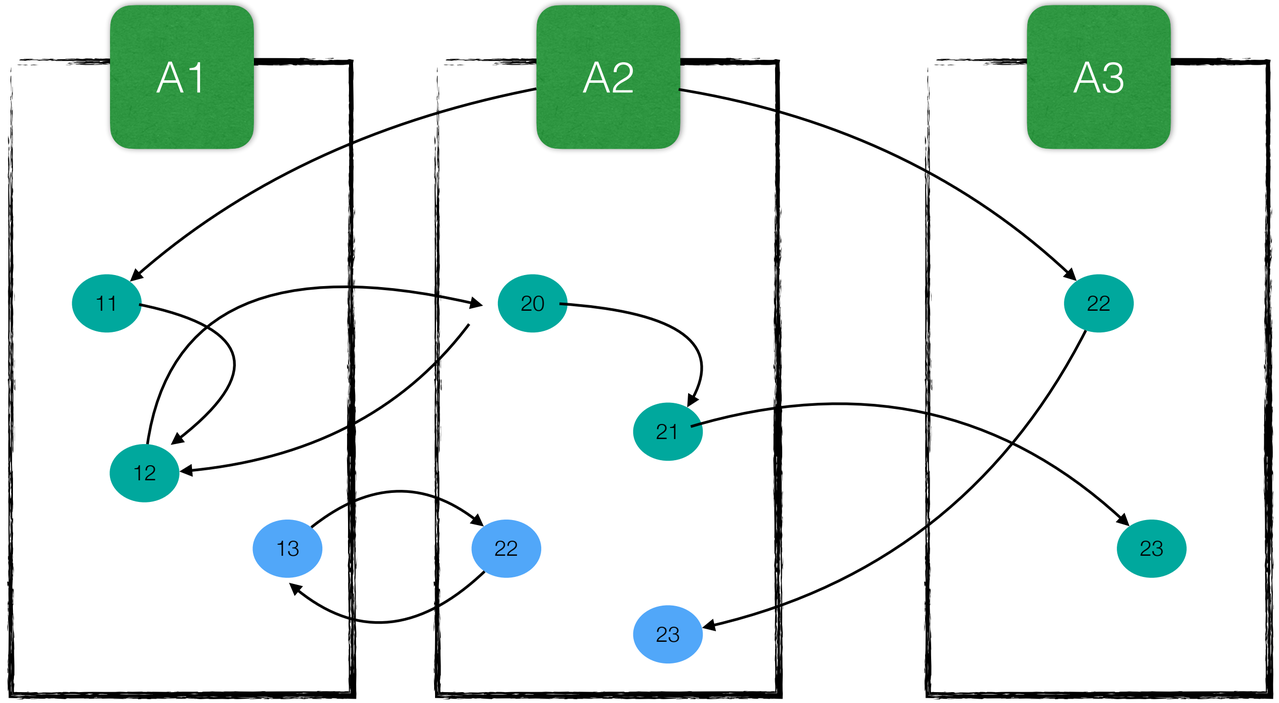
use INV; Deferred Reference Counts and Messaging Mechanism

Challenge_4: How deal with uncertainty in message delivery

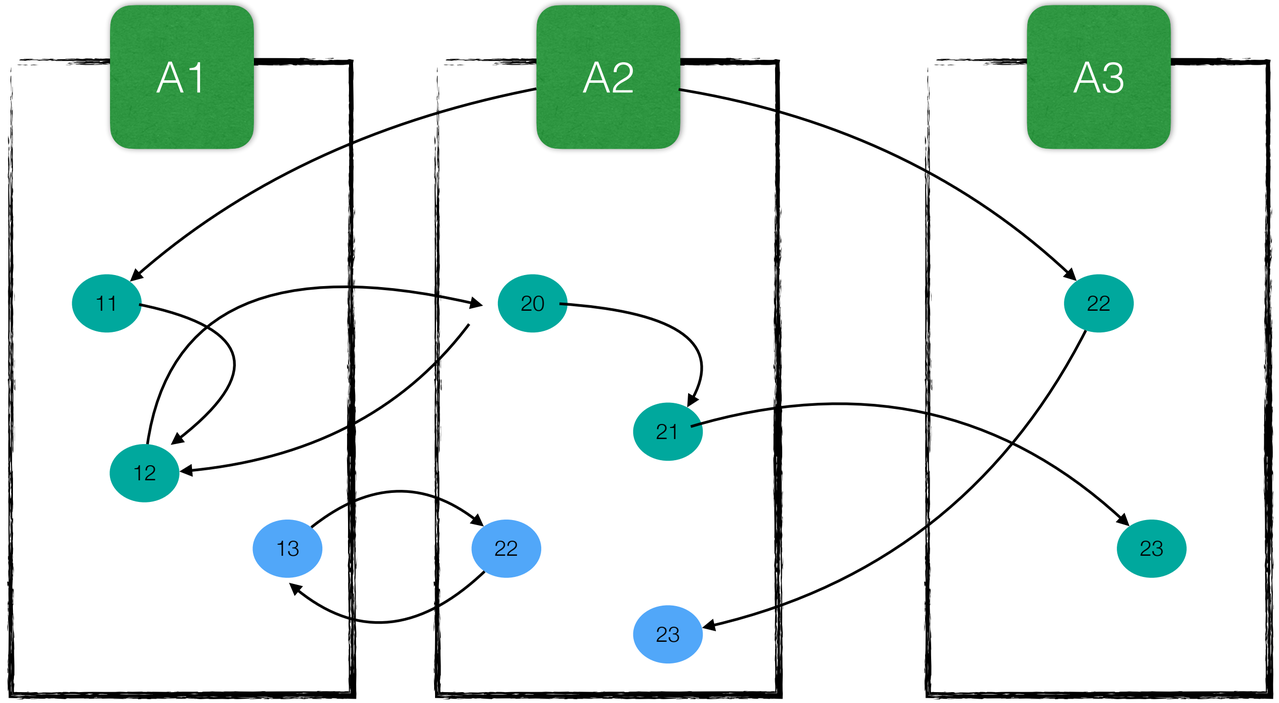
rely on Causal Message Delivery

Tight Connection between Language and Runtime Design


A1
A2
A3
11
20
22
21
12
13
22
23
23

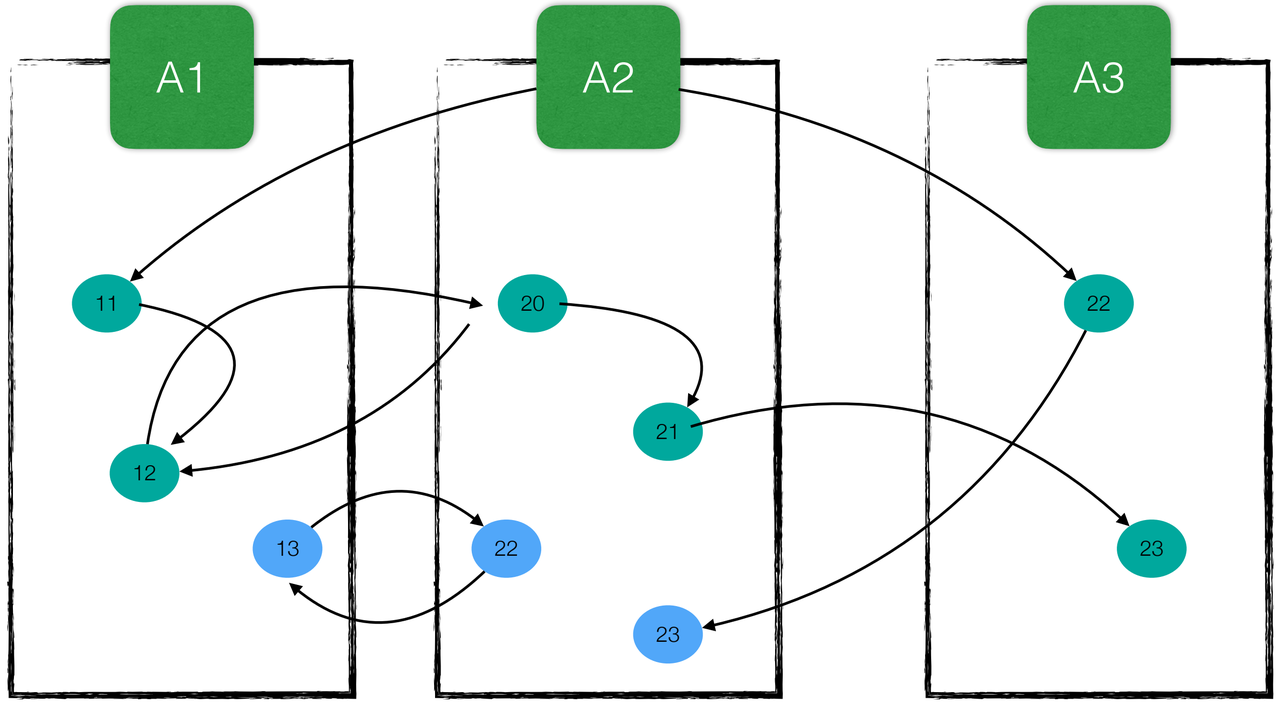


Challenges



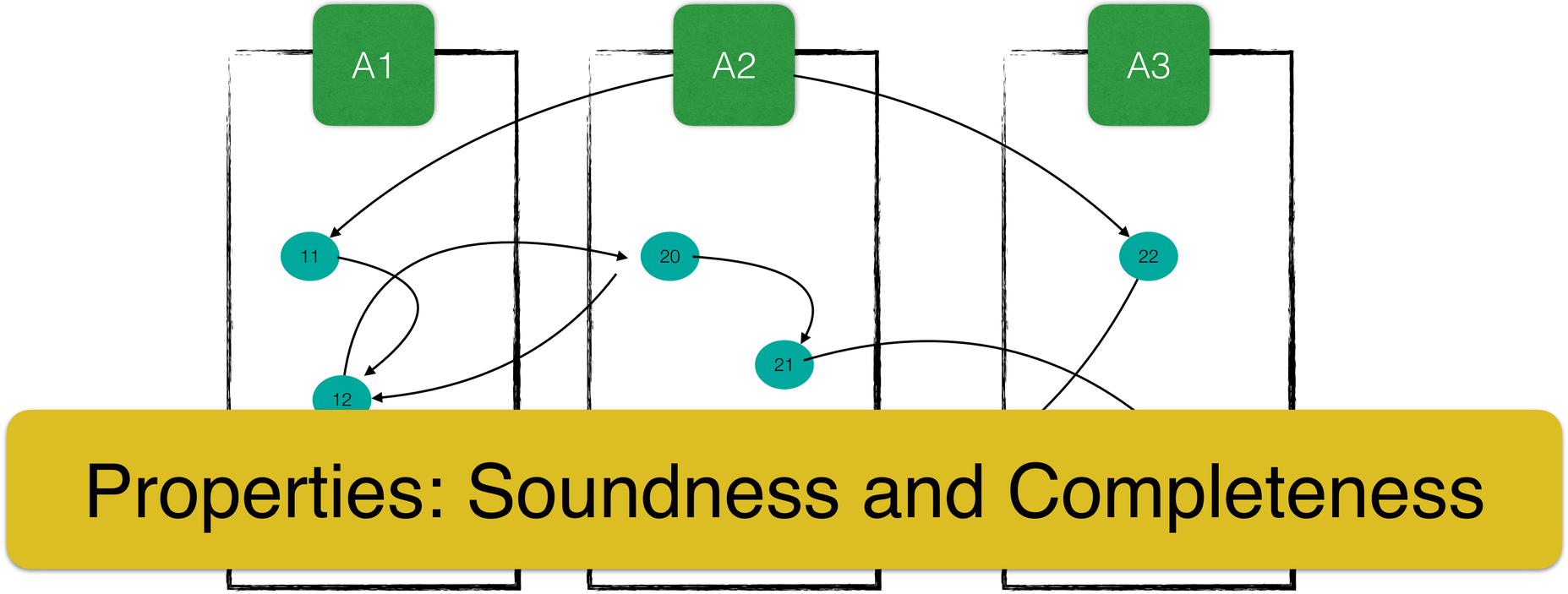
Challenges

- Owning actor might not have path to its live objects
- Cycles in object graph



Approach

- Owning actor keeps upper bound on number of actors which have a path to owned object
- Owning actor collects object when this number=0
- Foreign actor keeps count of references to un-owned objects
- Foreign actor informs owning actor when number of references to unowned objects changes (ie upon message send/receive or local tracing)

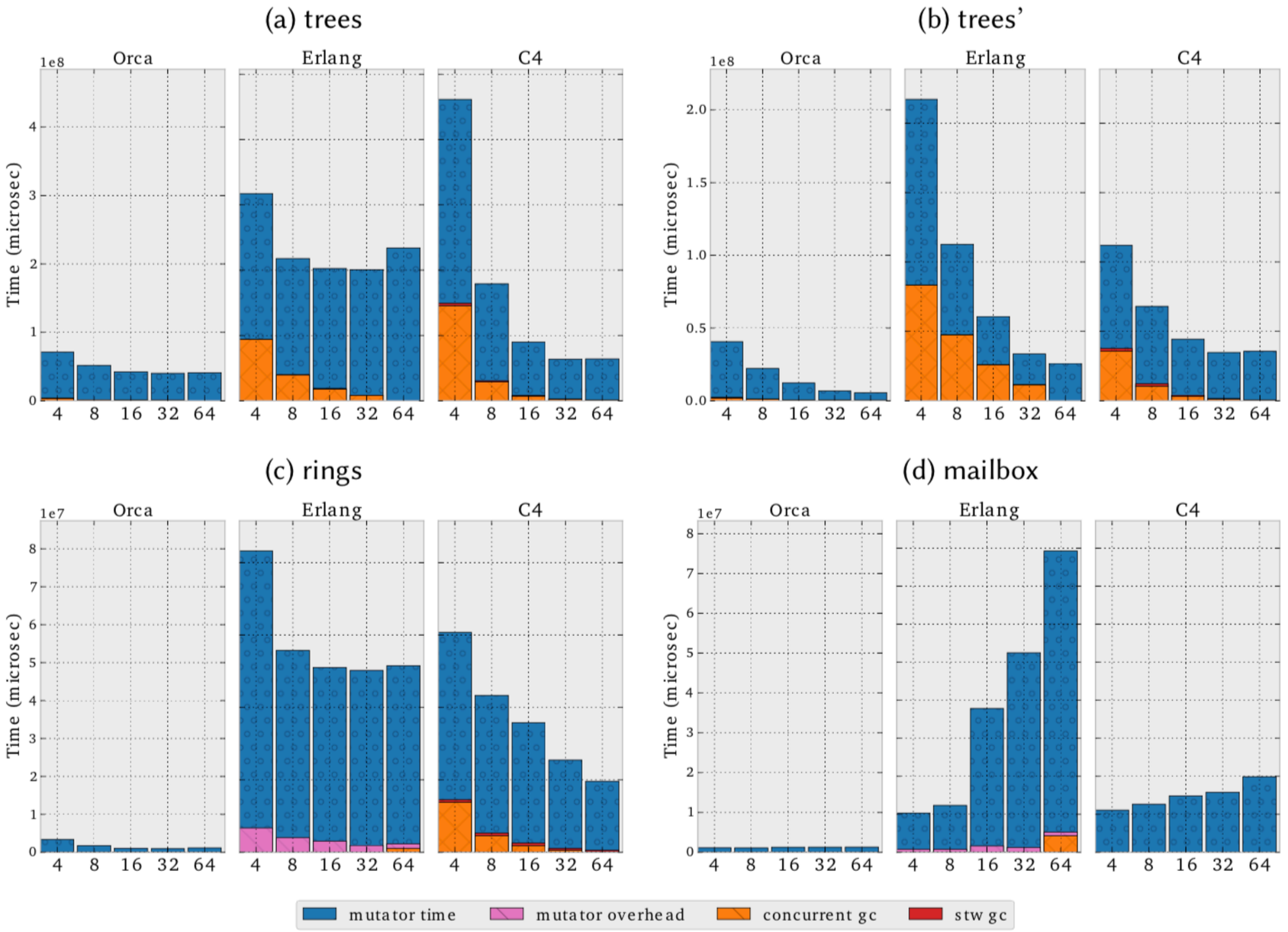


Properties: Soundness and Completeness

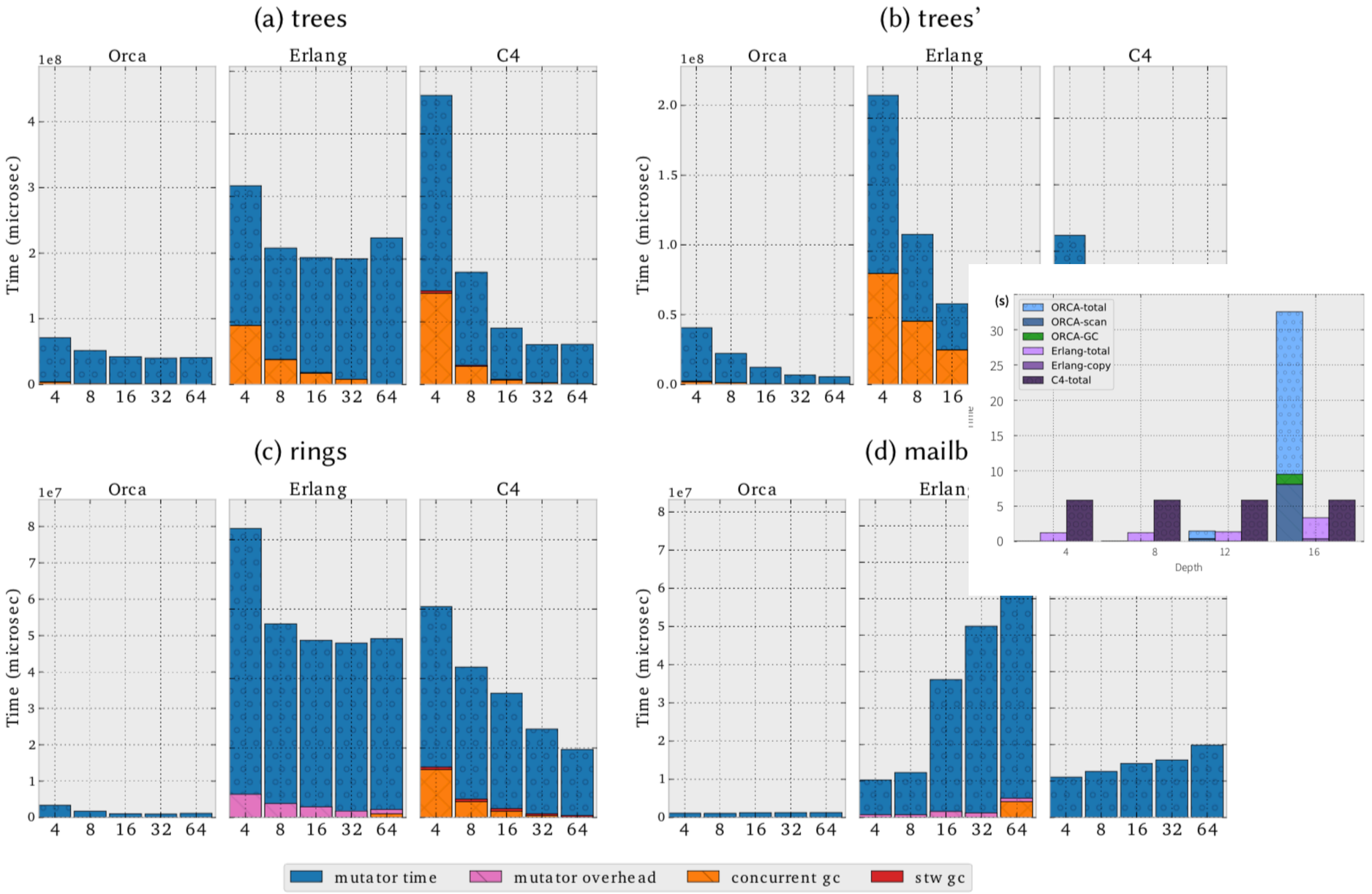
Approach

- Owning actor keeps upper bound on number of actors which have a path to owned object
- Owning actor collects object when this number=0
- Foreign actor keeps count of references to un-owned objects
- Foreign actor informs owning actor when number of references to unowned objects changes (ie upon message send/receive or local tracing)

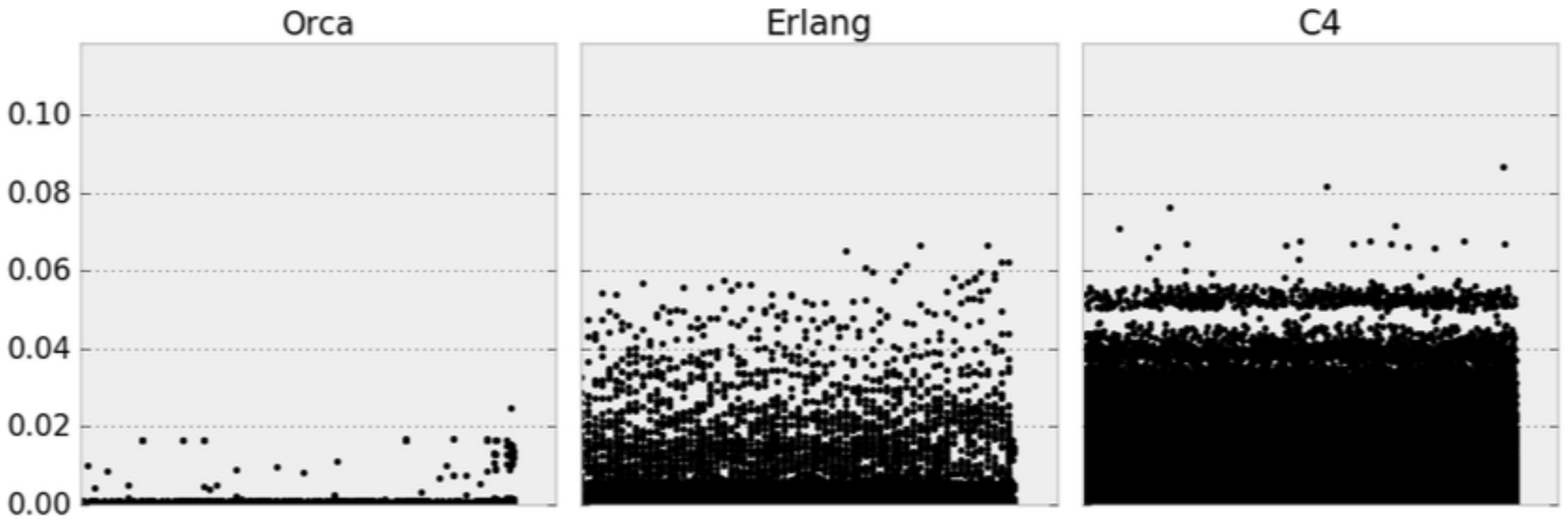
Pony vs Erlang vs Java



Pony vs Erlang vs Java



Responsiveness



Reflections on MAC

Is it useful?

Is it sound?

Is it complete?

What are its advantages/drawbacks?

Reflections on MAC

Is it useful? **YES**

Is it sound?

Is it complete?

What are its advantages/drawbacks?

Reflections on MAC

Is it useful? **YES**

Is it sound? **YES** — latex proof.

Is it complete?

What are its advantages/drawbacks?

Reflections on MAC

Is it useful? **YES**

Is it sound? **YES** — latex proof.

Is it complete? **YES** — latex proof.

What are its advantages/drawbacks?

Reflections on MAC

Is it useful? **YES**

Is it sound? **YES** — latex proof.

Is it complete? **YES** — latex proof.

What are its advantages/drawbacks?

(+) No central authority.

Reflections on MAC

Is it useful? **YES**

Is it sound? **YES** — latex proof.

Is it complete? **YES** — latex proof.

What are its advantages/drawbacks?

(+) No central authority.

(+) No complex algorithm on large data structure

Reflections on MAC

Is it useful? **YES**

Is it sound? **YES** — latex proof.

Is it complete? **YES** — latex proof.

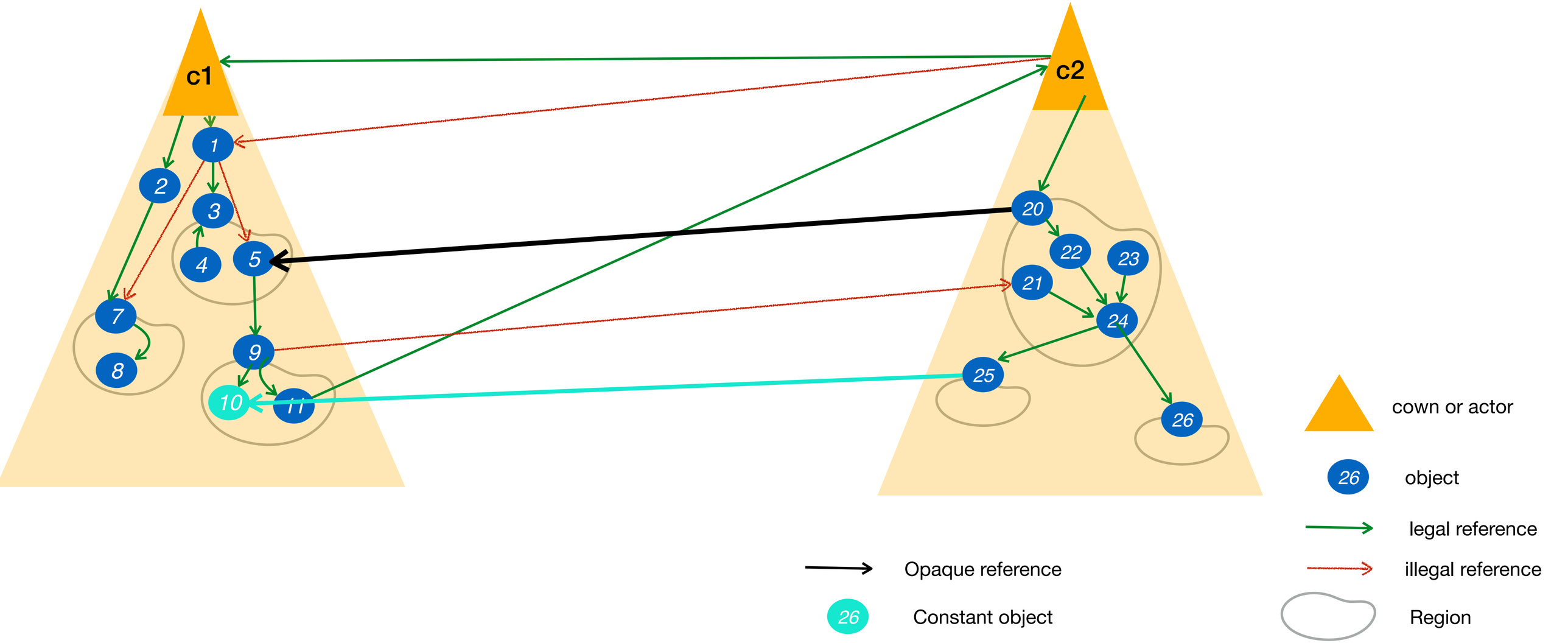
What are its advantages/drawbacks?

(+) No central authority.

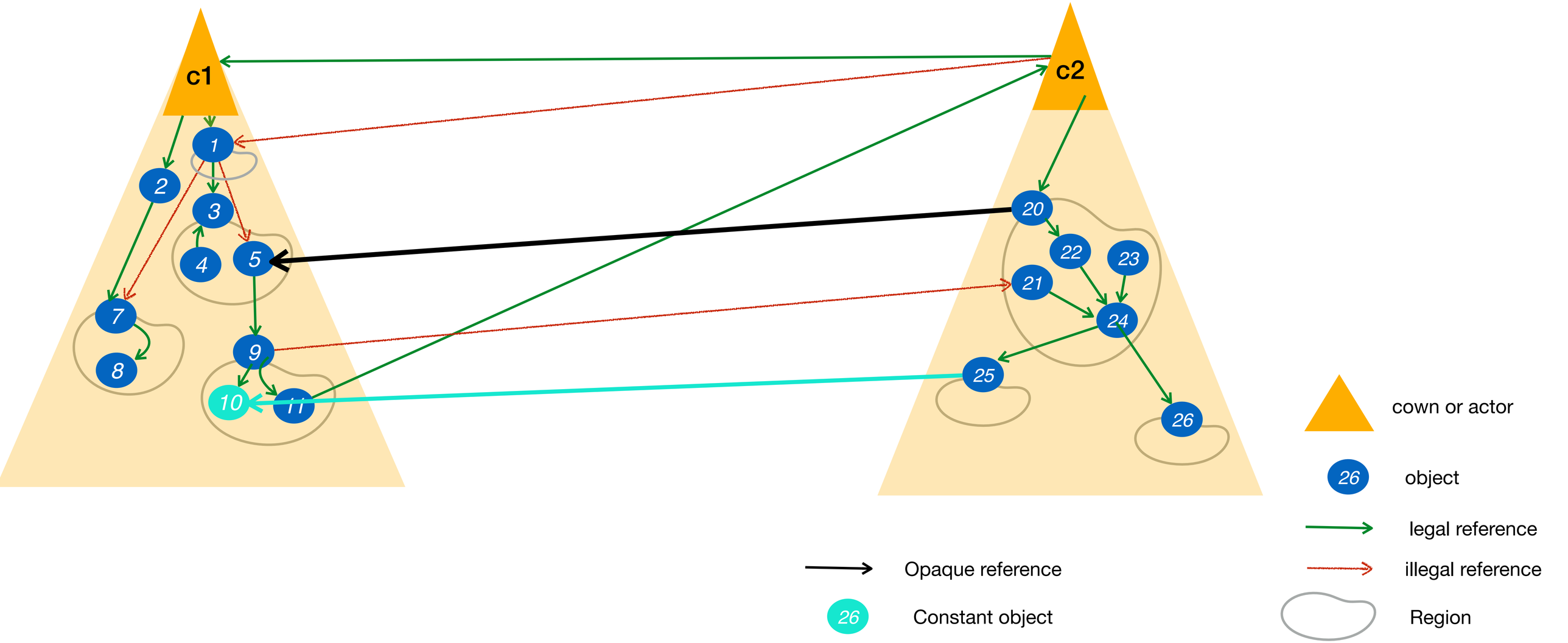
(+) No complex algorithm on large data structure

(-) We need to “scan” every application message upon send/receive.

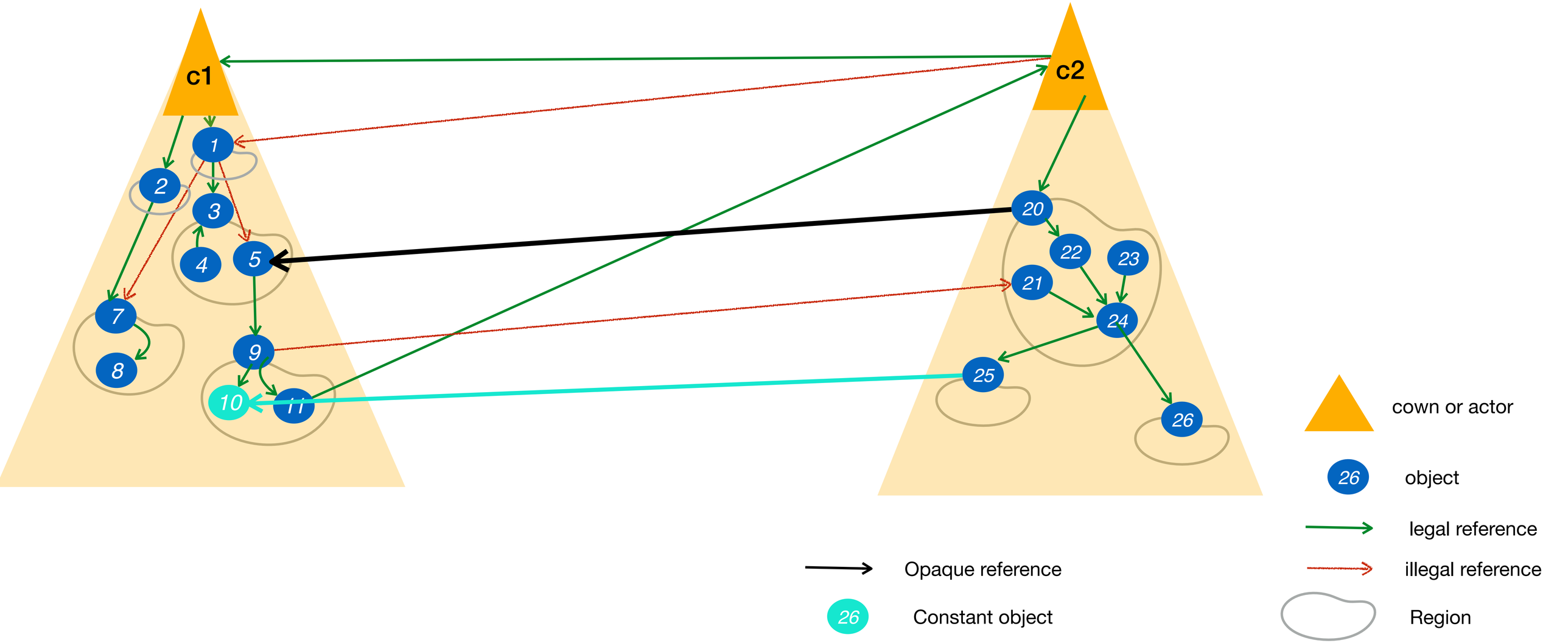
Pony Actors share state safely - more truth



Pony Actors share state safely - more truth



Pony Actors share state safely - more truth



Types and Garbage Collection

for objects (Verona)