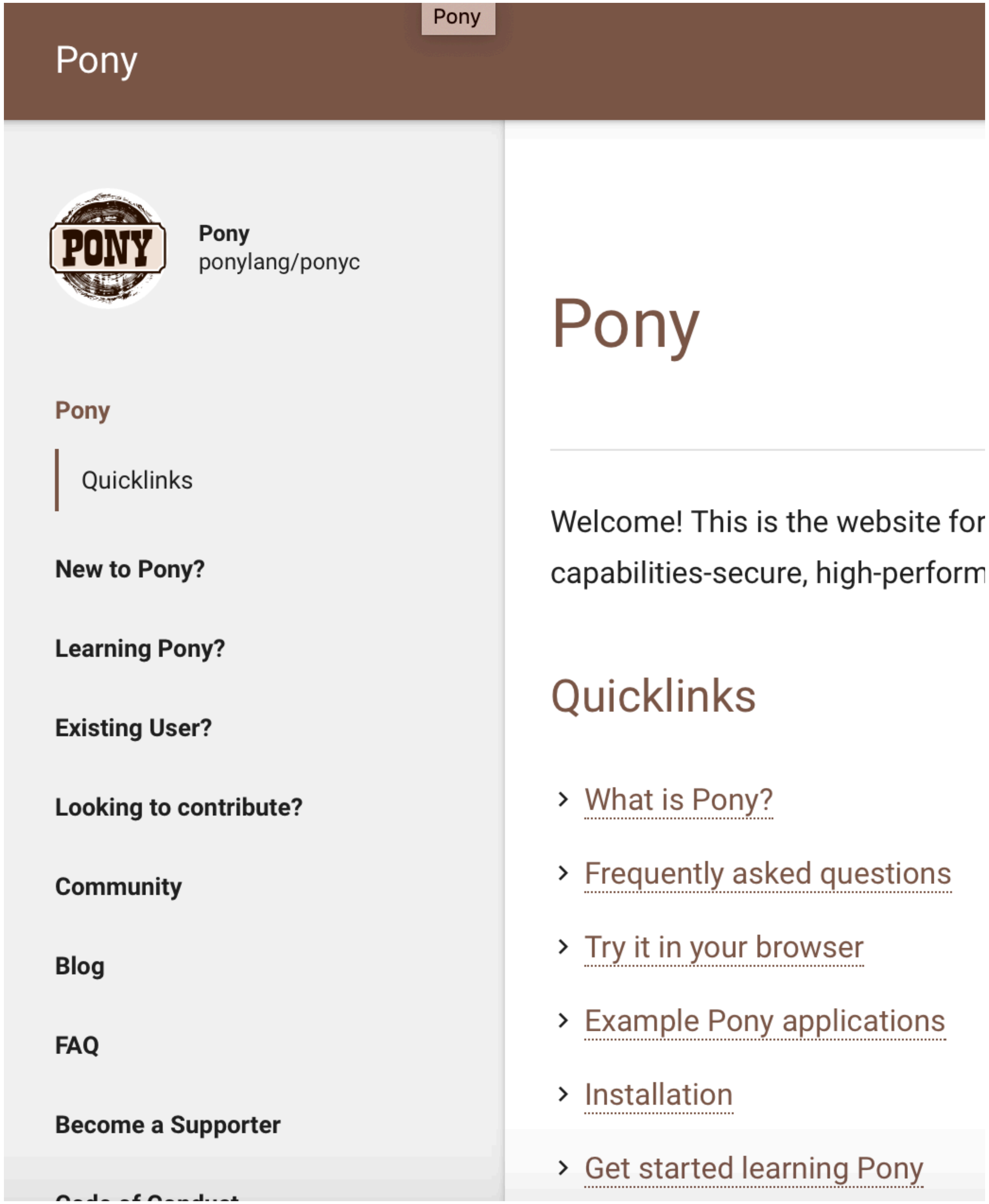
GC in the middle of behaviour
send
local
foreign
mutable
lrc++ trace
frc-- trace
immutable
lrc++ trace
frc-- trace
was mutable now immutable this.imm(o)=true
lrc++
trace
frc--
trace
?
IMM
recv
local
foreign
mutable
lrc-- trace
frc++ trace
immutable
lrc-- trace
frc++ trace
o: unshared → α traces o
passed → α does not trace o
shared → α traces o
OR o: traced by only one actor in mark phase
reachable
local
foreign
mutable
mark + trace
mark + trace
immutable
mark + trace
mark
trace
unreachable ∧ rc=0
local
foreign
mutable
collect
—
immutable
collect
—
unreachable ∧ rc>0
local
foreign
mutable
mark
DEC
immutable
mark + trace [as share]
DEC
ORCA −
ORCA +
root +
roots = {o | O(o) ∈ this ∧ o.rc>0} ∪ val
α₁
α₂
α₃

Conclu
mutable
lrc++ trace
frc-- trace
immutable
lrc++ trace
frc-- trace
lrc++
frc--
0 : un
shared → α traces 0
one actor in mark phase



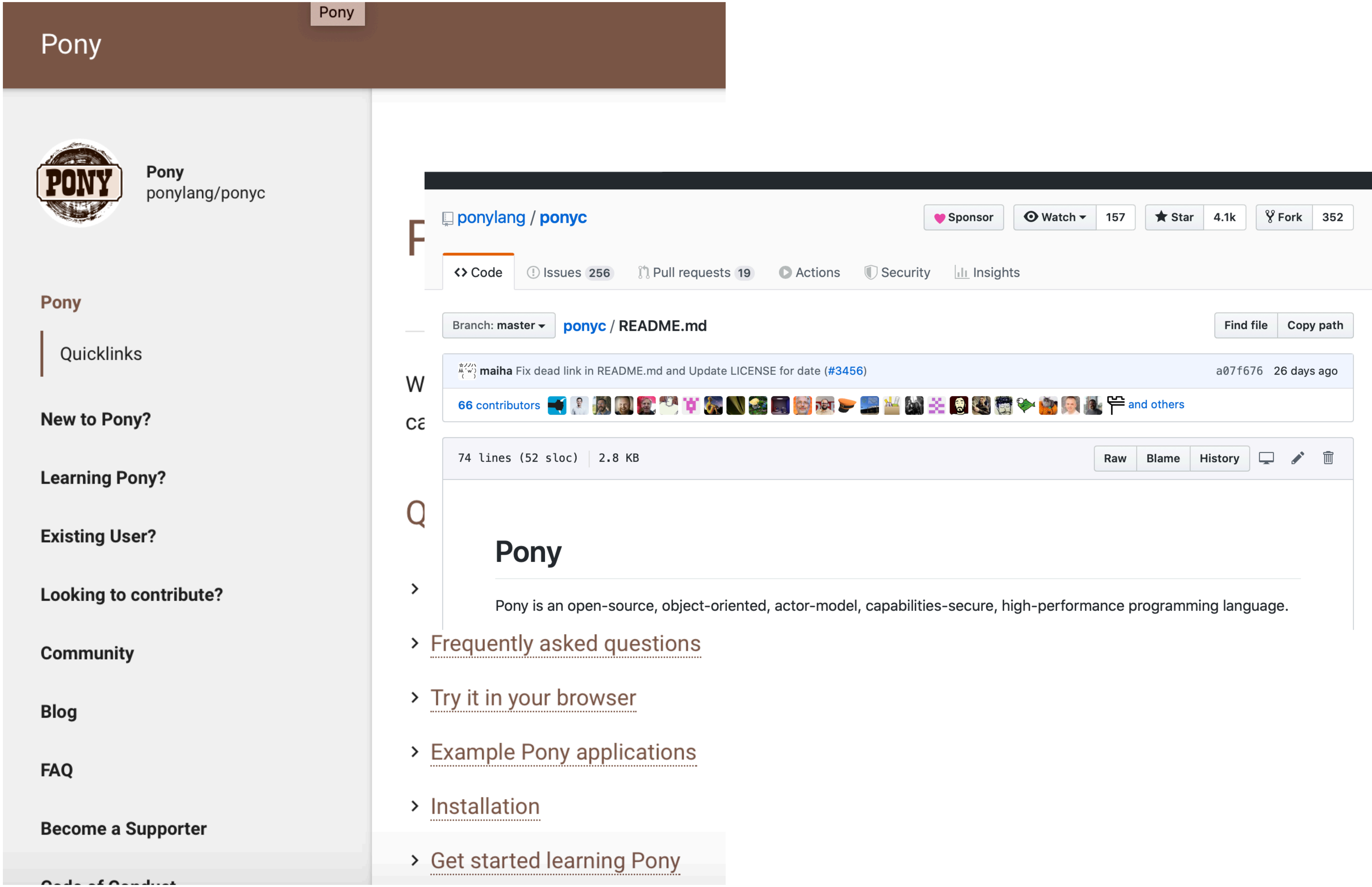
Pony

Welcome! This is the website for
capabilities-secure, high-perform

Quicklinks

- What is Pony?
- Frequently asked questions
- Try it in your browser
- Example Pony applications
- Installation
- Get started learning Pony

Today's examples at https://github.com/sophiaIC/SimplePonyPrograms.git



maiha Fix dead link in README.md and Update LICENSE for date (#3456) a07f676 26 days ago

74 lines (52 sloc) 2.8 KB

Pony

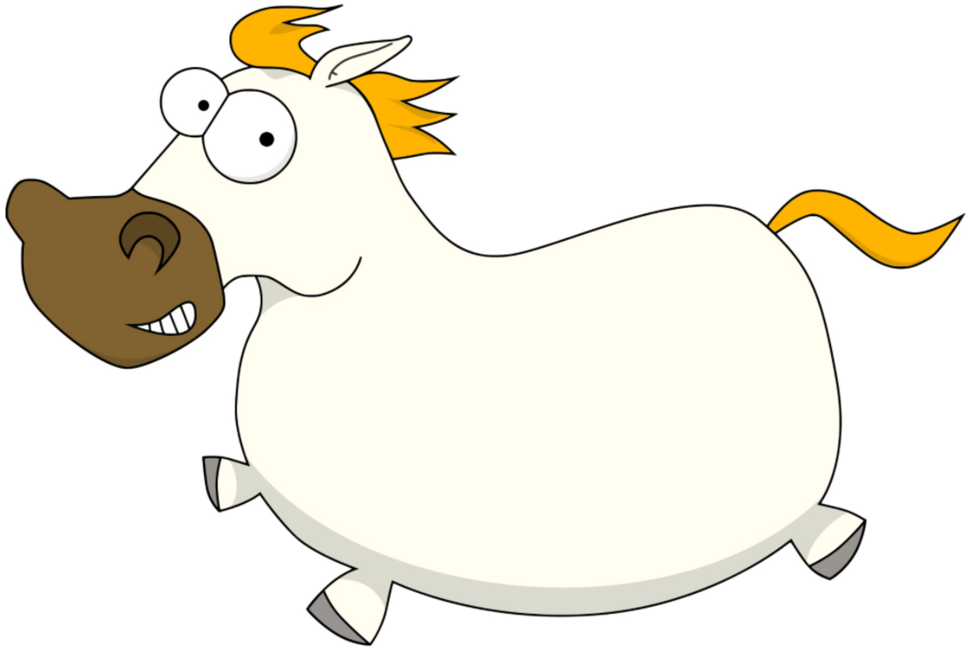
Pony is an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language.

Playground at **http://playground.ponylang.org**

Today's examples at https://github.com/sophiaIC/SimplePonyPrograms.git

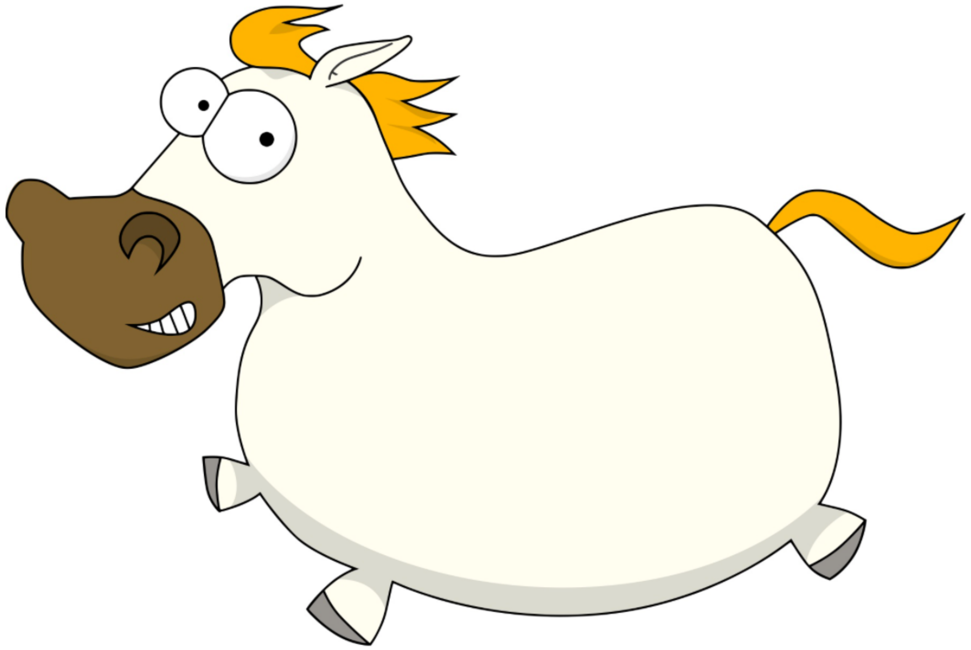
Pony - Goals

- concurrent (and distributed) programming,
- efficient,
- easy to write correct code.



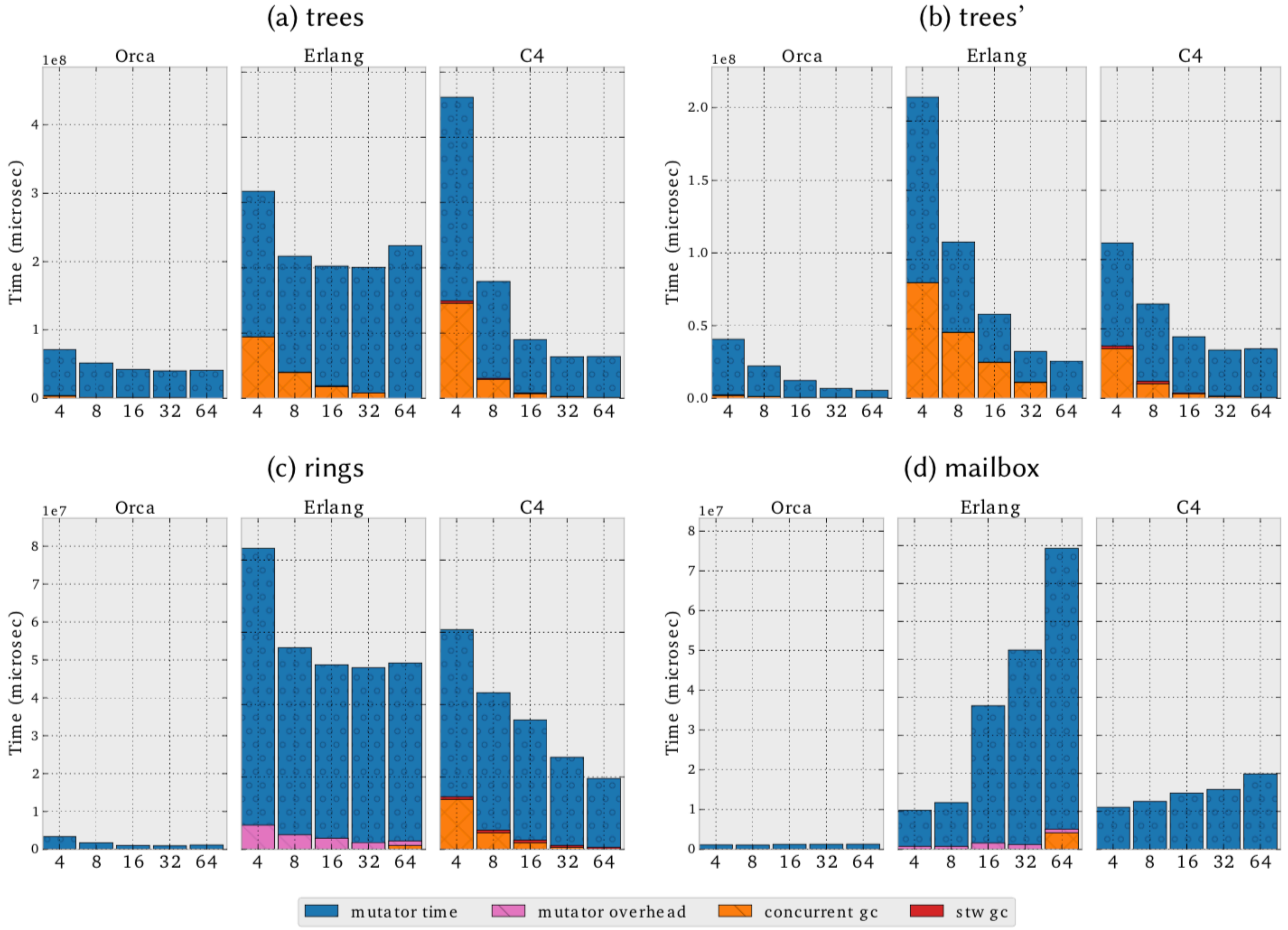
Pony - How

- concurrent (and distributed) programming,
    - actors first
- efficient,
    - no locks
    - sharing without copies
- easy to write correct code.
    - data race free,
    - deadlock - free,
    - safe object cycles,
    - figment of atomicity,
    - causality

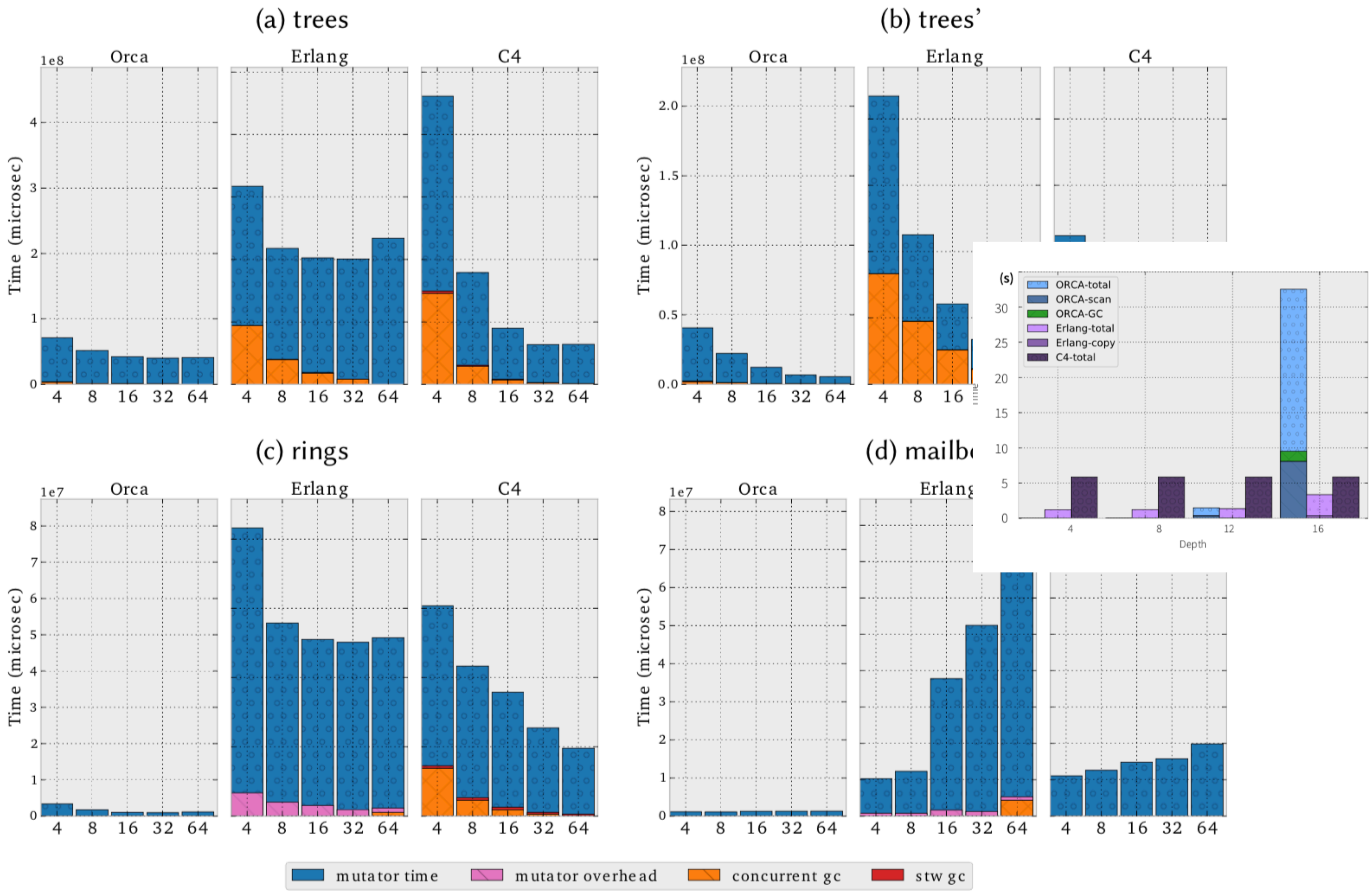


efficient?

Pony vs Erlang vs Java



Pony vs Erlang vs Java



Pony vs Akka vs CAF

Run, Actor, Run

Towards Cross-Actor Language Benchmarking

Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

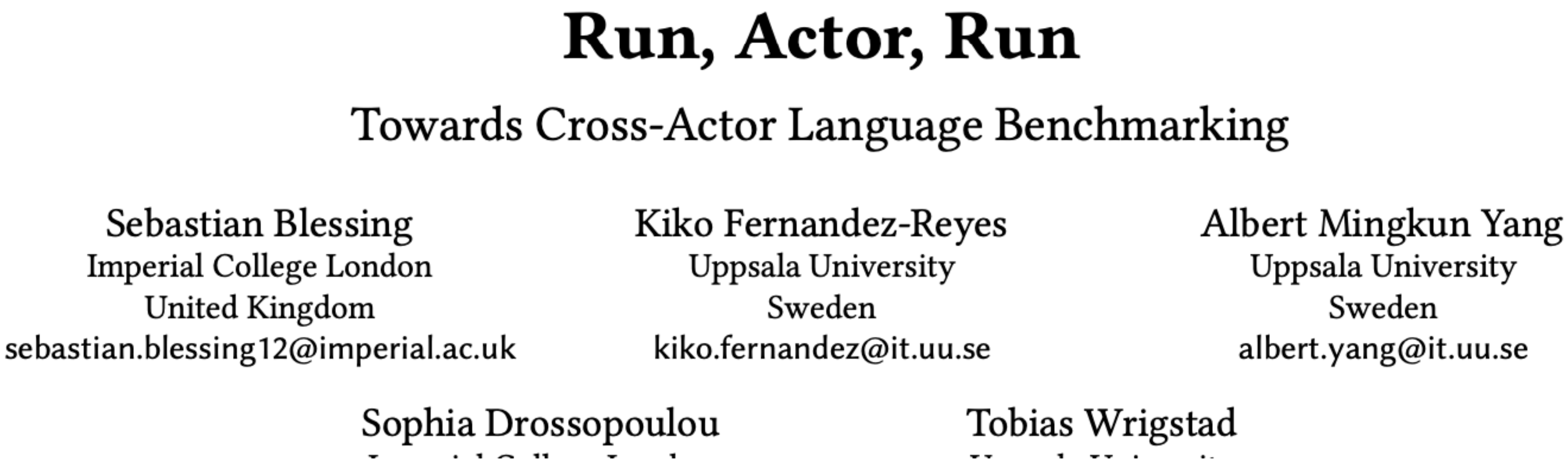
Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Sophia Drossopoulou

Tobias Wrigstad

Agere 2019

Pony vs Akka vs CAF



Run, Actor, Run

Towards Cross-Actor Language Benchmarking

Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

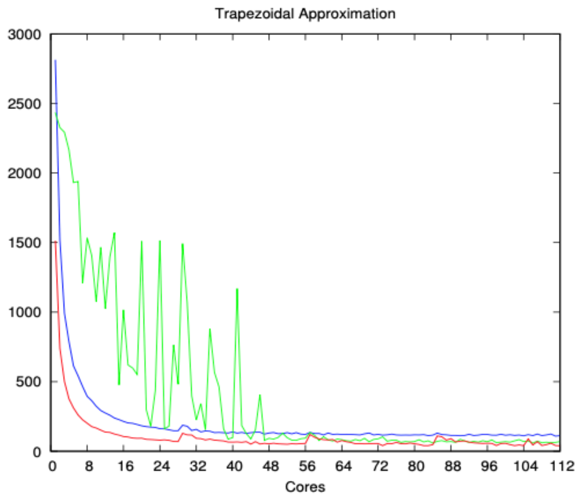
Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

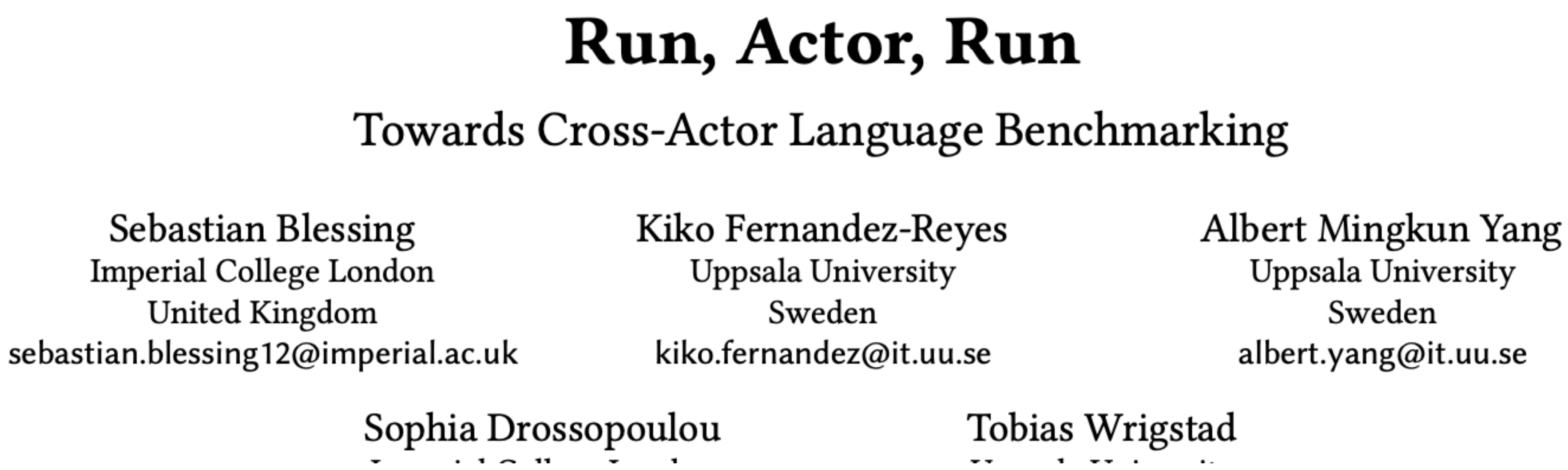
Sophia Drossopoulou

Tobias Wrigstad

Agere 2019



Pony vs **Akka** vs **CAF**



Run, Actor, Run

Towards Cross-Actor Language Benchmarking

Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

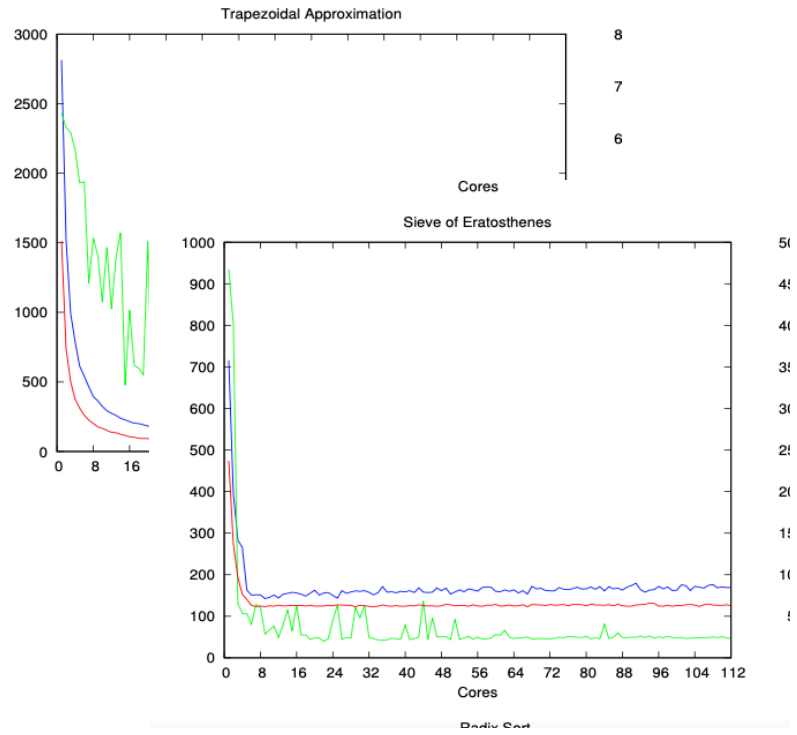
Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

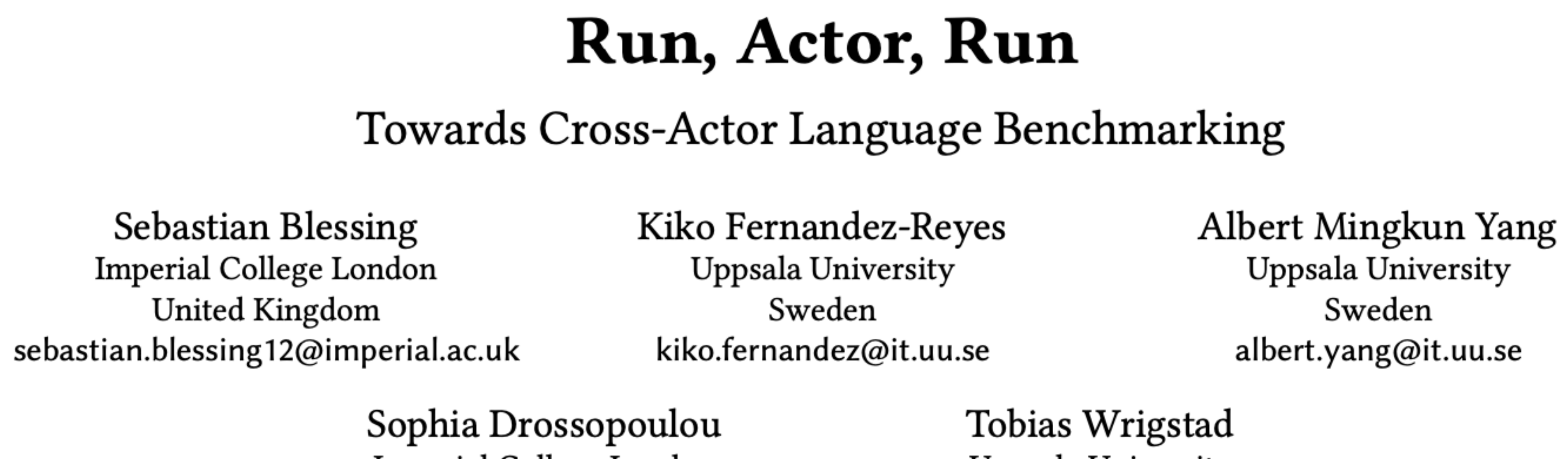
Sophia Drossopoulou

Tobias Wrigstad

Agere 2019



Pony vs Akka vs CAF



Run, Actor, Run

Towards Cross-Actor Language Benchmarking

Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

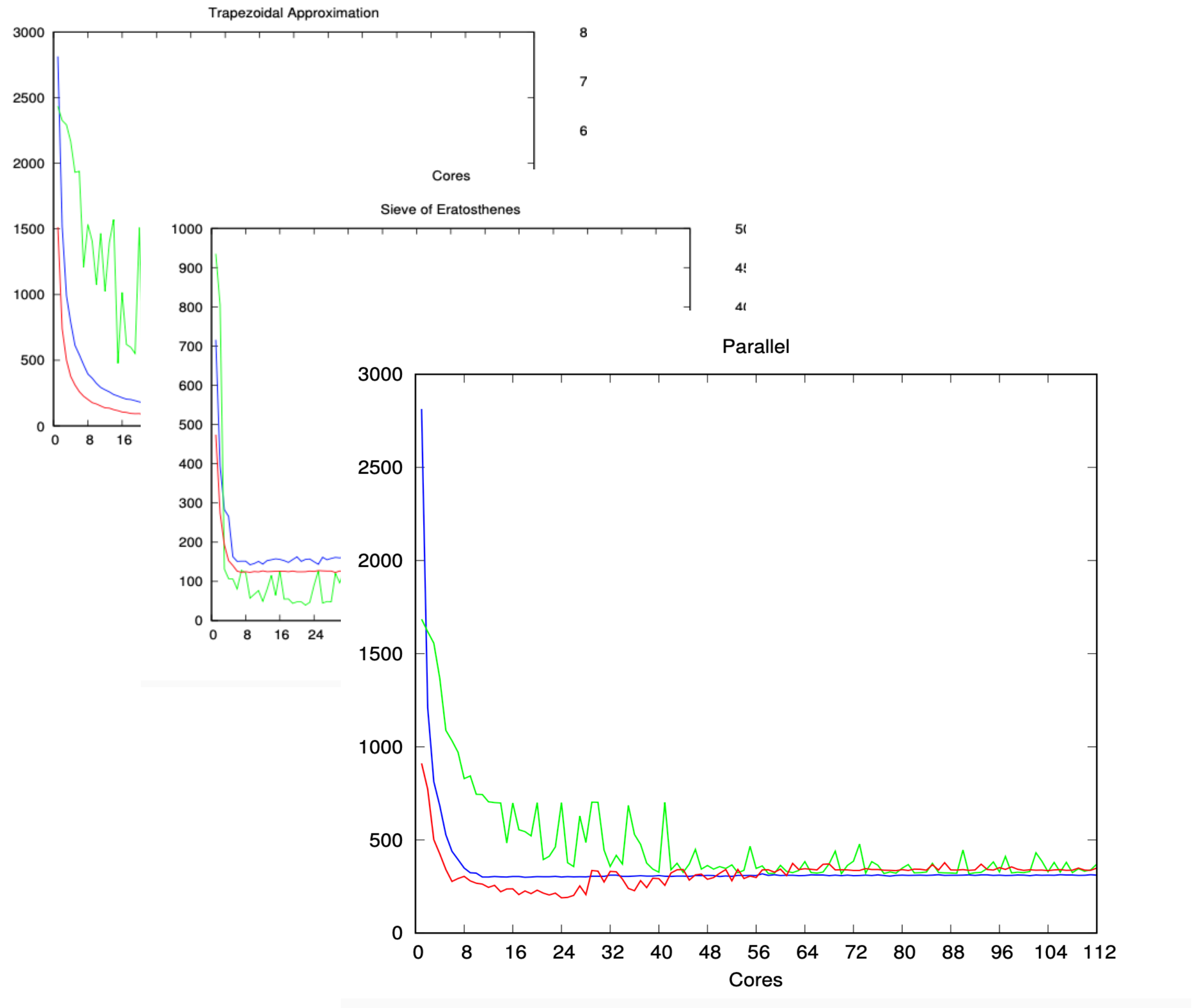
Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Sophia Drossopoulou

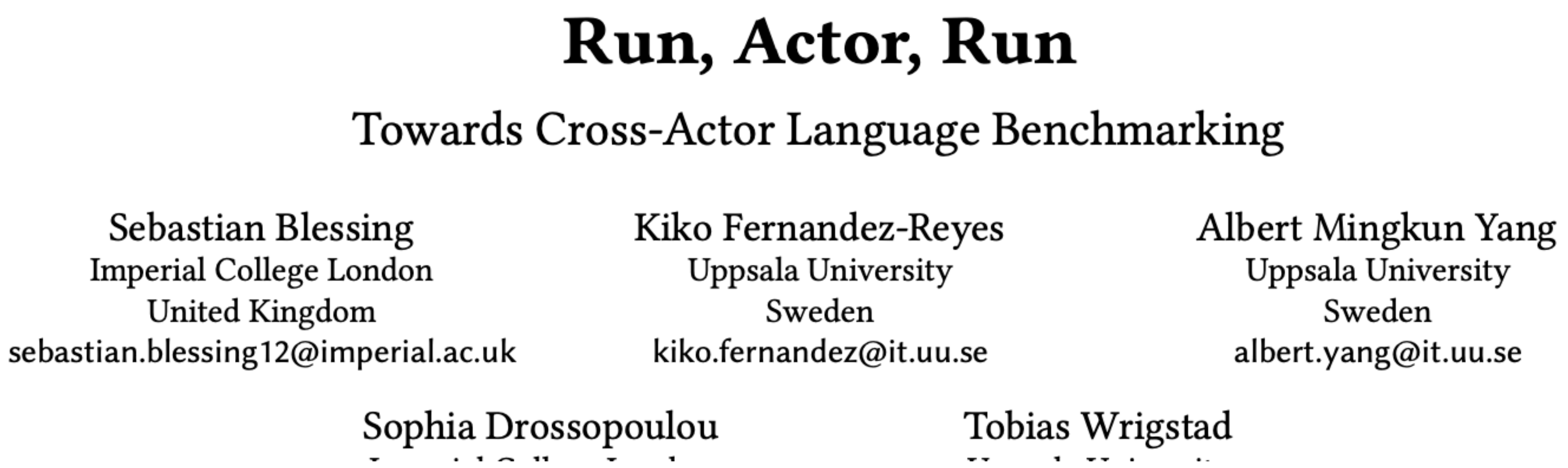
Tobias Wrigstad

Agere 2019



Pony vs Akka vs CAF

Agere 2019



Run, Actor, Run

Towards Cross-Actor Language Benchmarking

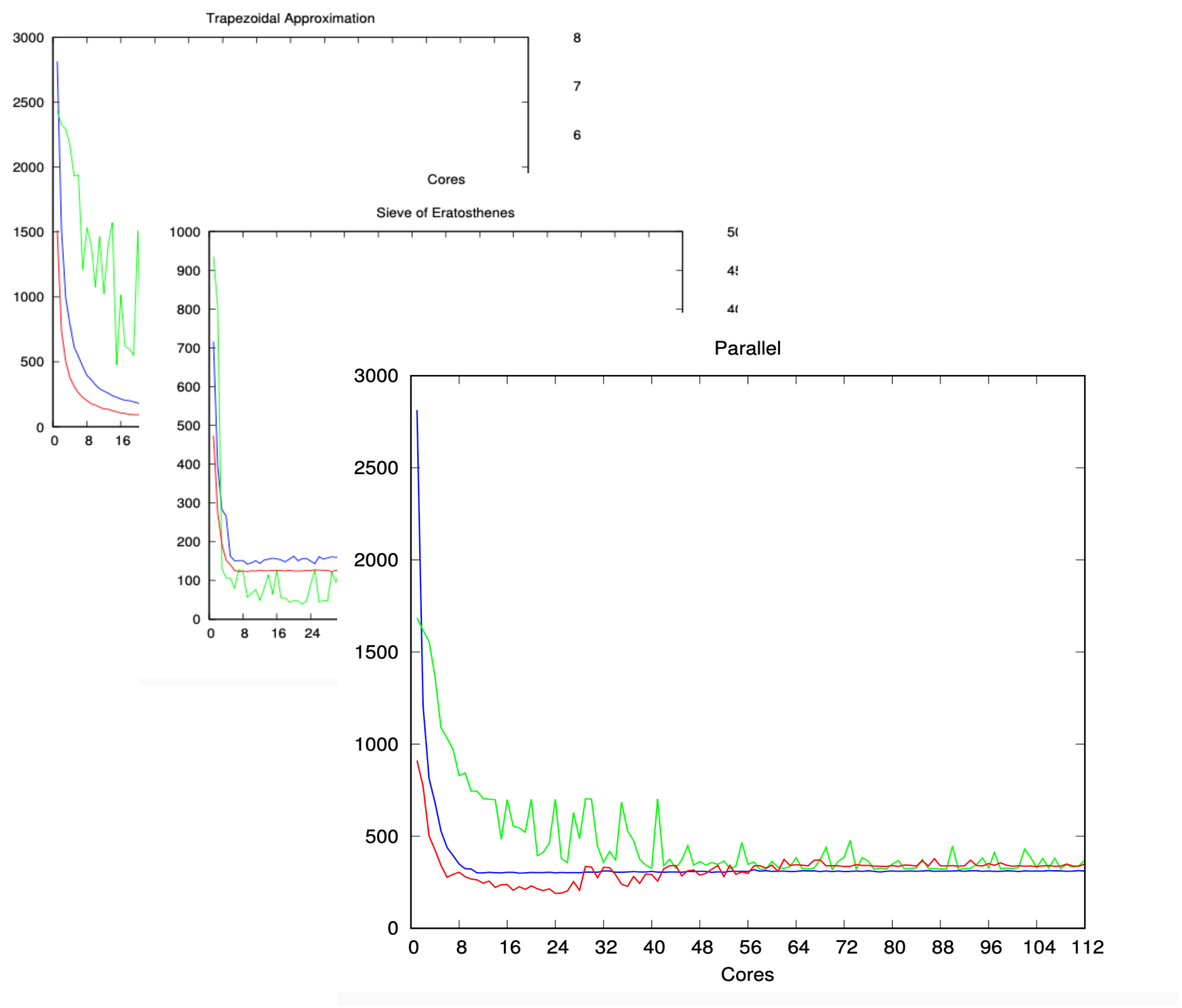
Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

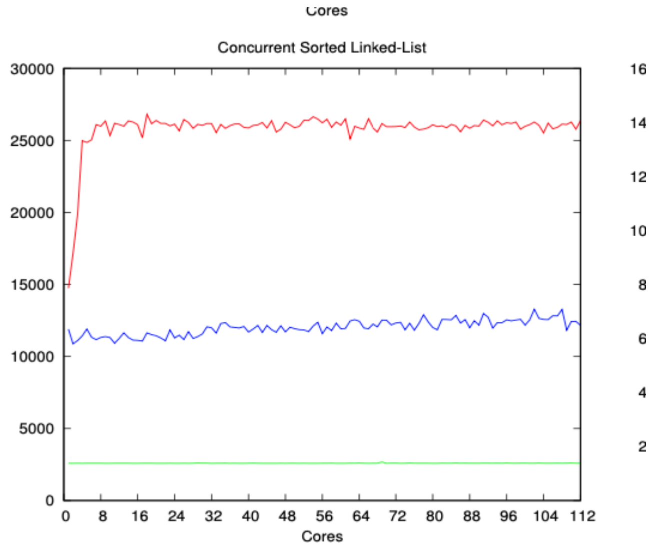
Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Sophia Drossopoulou

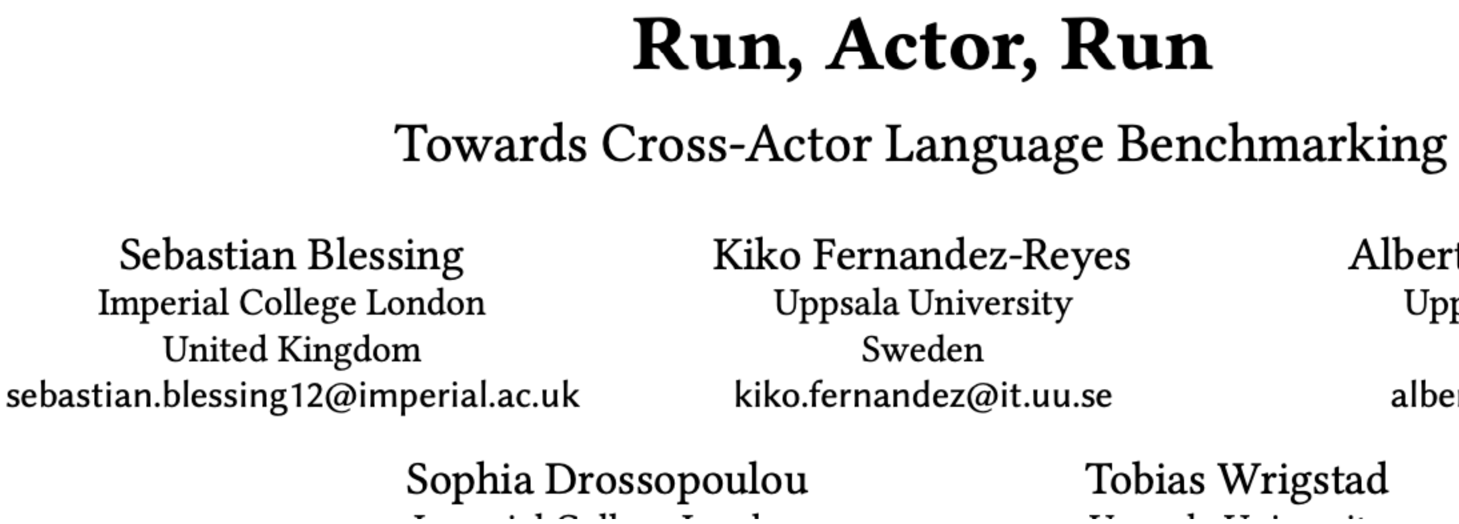
Tobias Wrigstad





Pony vs Akka vs CAF

Agere 2019



Run, Actor, Run

Towards Cross-Actor Language Benchmarking

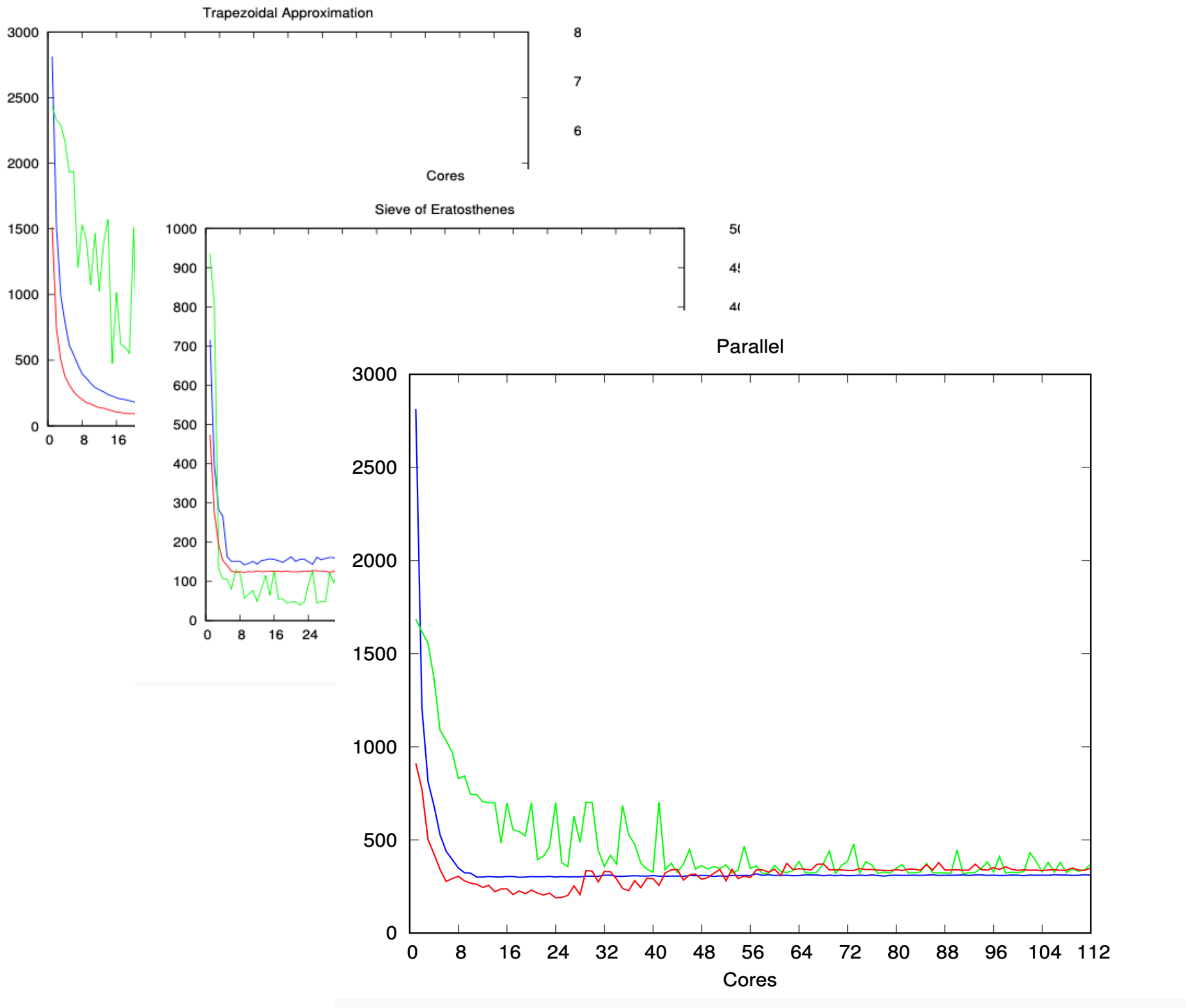
Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

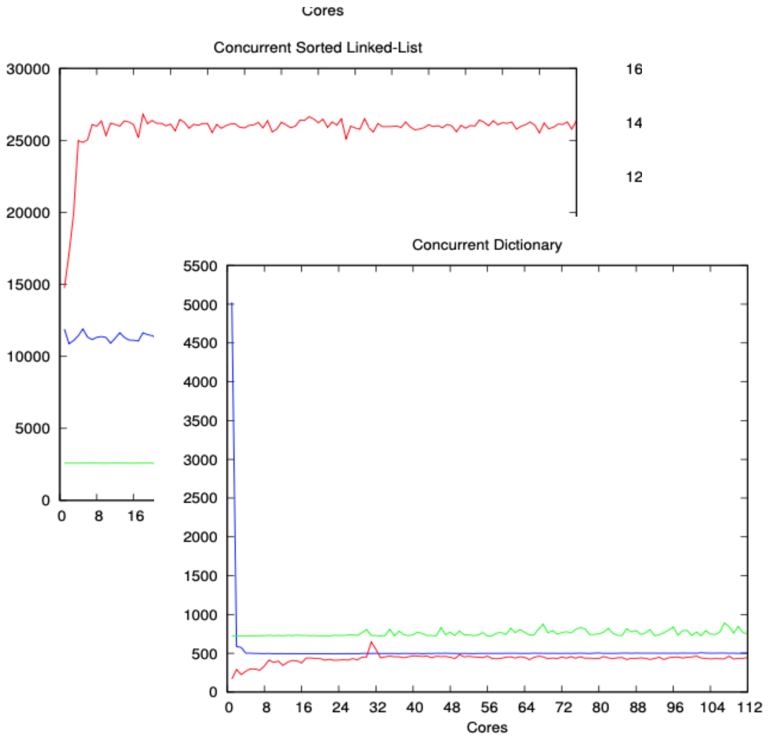
Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Sophia Drossopoulou

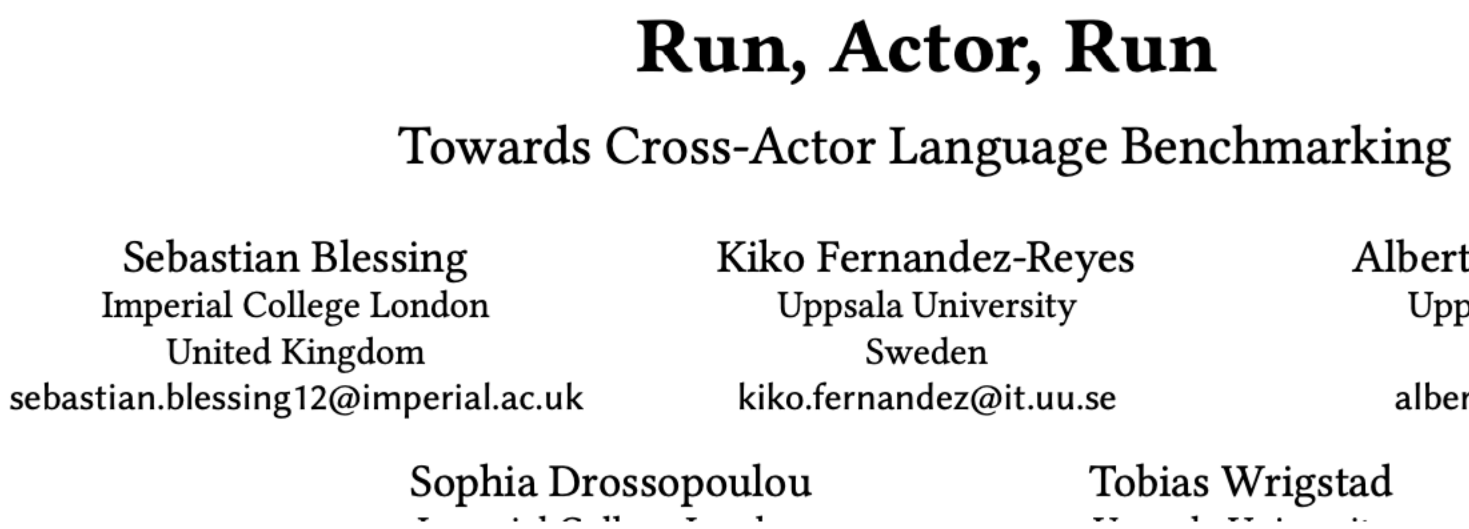
Tobias Wrigstad





Pony vs Akka vs CAF

Agere 2019



Run, Actor, Run

Towards Cross-Actor Language Benchmarking

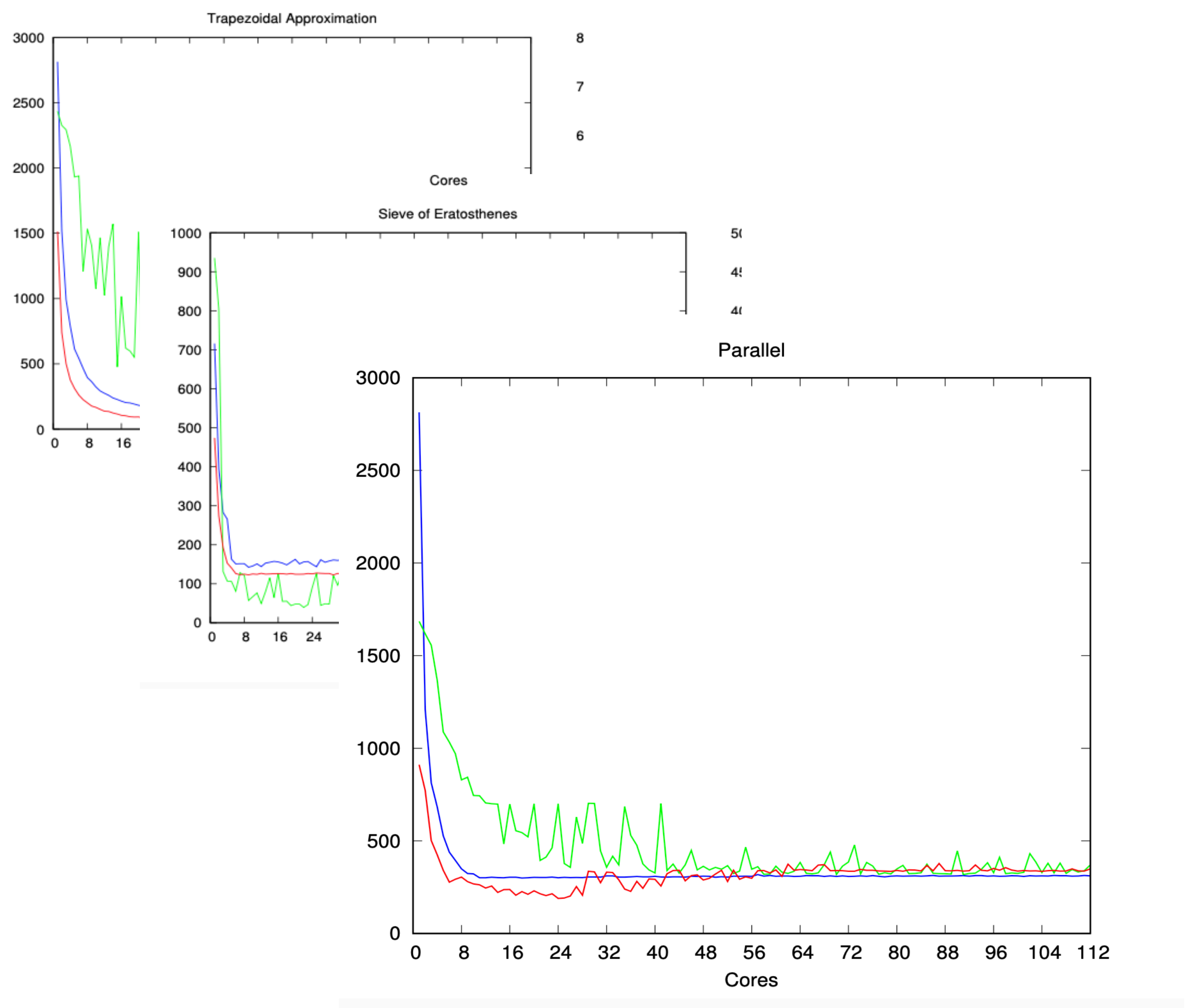
Sebastian Blessing
Imperial College London
United Kingdom
sebastian.blessing12@imperial.ac.uk

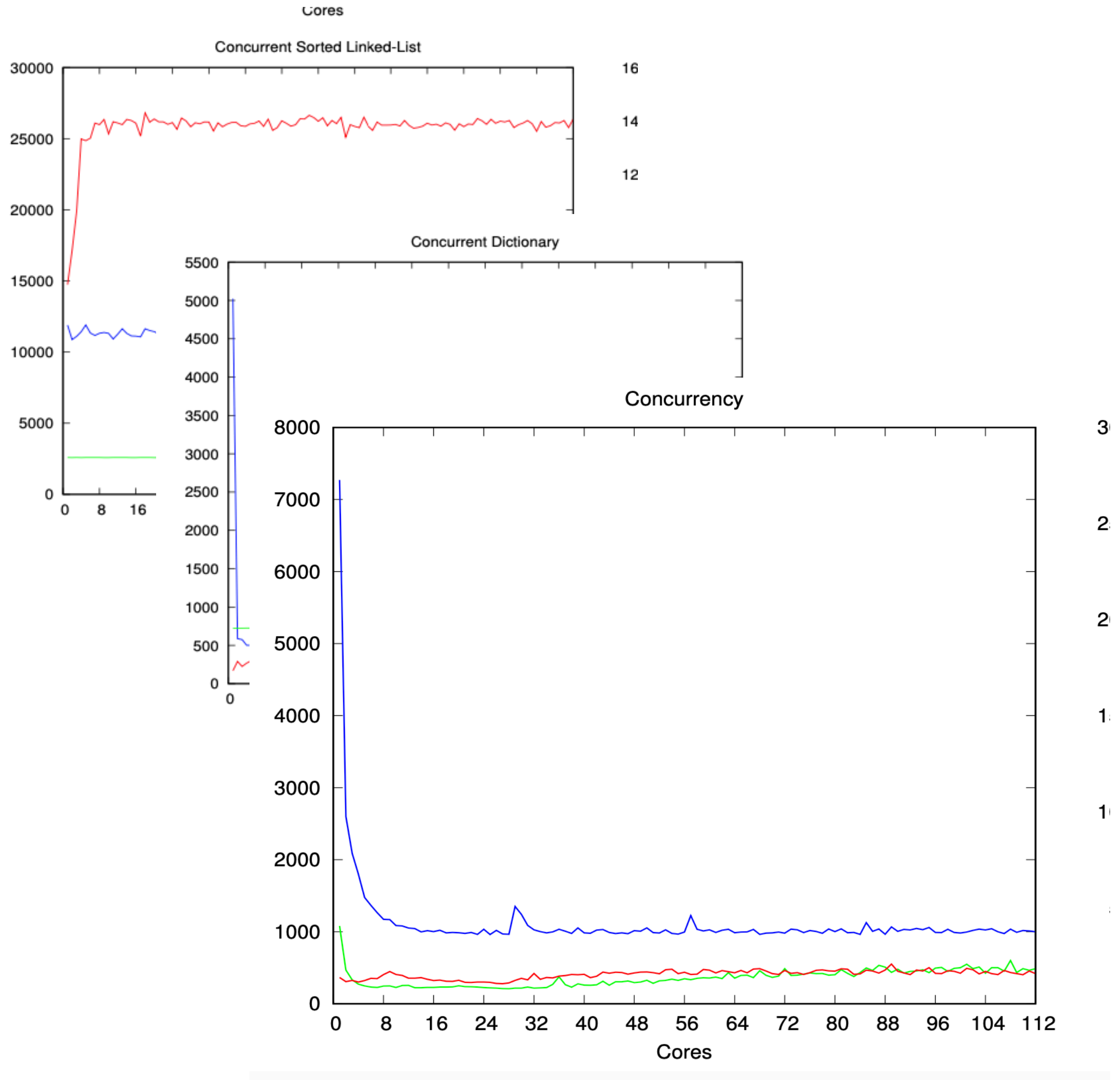
Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

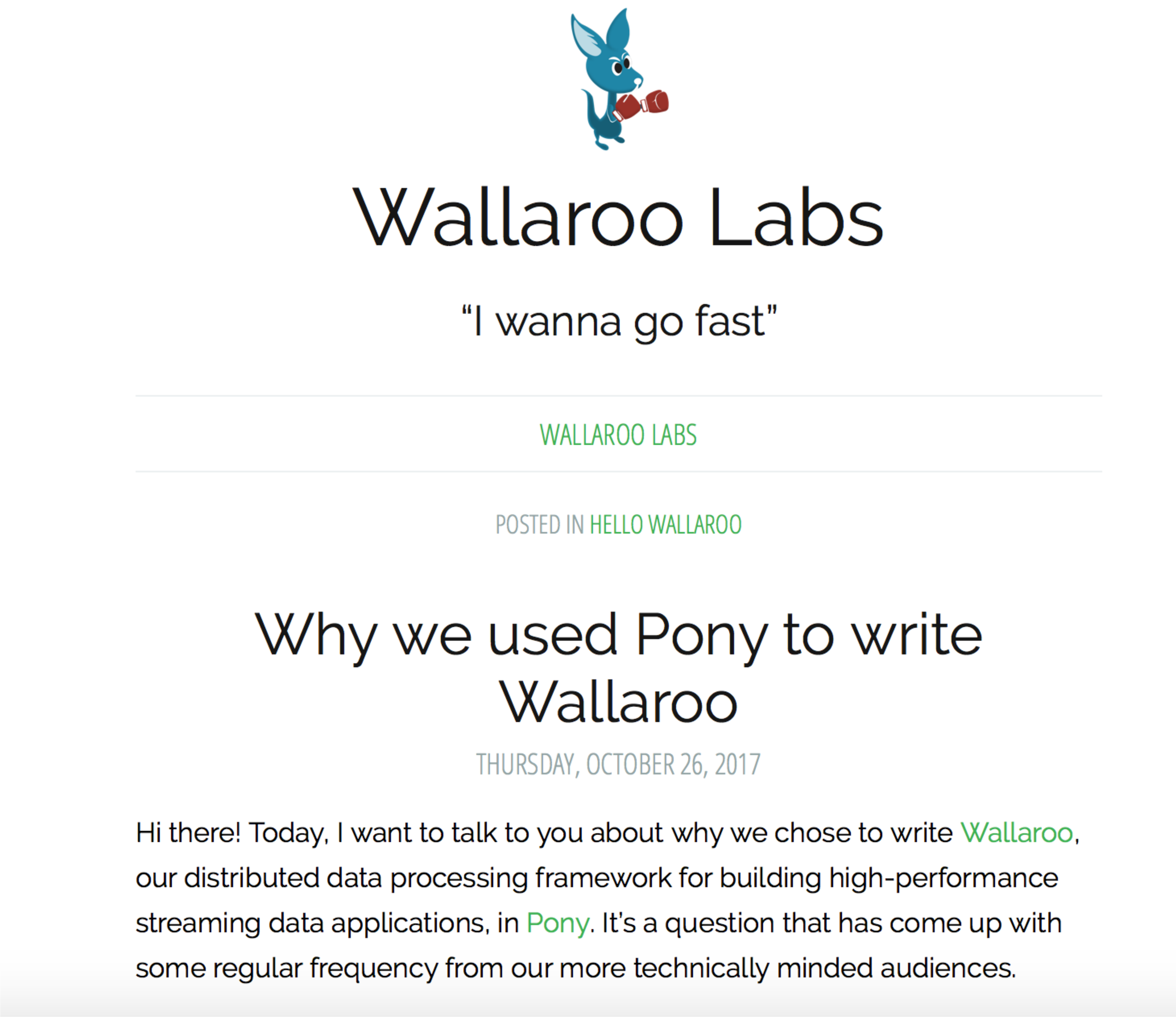
Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Sophia Drossopoulou

Tobias Wrigstad







Wallaroo Labs

“I wanna go fast”

WALLAROO LABS

POSTED IN HELLO WALLAROO

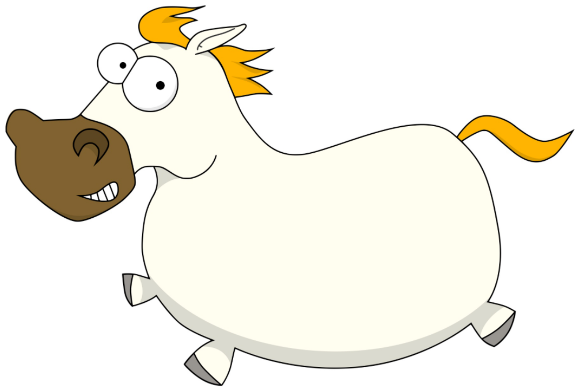
Why we used Pony to write Wallaroo

THURSDAY, OCTOBER 26, 2017

Hi there! Today, I want to talk to you about why we chose to write Wallaroo, our distributed data processing framework for building high-performance streaming data applications, in Pony. It’s a question that has come up with some regular frequency from our more technically minded audiences.

Pony features

- actors, objects
- pass mutable state without copying
- static types, type safe
- no Null values
- capabilities
- checked exceptions
- pattern marching
- lambda-s and partial applications
- causality
- traits and interfaces (nominal and structural types)
- union and intersection types
- generics ala f-bounded polymorphism
- consuming and destructive read
- alias/unalias and viewpoints in types
- C ffi
- small library



Today’s Talk

- Pony - the language and its design
    - Actors
    - Causality
    - The Type System
    - Garbage Collection

Today’s Talk

- Pony - the language and its design
    - Actors
    - Causality
    - The Type System
    - Garbage Collection

Today’s Talk

- Pony - the language and its design
    - **Actors**
    - Causality
    - The Type System
    - Garbage Collection

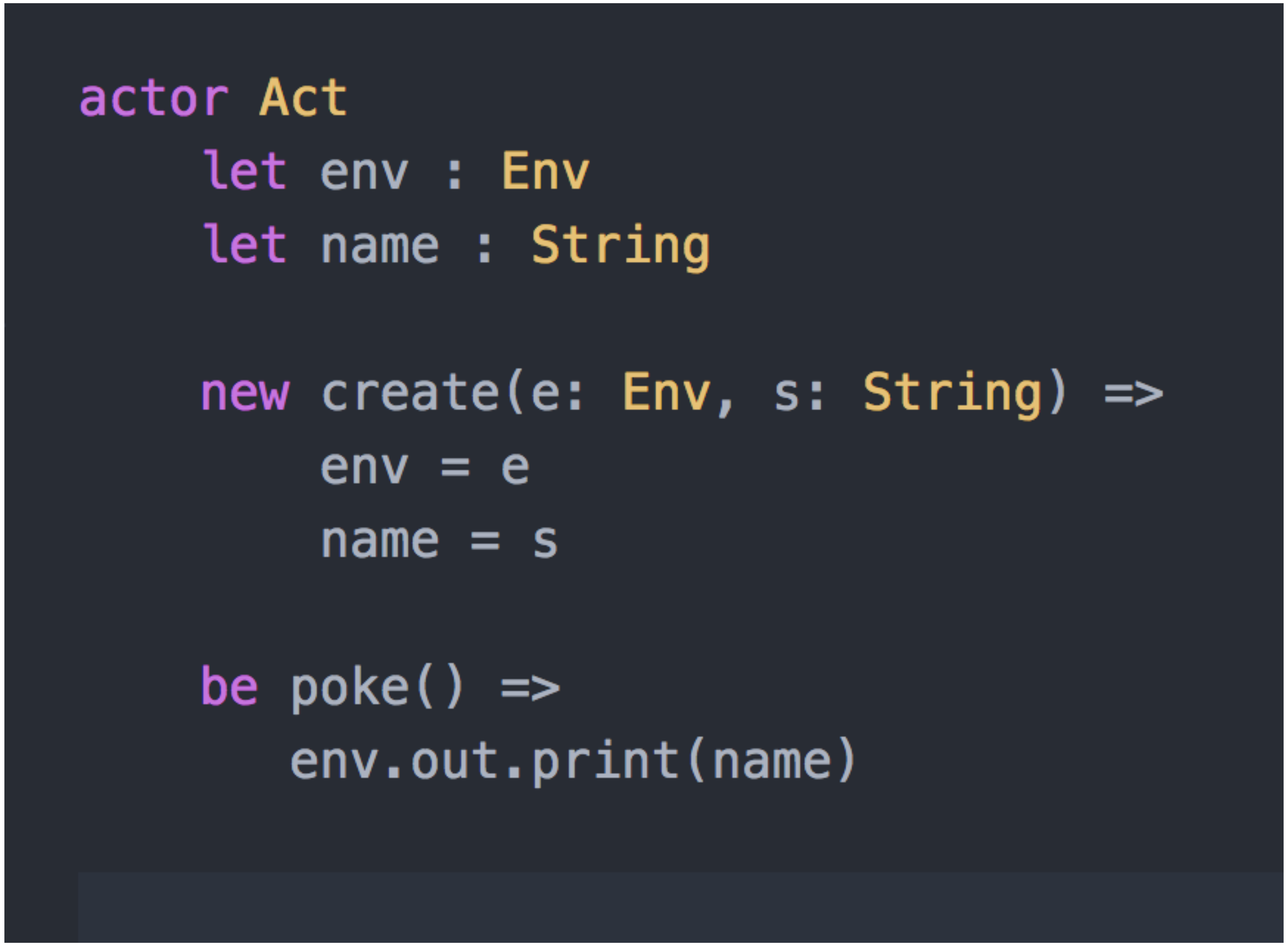
The actor paradigm in Pony

- 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)

Actors in Pony

Code in `1_Actors/ABC.pony`

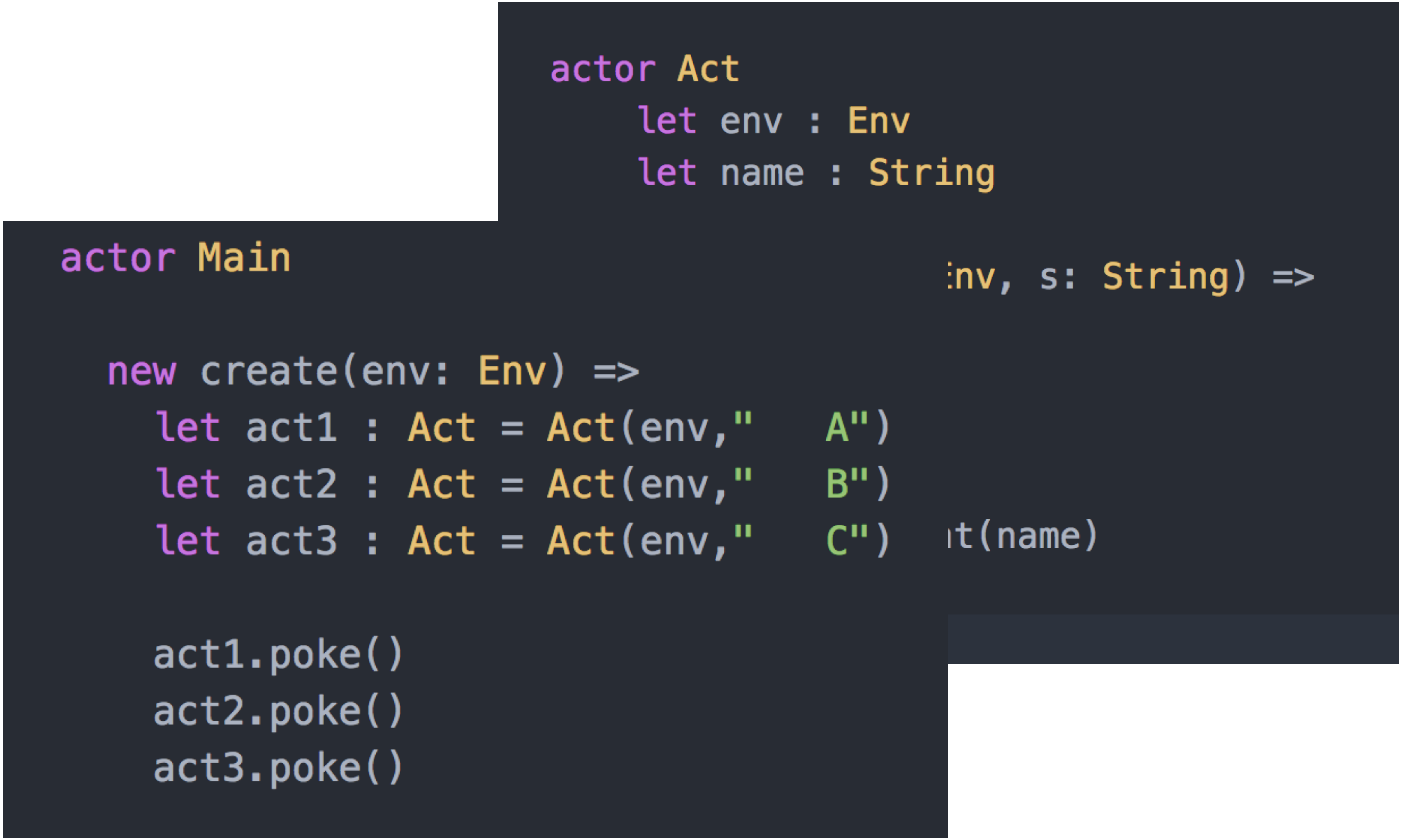
Actors in Pony



```
actor Act
    let env : Env
    let name : String

    new create(e: Env, s: String) =>
        env = e
        name = s

    be poke() =>
       env.out.print(name)
```

Code in `1_Actors/ABC.pony`

Actors in Pony

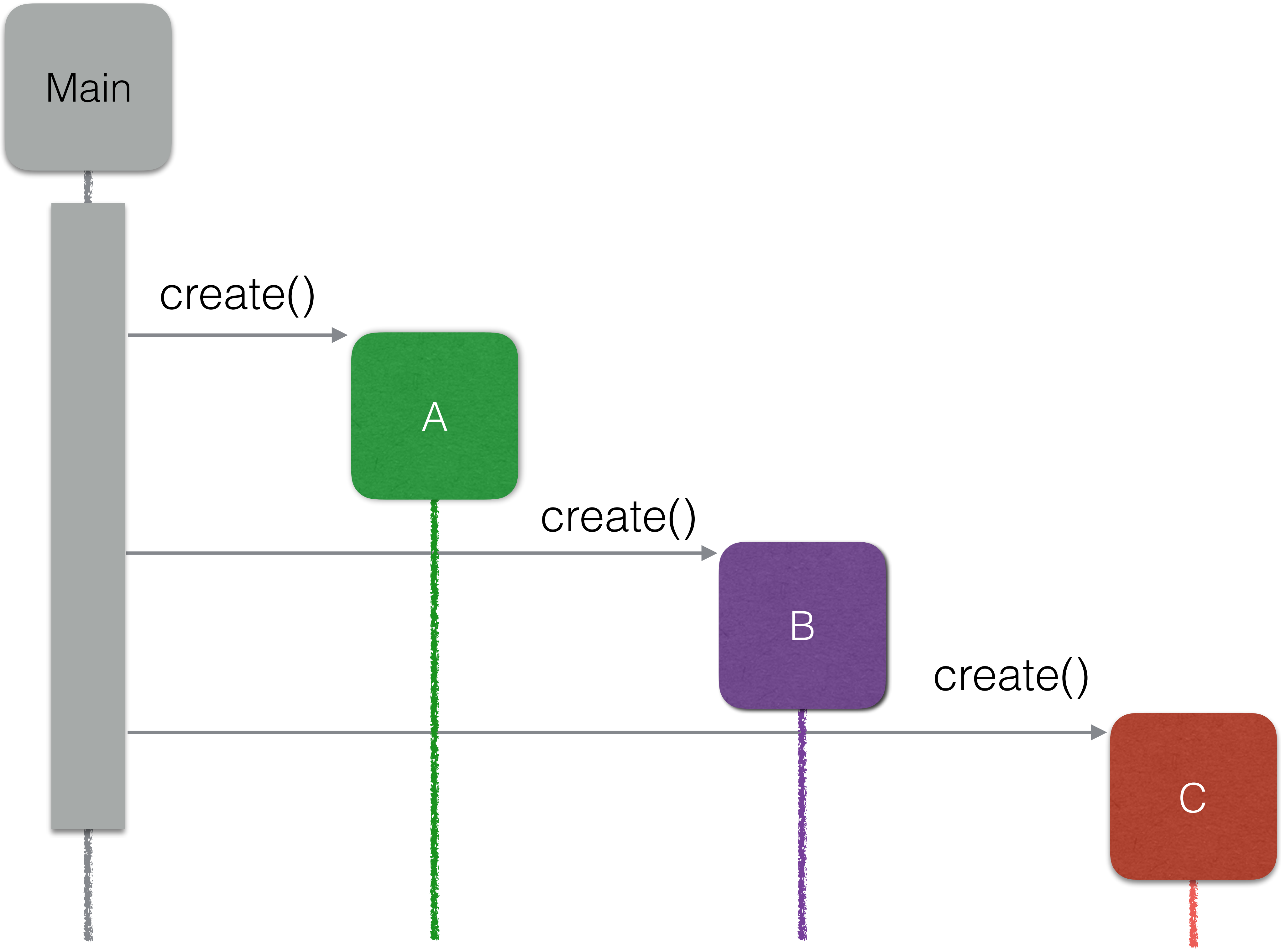


```
actor Act
    let env : Env
    let name : String

                  nv, s: String) =>




                  t(name)
```

```
actor Main

  new create(env: Env) =>
    let act1 : Act = Act(env,"   A")
    let act2 : Act = Act(env,"   B")
    let act3 : Act = Act(env,"   C")

    act1.poke()
    act2.poke()
    act3.poke()
```

Code in `1_Actors/ABC.pony`

Main.create(...)

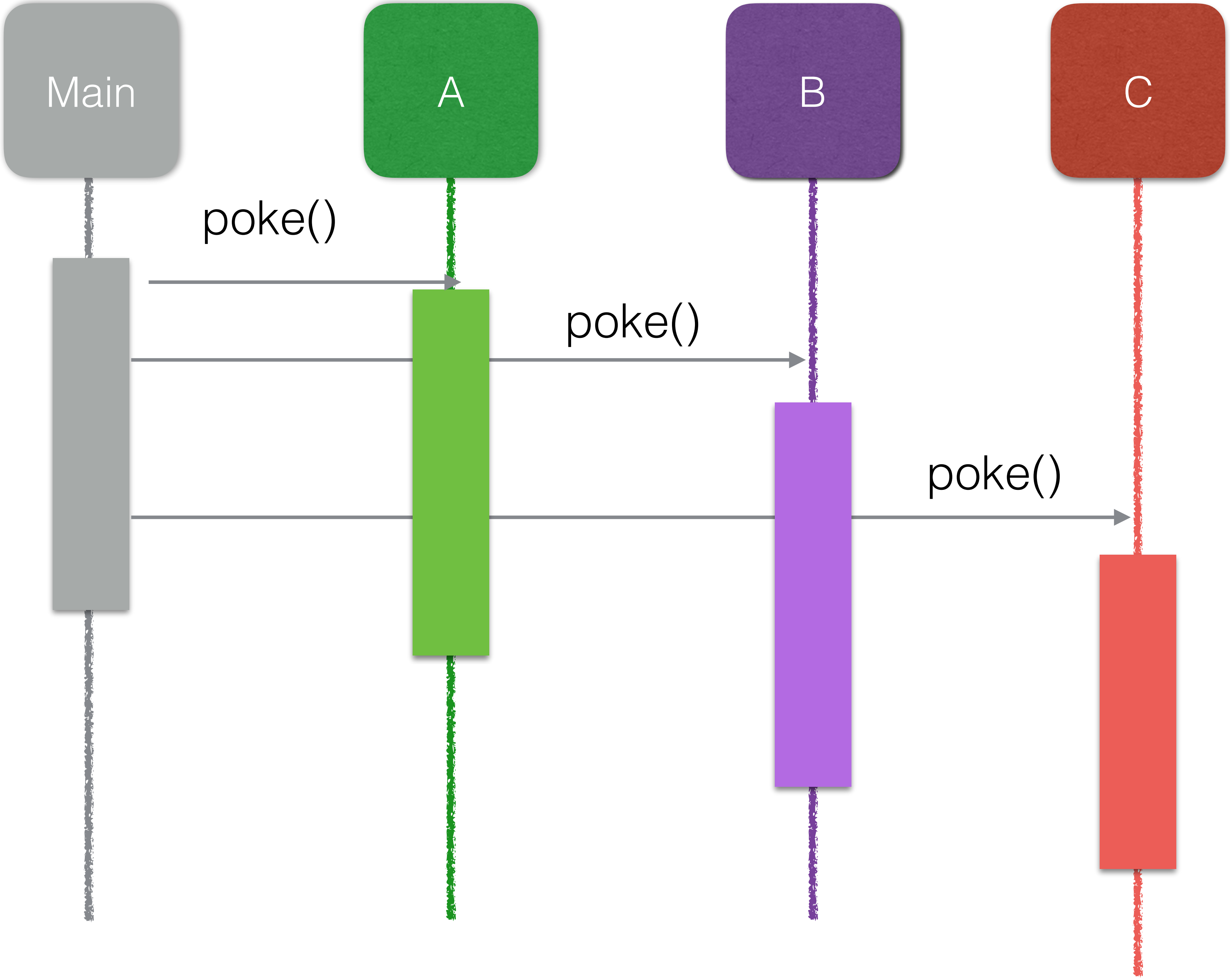


Main.run()

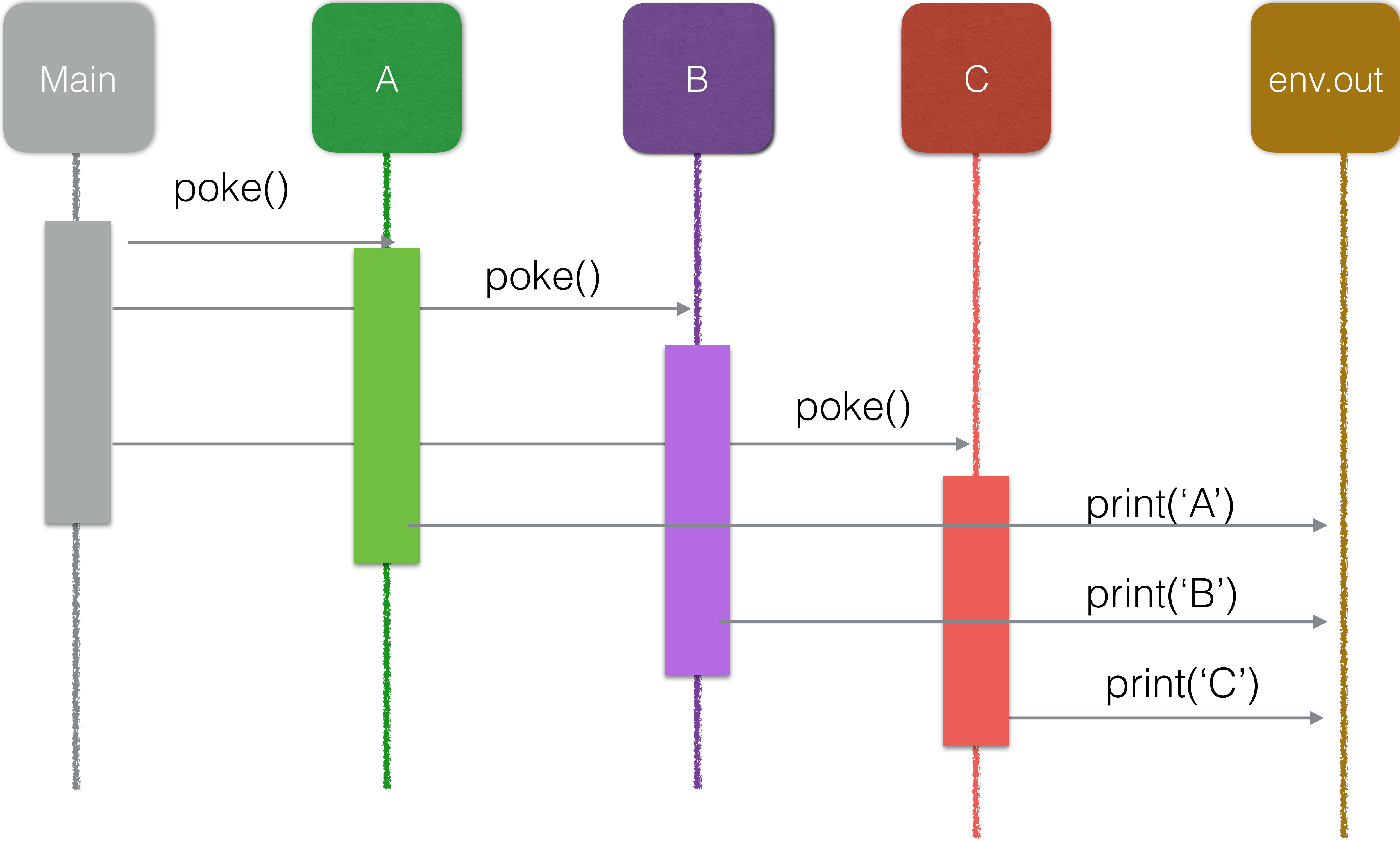
```
actor Main
  let env  : Env
  let actA : Act
  …

  new create(e: Env) =>
    …
    run()

  be run() =>
        actA.poke()
        actB.poke()
        actC.poke ()
```

```
actor Act
  let env  : Env
  let name : string

  new create(e:Env,s:String)=>
        …

  be poke() =>
      env.out.print(name)
```

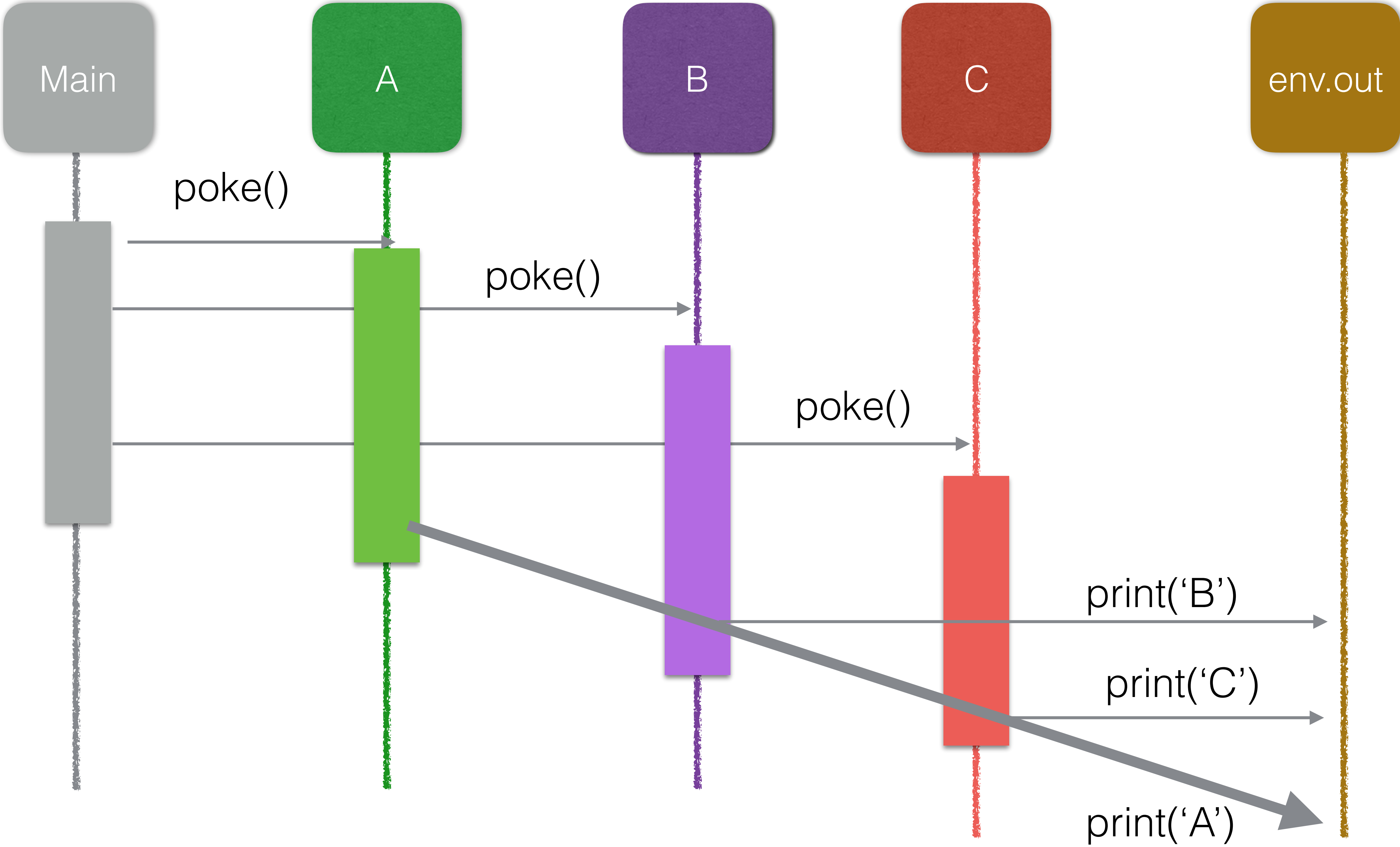
Main.run()



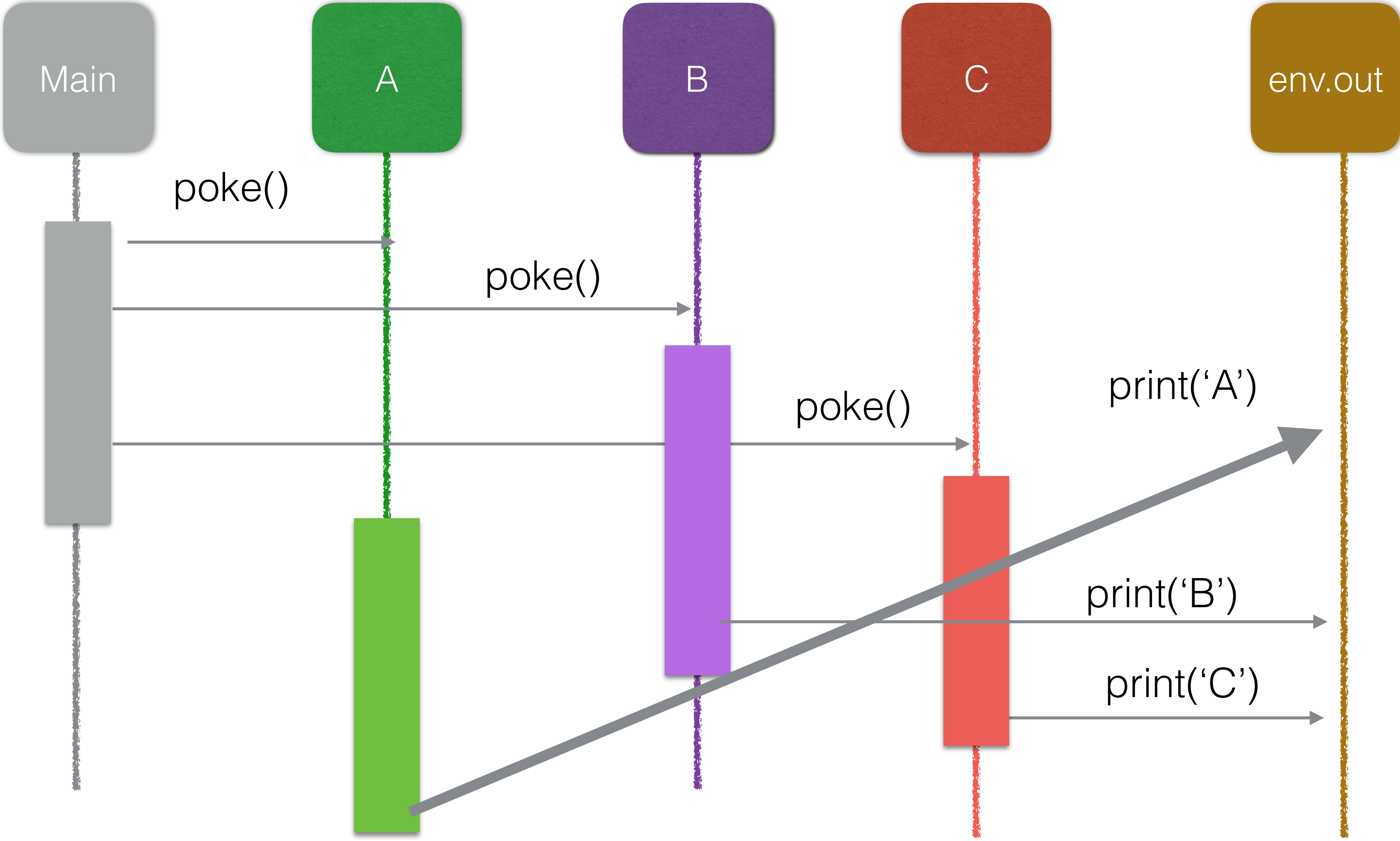
Main.run()



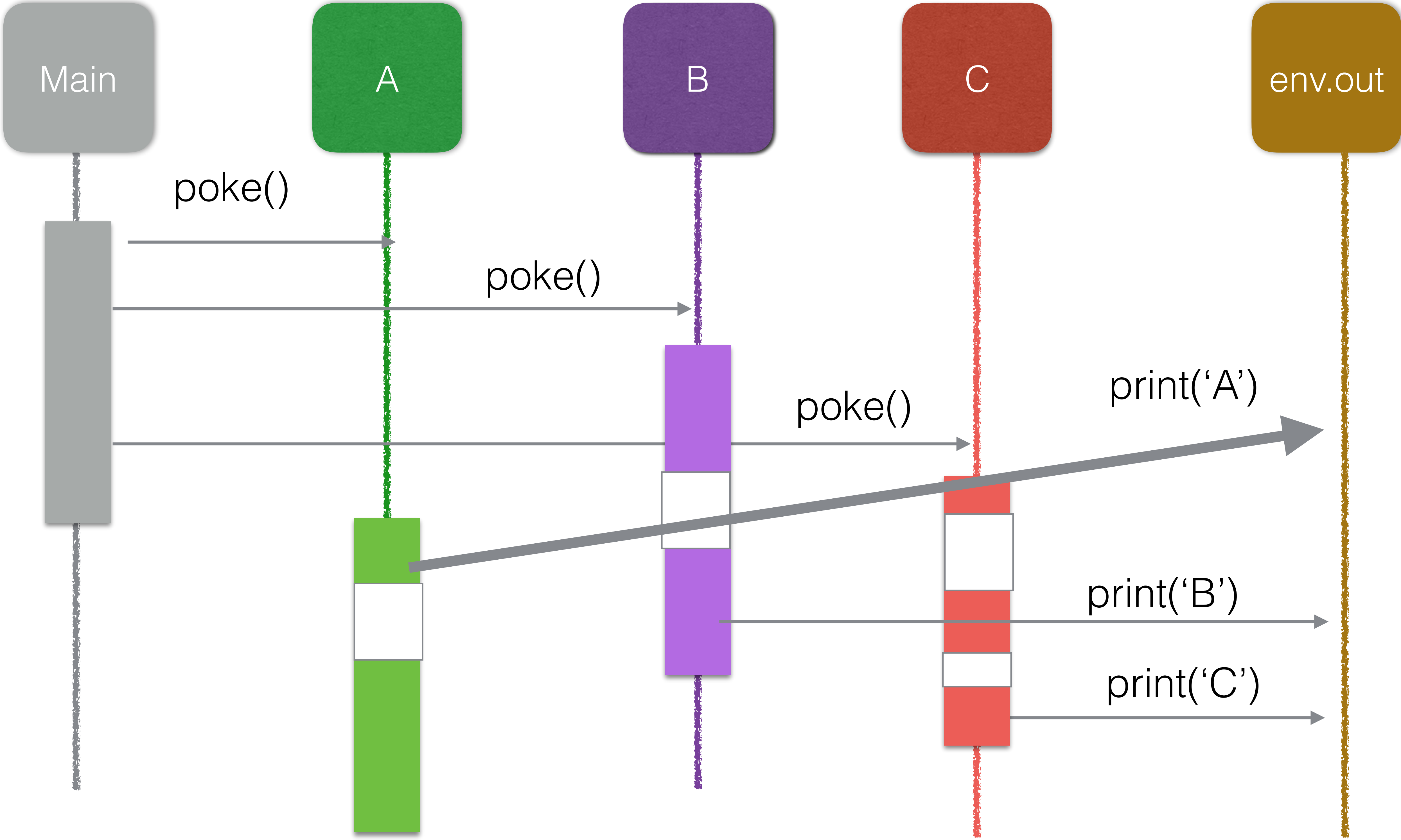
Main.run() -also poss



Main.run() -also poss



Main.run() -also poss

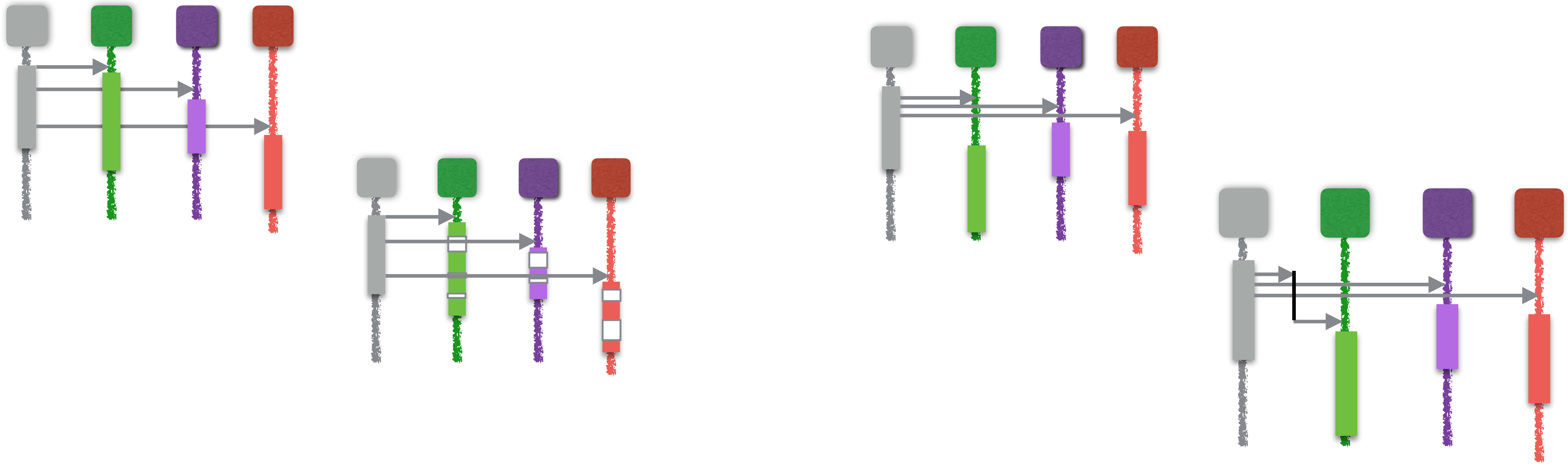


Today's Talk

- Pony - the language and its design
  - Actors
  - **Causality**
  - The Type System
  - Garbage Collection

Uncertainty?

Uncertainty?

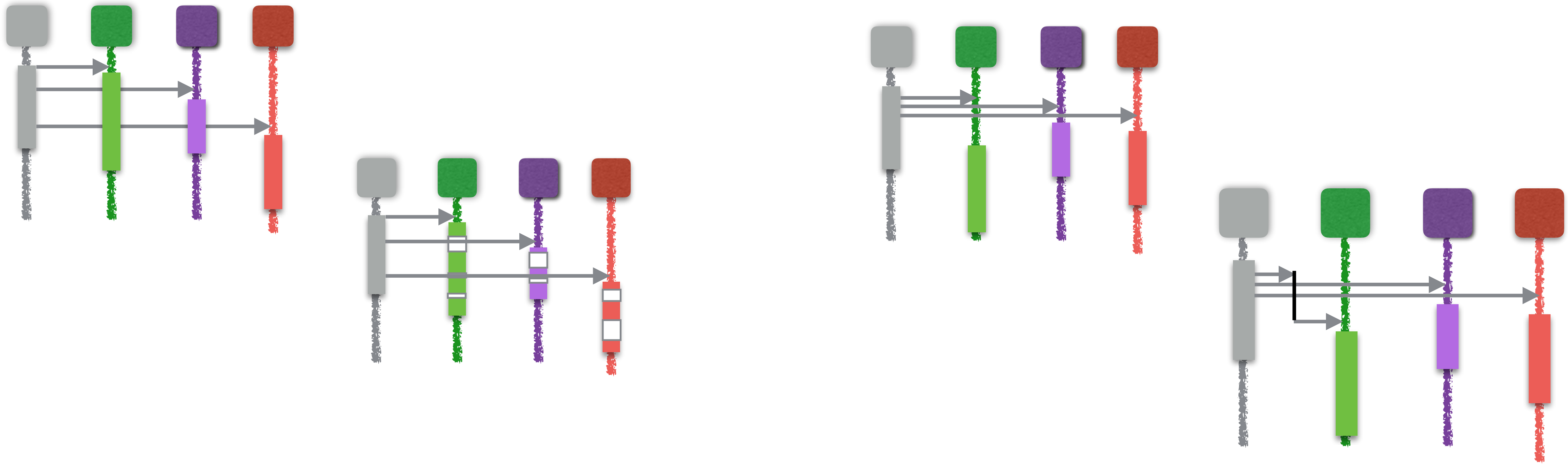


- What do the other actors do while I am executing?
- When will the message be taken off the queue?
- When will the message be delivered to the queue?

Uncertainty alleviated through Types and through Causal Message Delivery

Uncertainty alleviated through Types and through Causal Message Delivery

Uncertainty alleviated through **types**



- What do the other actors do while I am executing?
- When will the message be taken off the queue?
- When will the message be delivered to the queue?

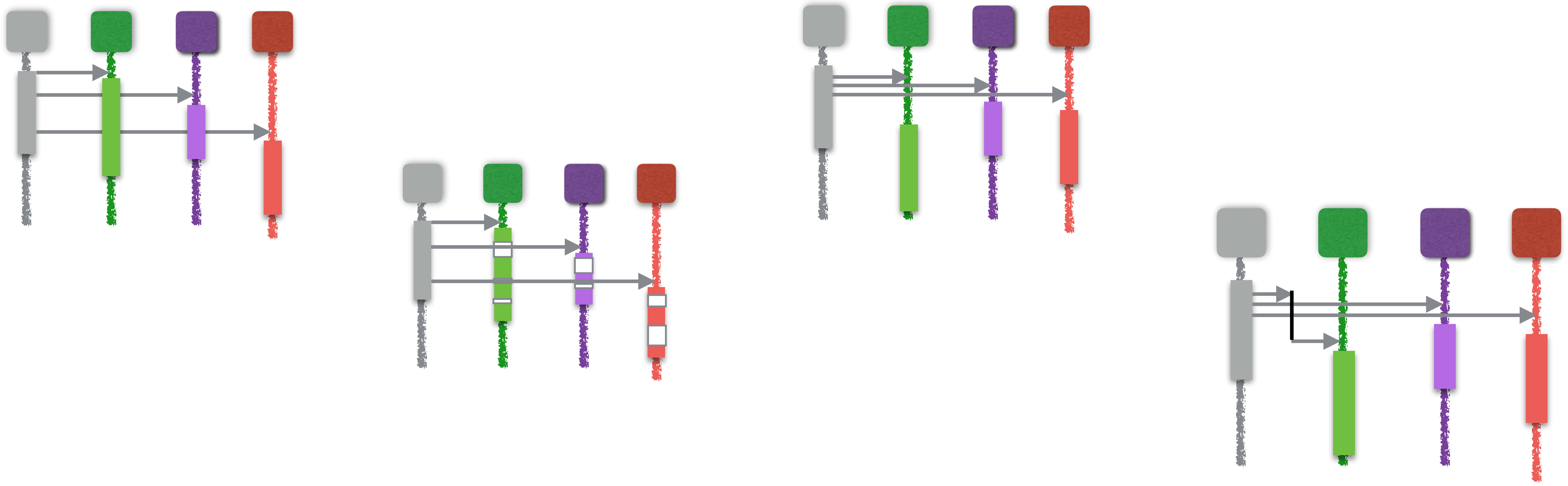
- When message taken off the queue, any changes to rest of world invisible to receiver.
- In fact, this guarantee holds upon message send.

Uncertainty alleviated

through Types

and through Causal Message Delivery

Uncertainty

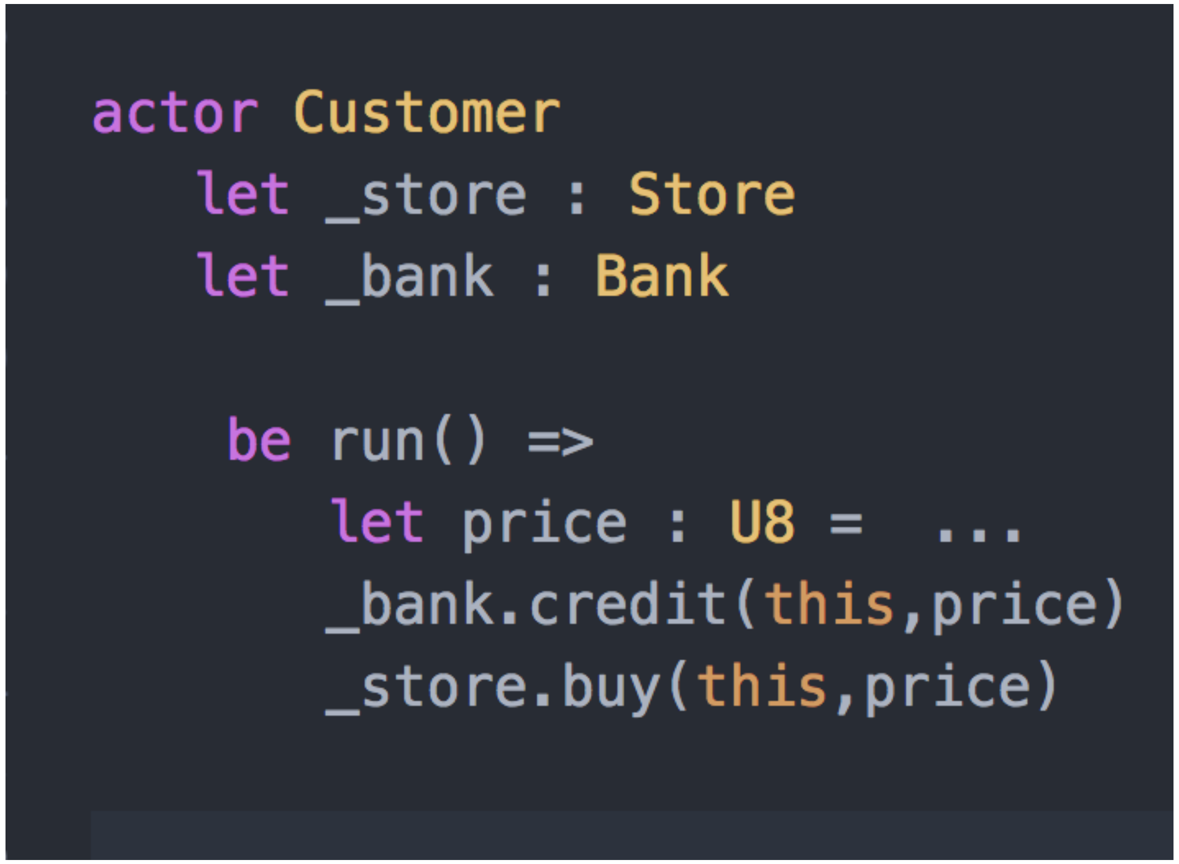


- What do the other actors do while I am executing?
- When will the message be taken off the queue?
- When will the message be delivered to the queue?

Message Delivery

Code in `2_CausalDelivery/MssgDelivery.pony`

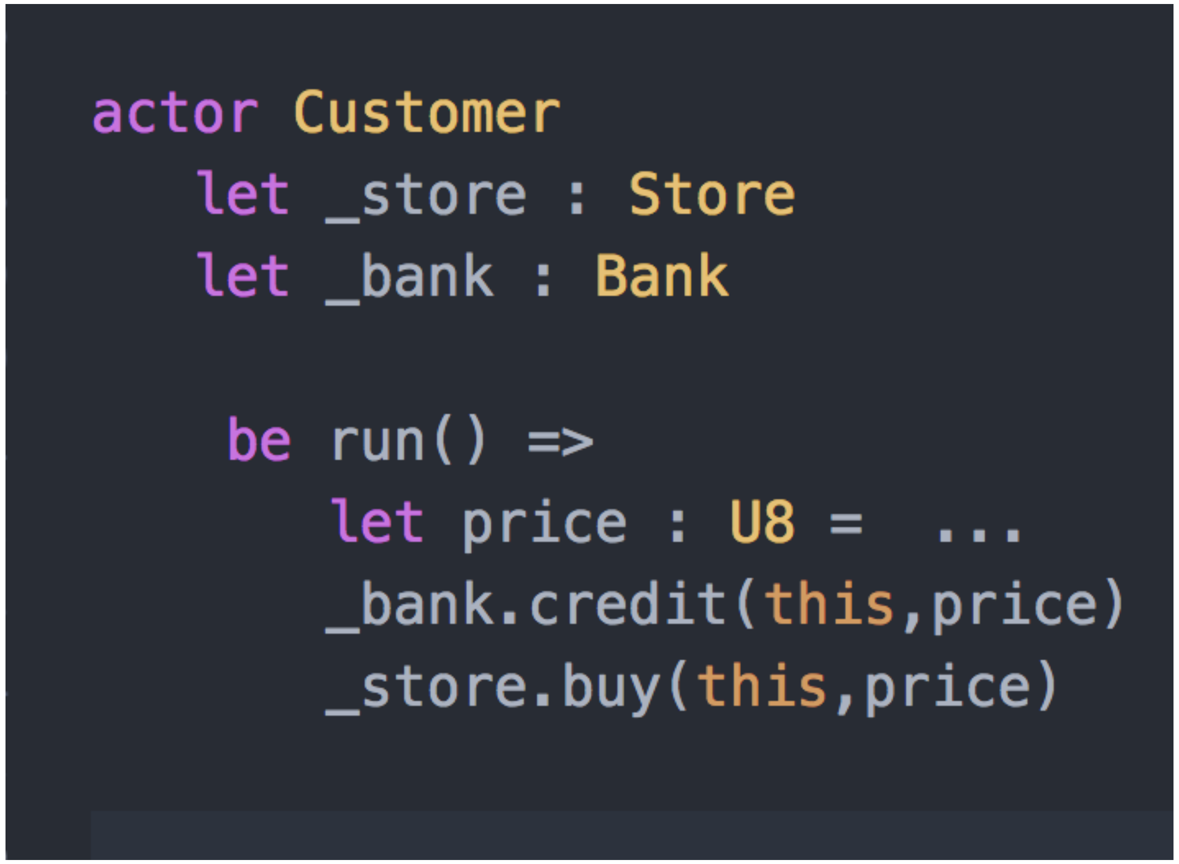
Message Delivery



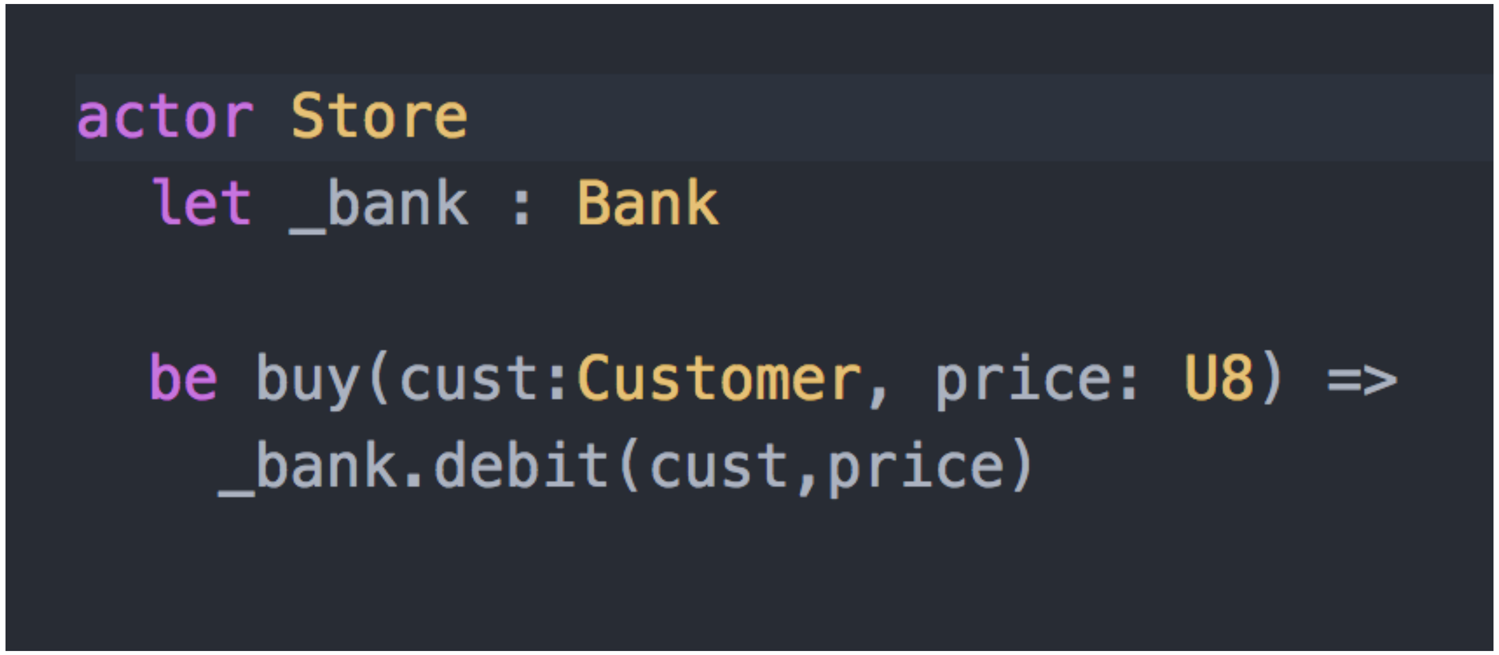
```
actor Customer
   let _store : Store
   let _bank : Bank

    be run() =>
       let price : U8 =  ...
       _bank.credit(this,price)
       _store.buy(this,price)
```

Code in `2_CausalDelivery/MssgDelivery.pony`

Message Delivery



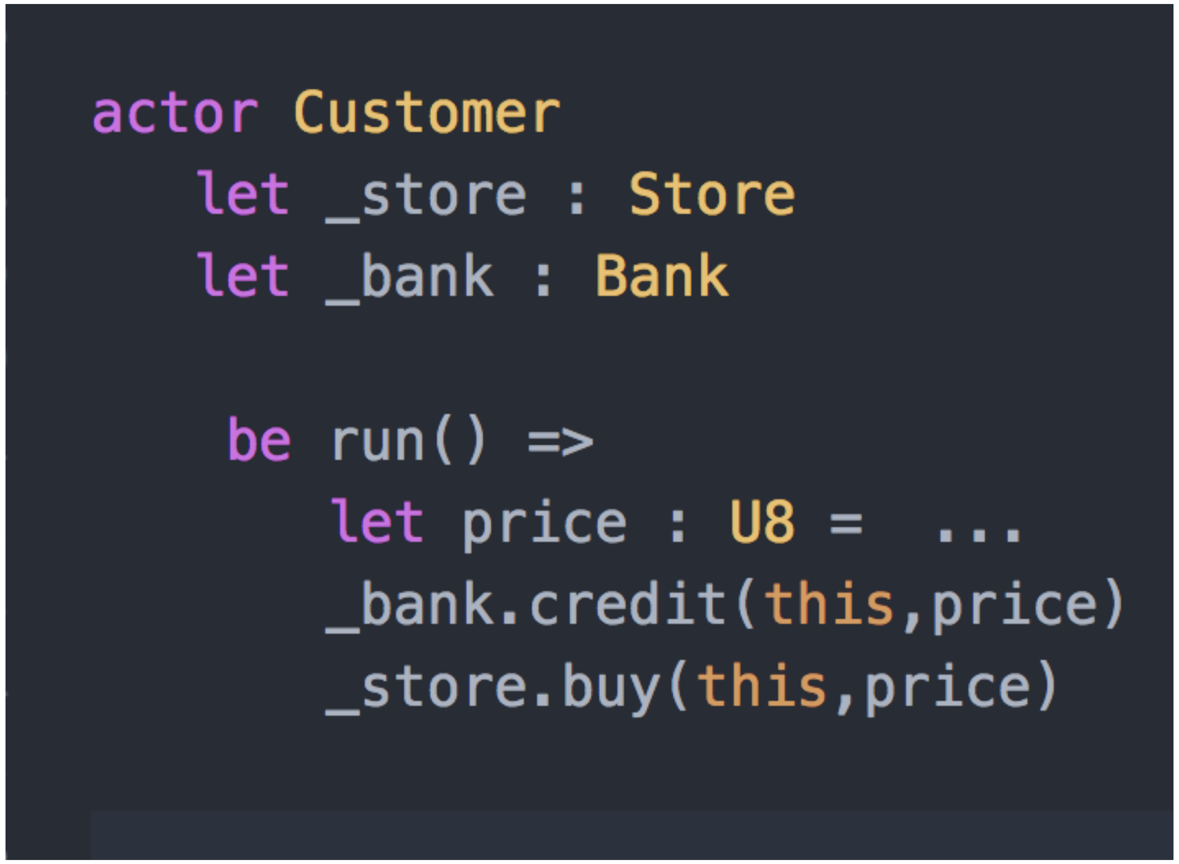
```
actor Customer
   let _store : Store
   let _bank : Bank

    be run() =>
       let price : U8 =  ...
       _bank.credit(this,price)
       _store.buy(this,price)
```



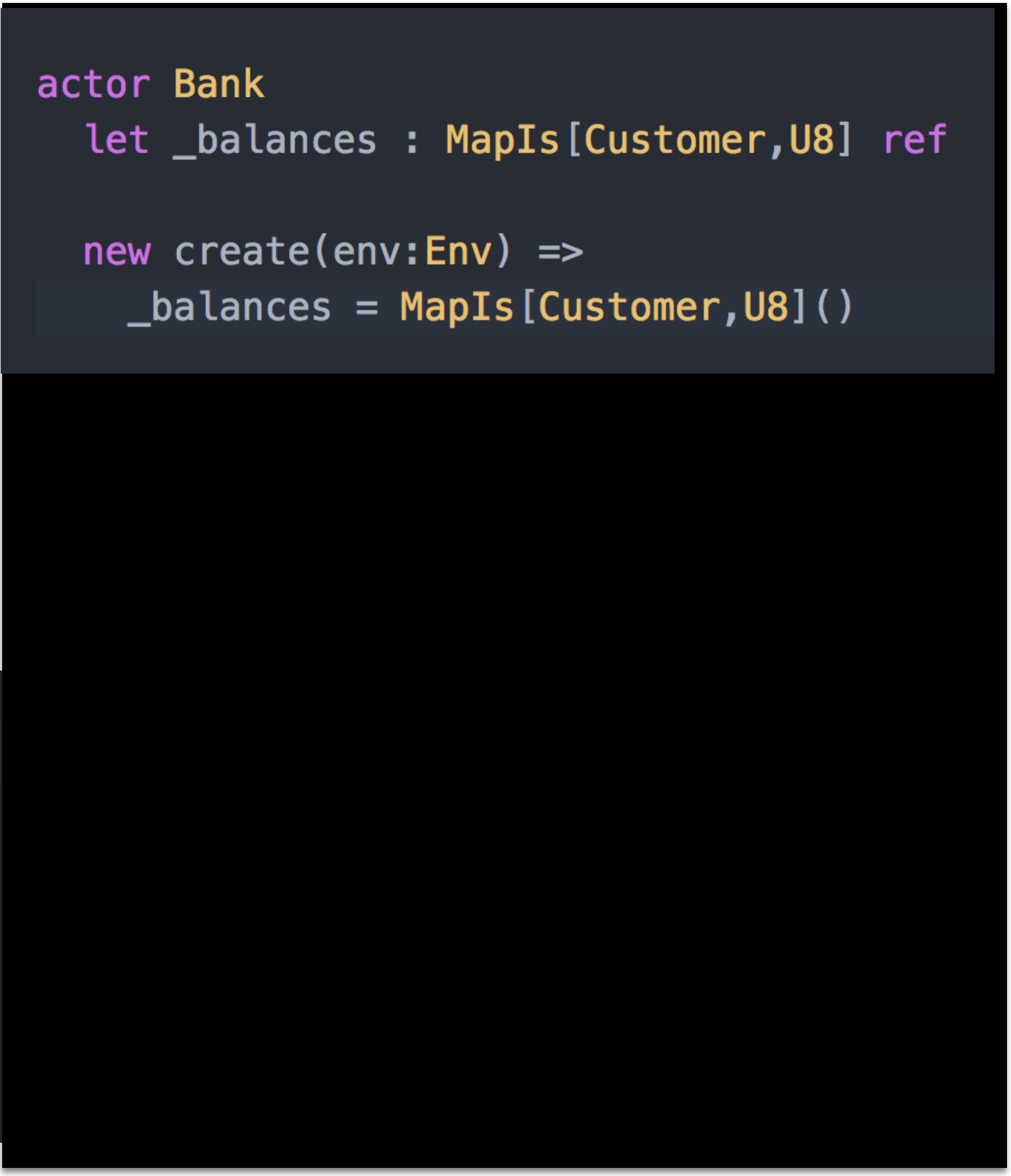
```
actor Store
  let _bank : Bank

  be buy(cust:Customer, price: U8) =>
    _bank.debit(cust,price)
```

Code in `2_CausalDelivery/MssgDelivery.pony`

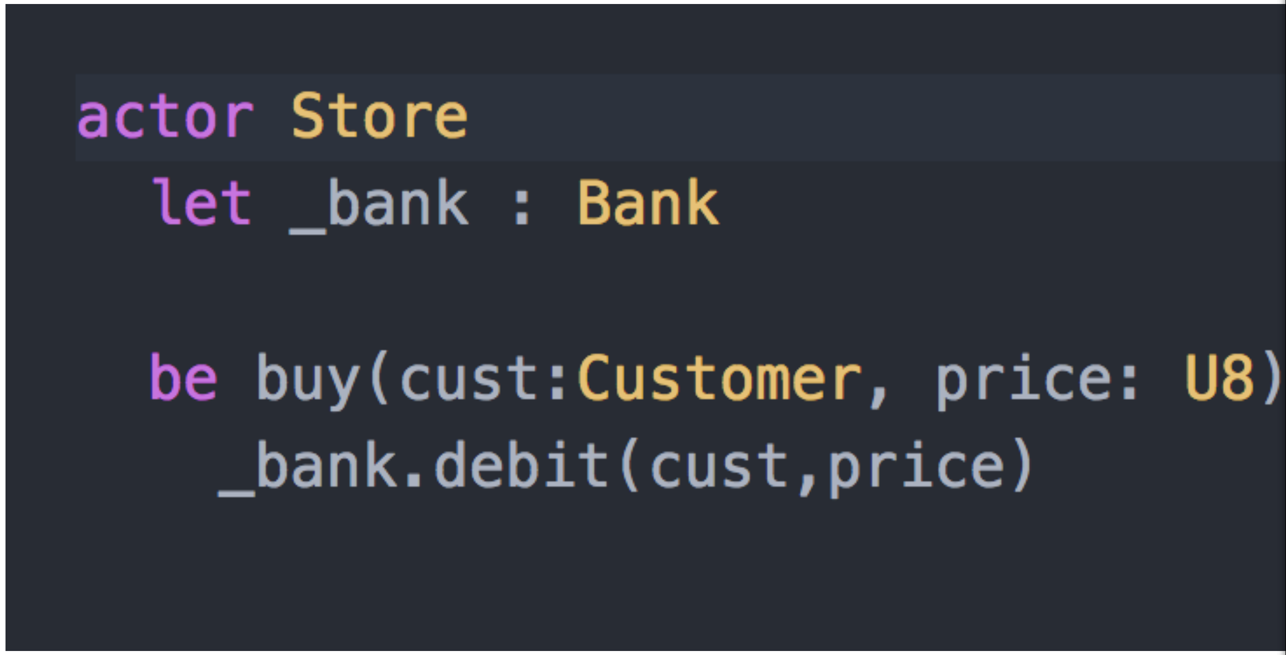
Message Delivery



```
actor Customer
   let _store : Store
   let _bank : Bank

    be run() =>
       let price : U8 =  ...
       _bank.credit(this,price)
       _store.buy(this,price)
```



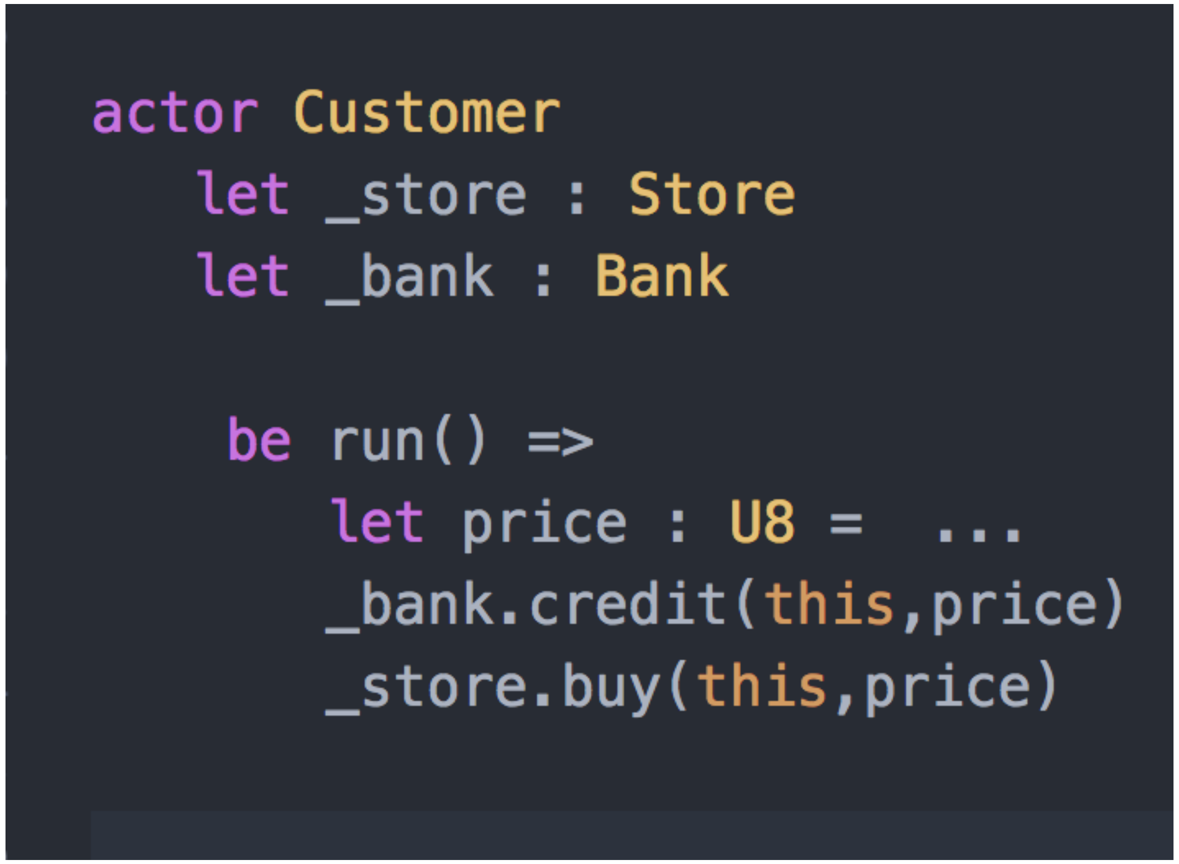
```
actor Bank
  let _balances : MapIs[Customer,U8] ref

  new create(env:Env) =>
    _balances = MapIs[Customer,U8]()
```



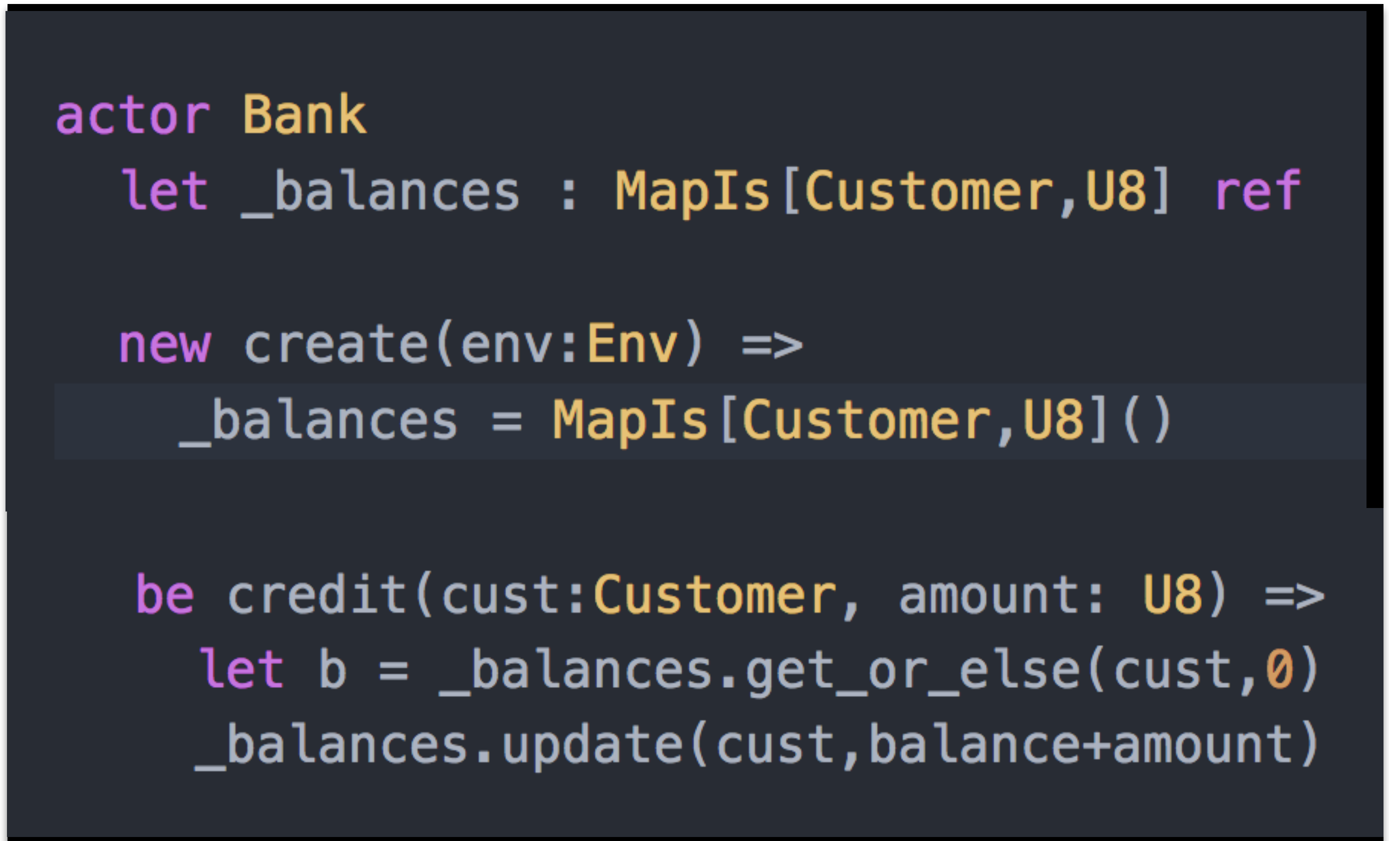
```
actor Store
  let _bank : Bank

  be buy(cust:Customer, price: U8)
    _bank.debit(cust,price)
```

Code in `2_CausalDelivery/MssgDelivery.pony`

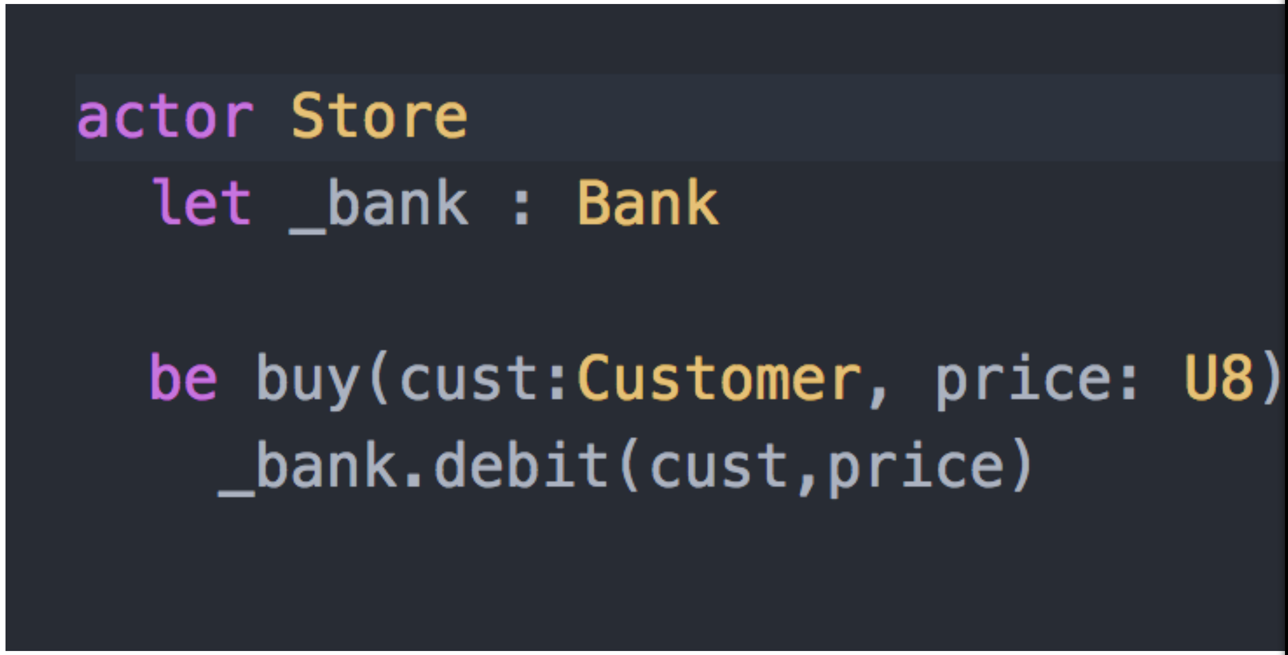
Message Delivery



```
actor Customer
   let _store : Store
   let _bank : Bank

    be run() =>
       let price : U8 =  ...
       _bank.credit(this,price)
       _store.buy(this,price)
```



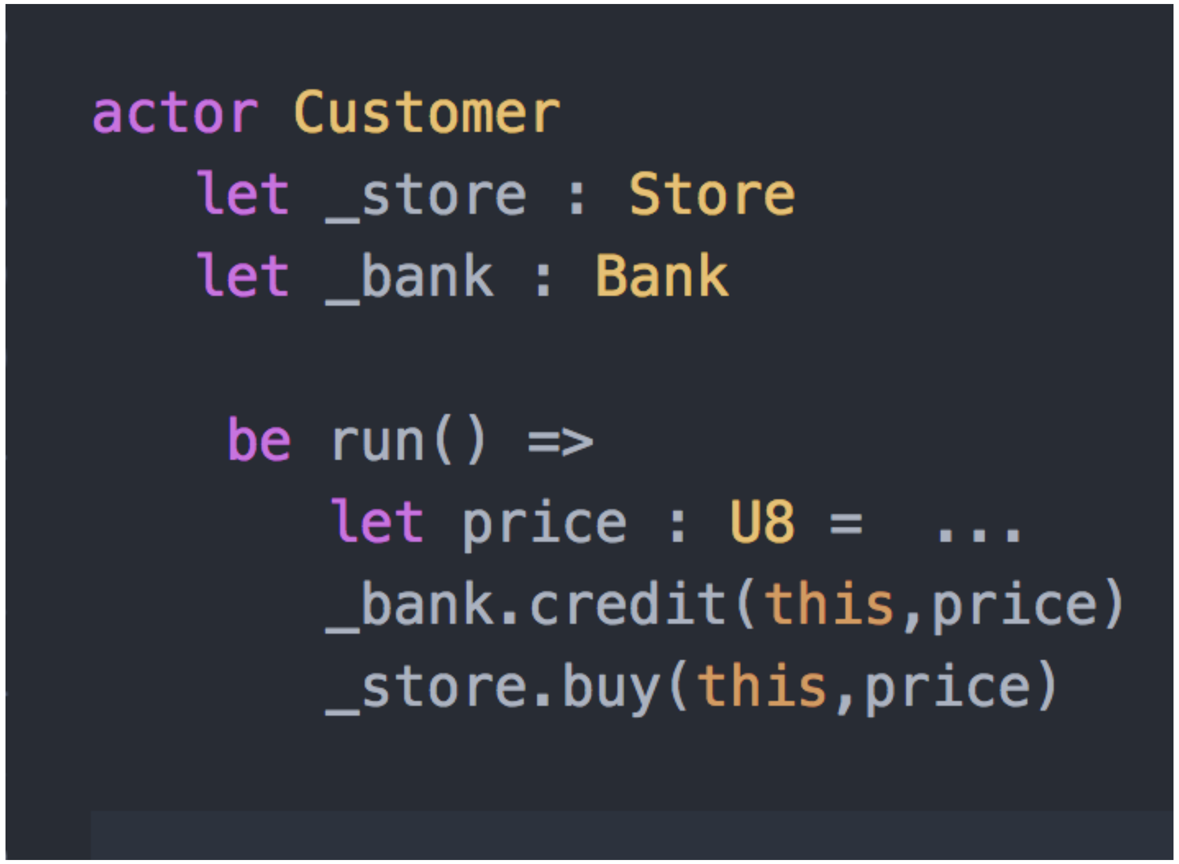
```
actor Bank
  let _balances : MapIs[Customer,U8] ref

  new create(env:Env) =>
    _balances = MapIs[Customer,U8]()

   be credit(cust:Customer, amount: U8) =>
     let b = _balances.get_or_else(cust,0)
     _balances.update(cust,balance+amount)
```



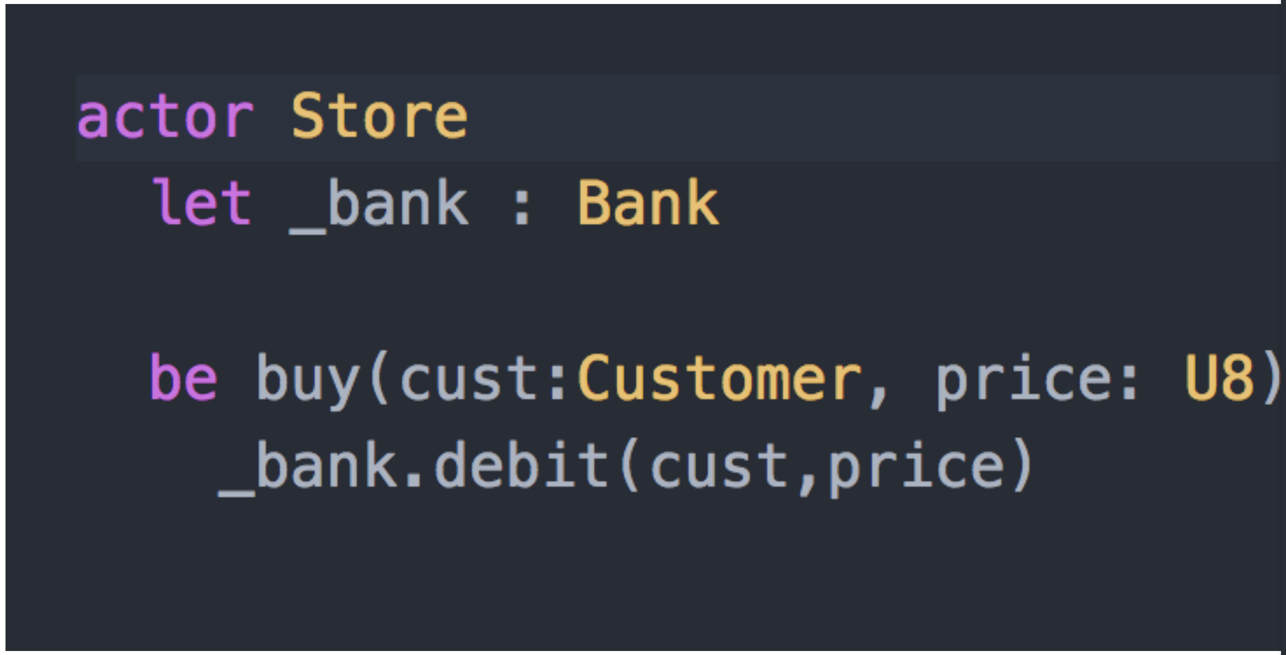
```
actor Store
  let _bank : Bank

  be buy(cust:Customer, price: U8)
    _bank.debit(cust,price)
```

Code in   `2_CausalDelivery/MssgDelivery.pony`

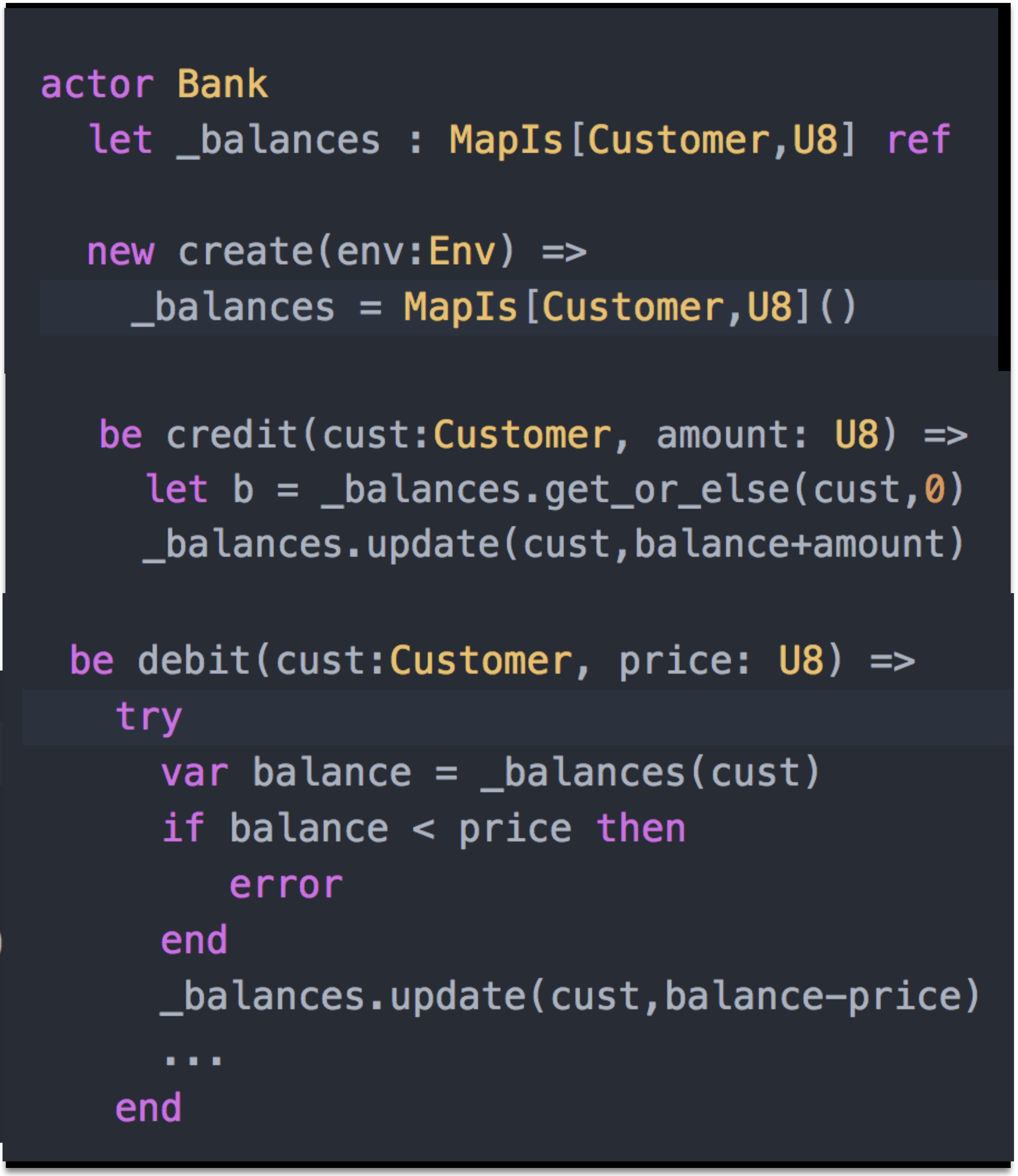
Message Delivery



```
actor Customer
    let _store : Store
    let _bank : Bank

     be run() =>
        let price : U8 =  ...
        _bank.credit(this,price)
        _store.buy(this,price)
```



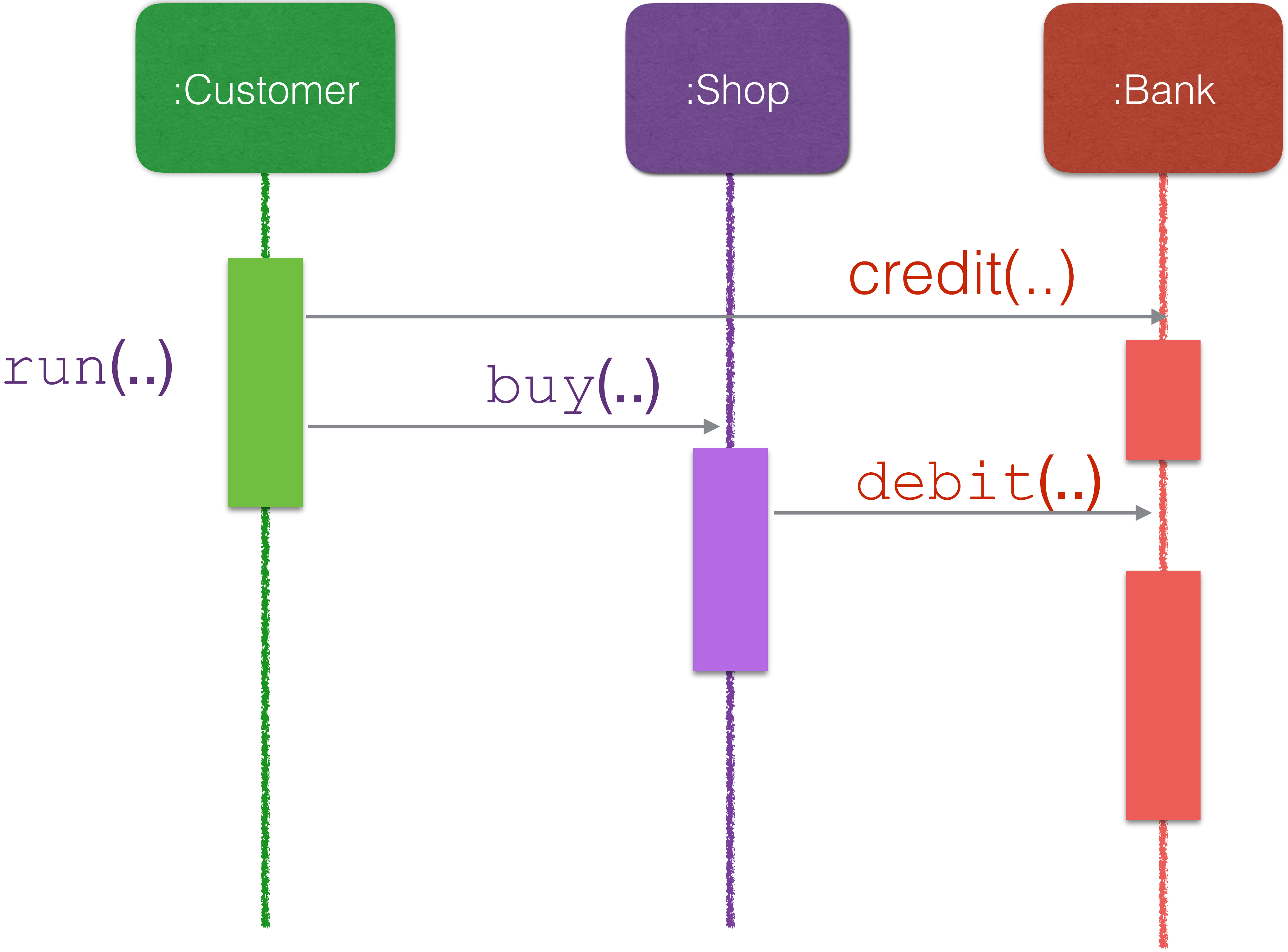
```
actor Store
  let _bank : Bank

  be buy(cust:Customer, price: U8)
    _bank.debit(cust,price)
```



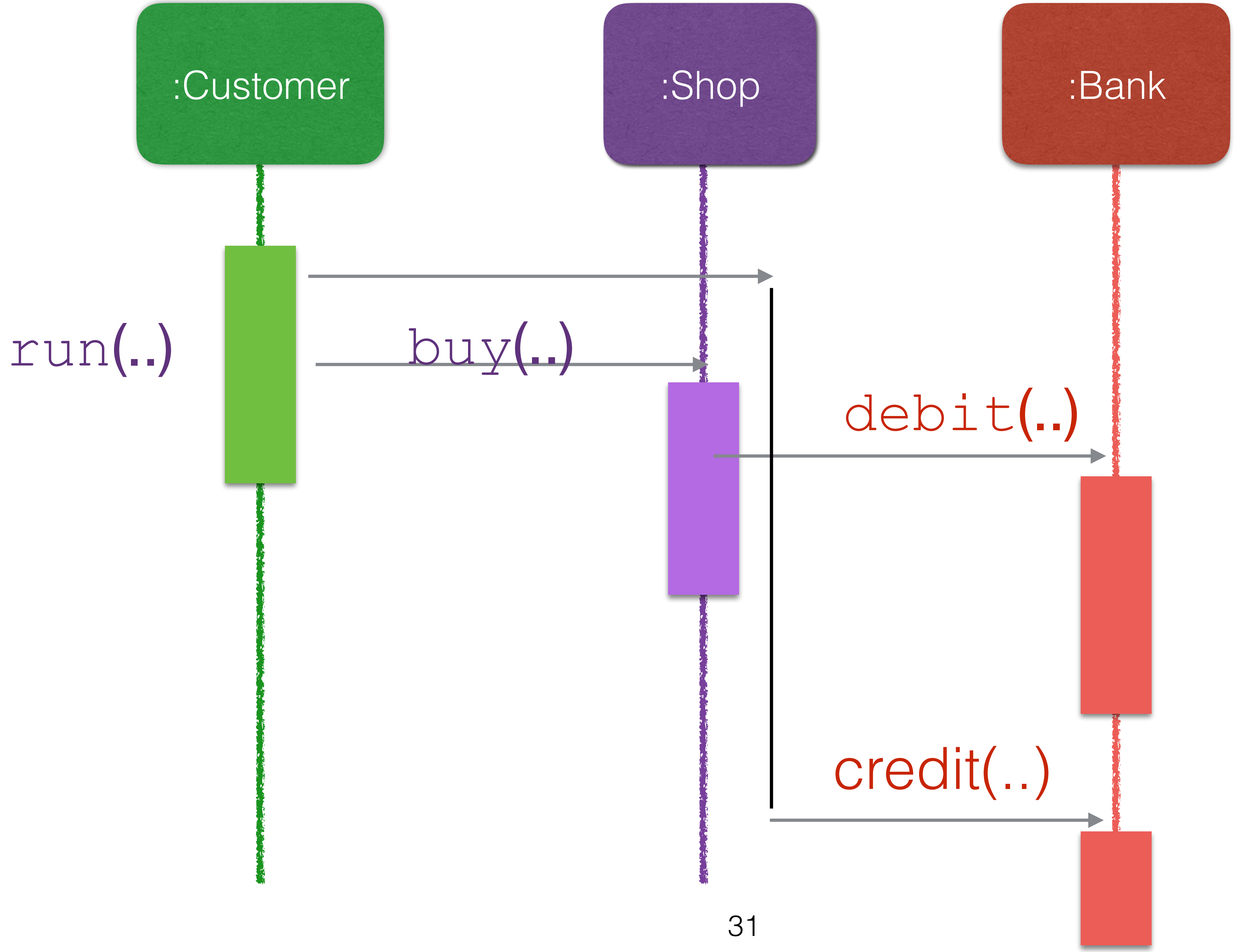
```
actor Bank
  let _balances : MapIs[Customer,U8] ref

  new create(env:Env) =>
    _balances = MapIs[Customer,U8]()

   be credit(cust:Customer, amount: U8) =>
     let b = _balances.get_or_else(cust,0)
     _balances.update(cust,balance+amount)

 be debit(cust:Customer, price: U8) =>
   try
     var balance = _balances(cust)
     if balance < price then
        error
     end
     _balances.update(cust,balance-price)
     ...
   end
```

Code in `2_CausalDelivery/MssgDelivery.pony`

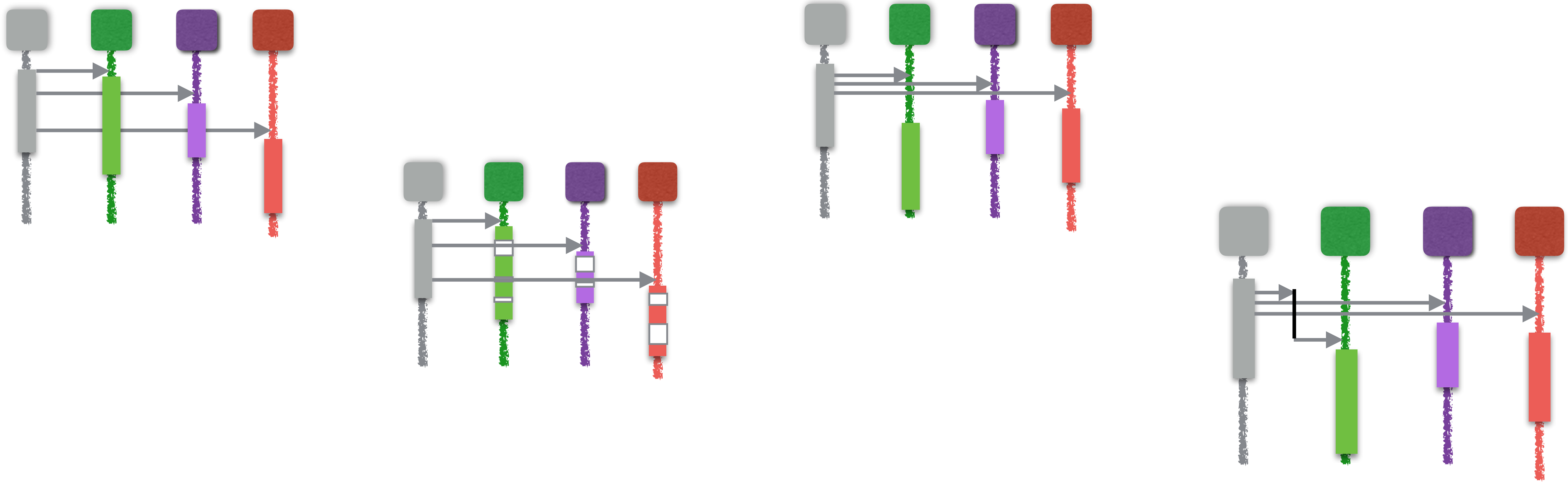
Customer.run()



Customer.run() — can be?



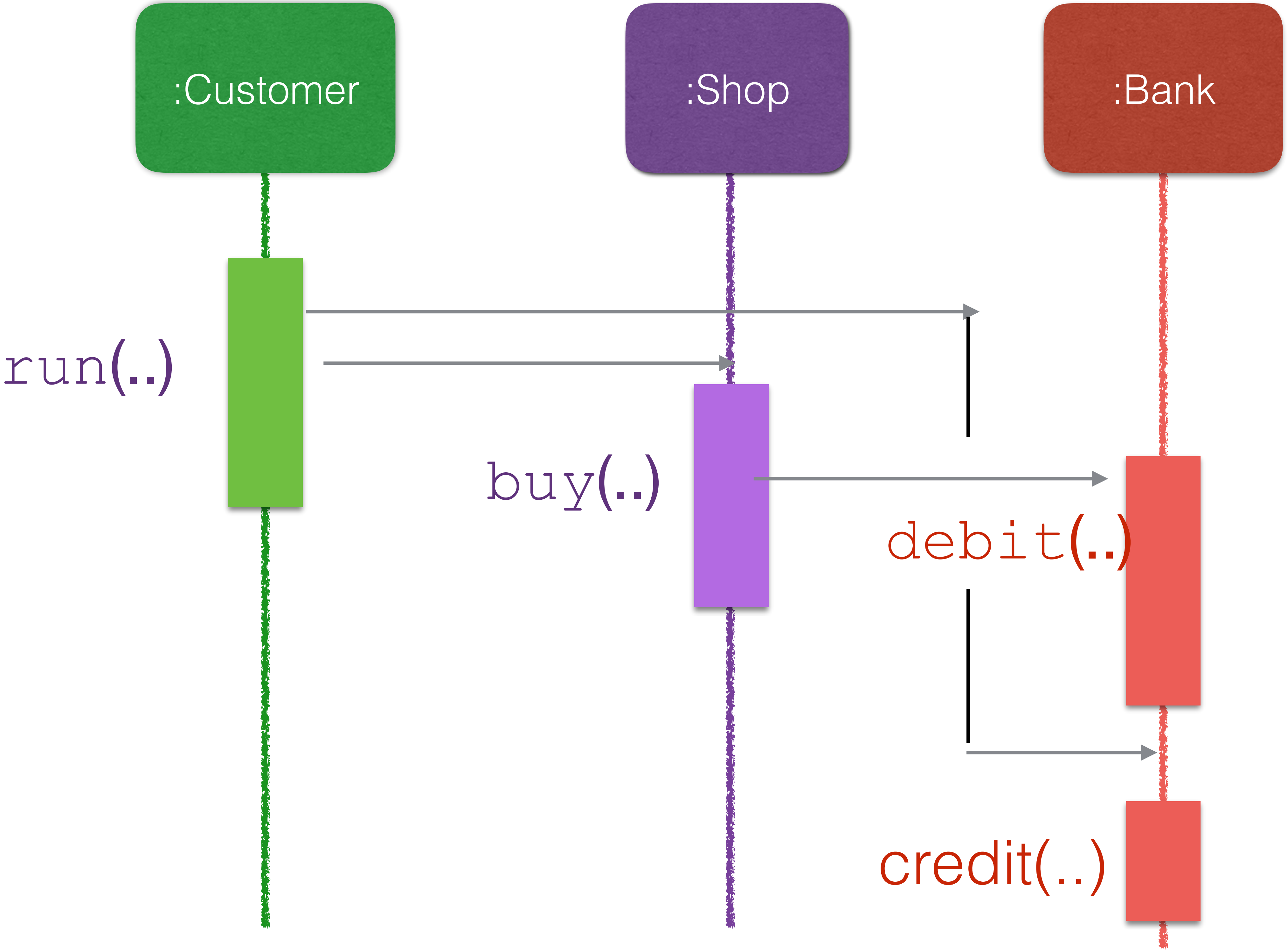
Uncertainty alleviated through **causality**



- What do the other actors do while I am executing?
- When will the message be taken off the queue?
- When will the message be delivered to the queue?

- Messages arrive at queues in *causal* order
  - If I receive **m**, and then send **m'**, then **m** *causes* **m'**
  - If I send **m** and then send **m'**, then **m** *causes* **m'**
  - causality is transitive

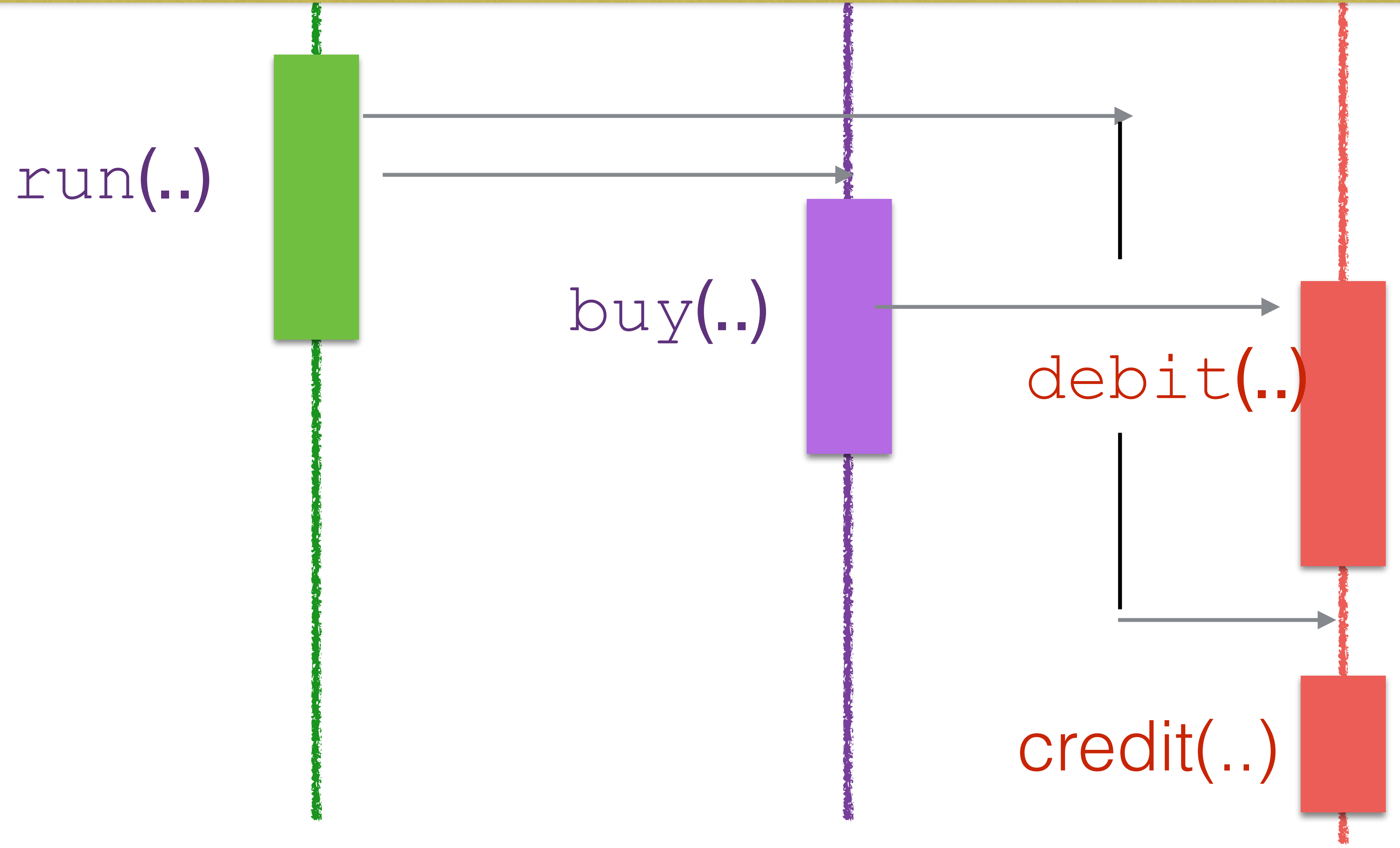
Customer.run() — can be?



This scenario cannot happen, because:

- * Customer sends credit and then buy; ie credit causes buy,
- * Shop receives buy and sends debit; ie buy causes debit,

Therefore credit causes debit.

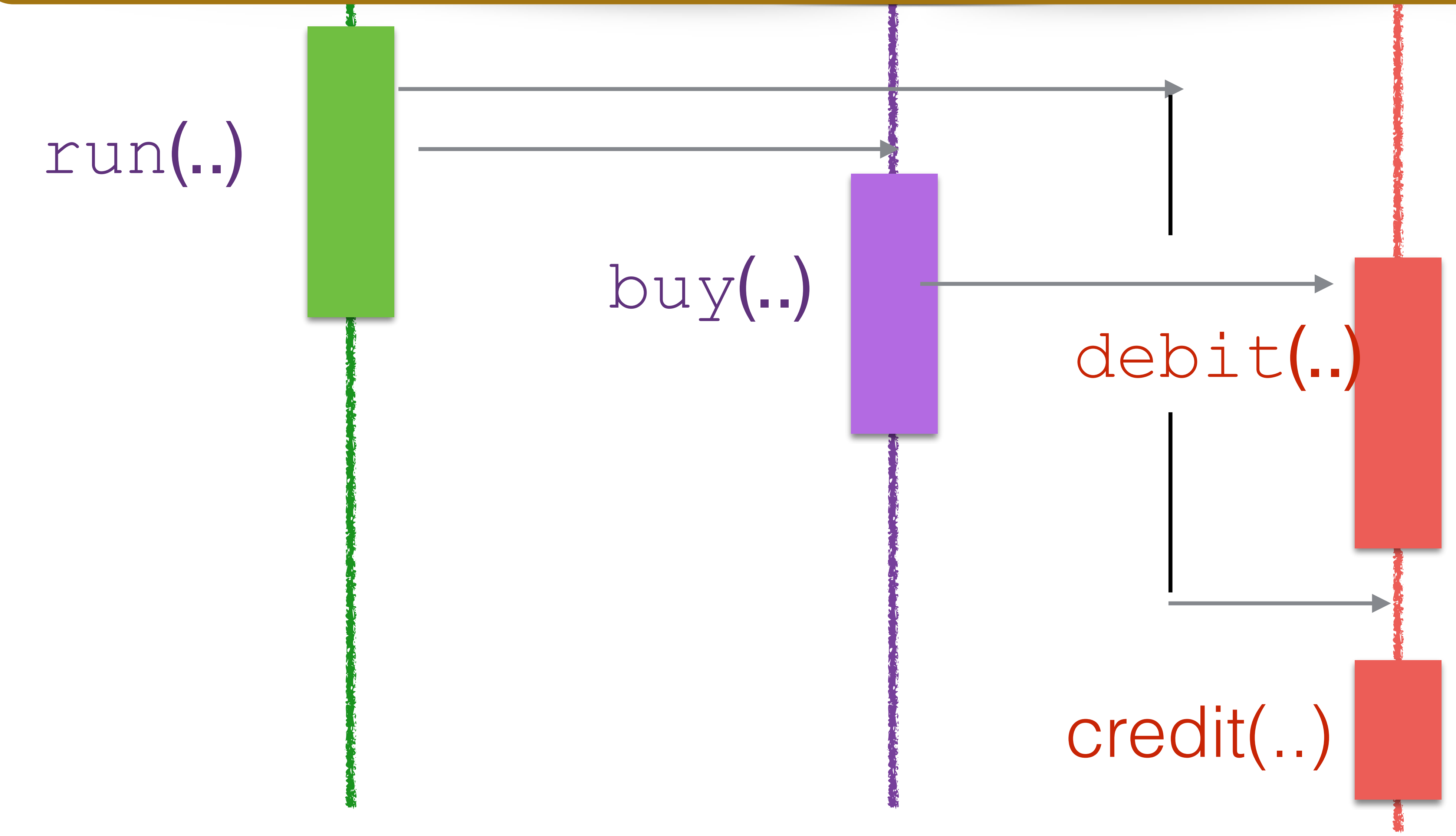


This scenario cannot happen, because:

- * Customer sends credit and then buy; ie credit causes buy,
- * Shop receives buy and sends debit; ie buy causes debit,

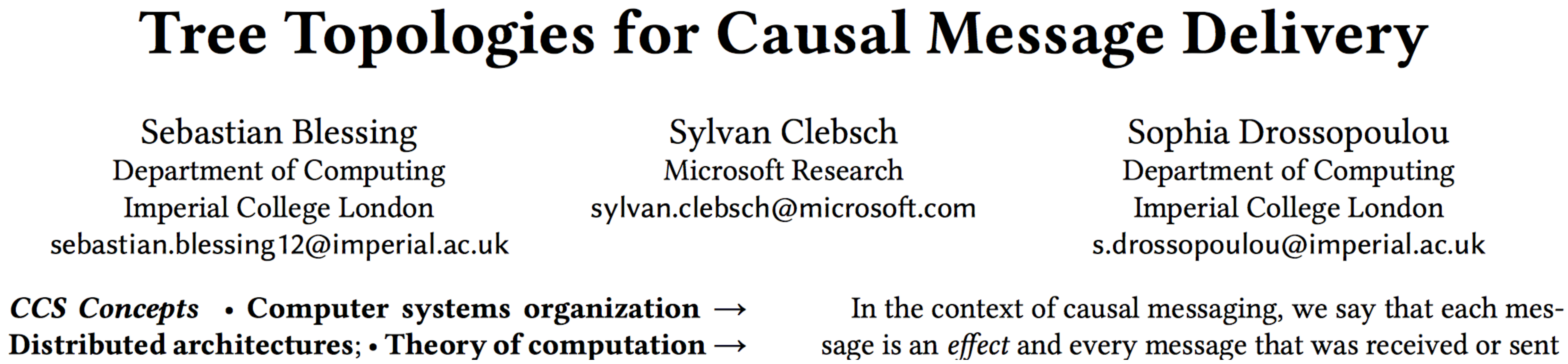
Therefore credit causes debit.

Therefore credit will be delivered before debit



Causality & distribution

Causality & distribution



Tree Topologies for Causal Message Delivery

Sebastian Blessing
Department of Computing
Imperial College London
sebastian.blessing12@imperial.ac.uk

Sylvan Clebsch
Microsoft Research
sylvan.clebsch@microsoft.com

Sophia Drossopoulou
Department of Computing
Imperial College London
s.drossopoulou@imperial.ac.uk

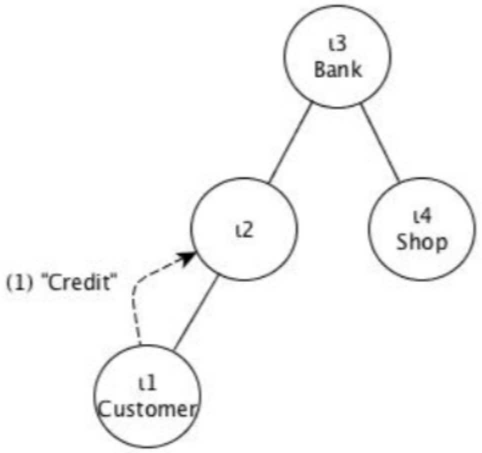
CCS Concepts • **Computer systems organization → Distributed architectures;** • **Theory of computation →**

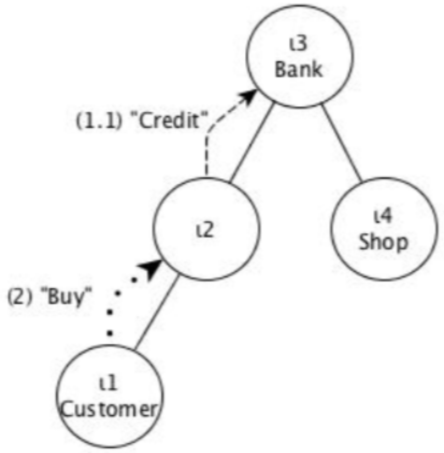
In the context of causal messaging, we say that each message is an *effect* and every message that was received or sent

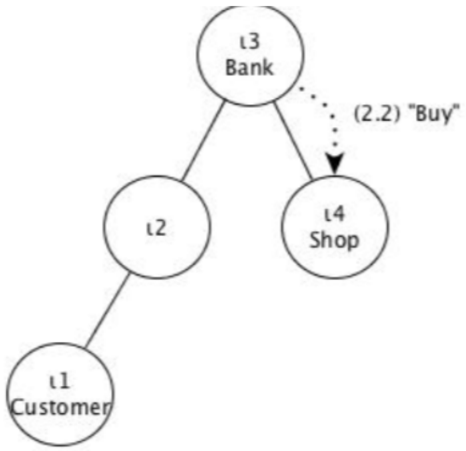
Agere 2017

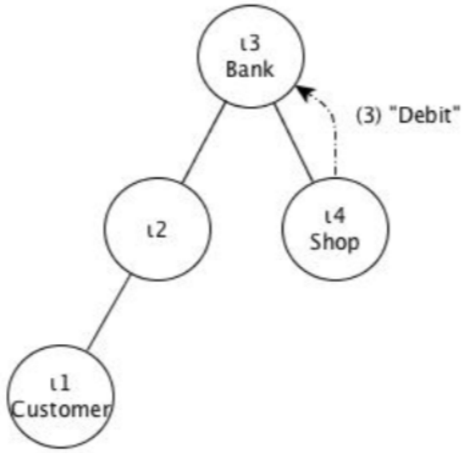
Causality & distribution

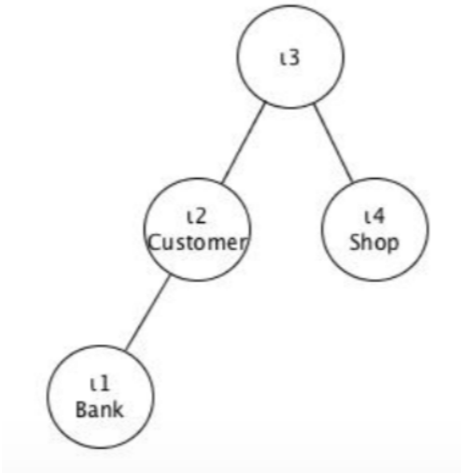
Causality & distribution

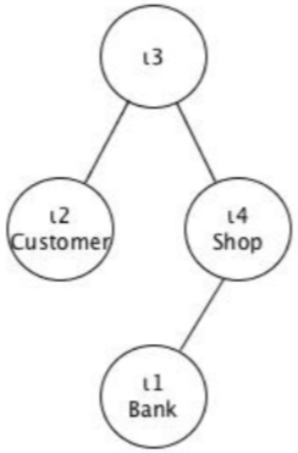












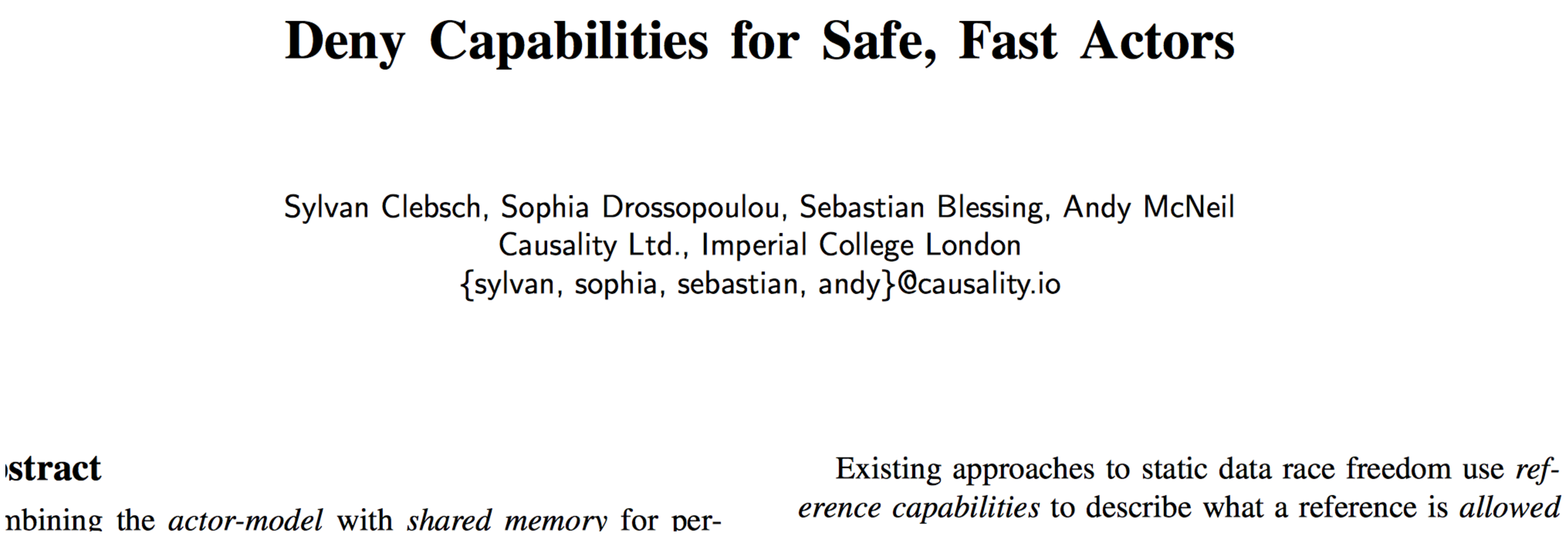
Uncertainty alleviated through Types

Today’s Talk

- Pony - the language and its design
    - Actors
    - Causality
    - **The Type System**
    - The Garbage Collector

The type system

The type system



Deny Capabilities for Safe, Fast Actors

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, Andy McNeil
Causality Ltd., Imperial College London
{sylvan, sophia, sebastian, andy}@causality.io

stract

nbining the *actor-model* with *shared memory* for per-

Existing approaches to static data race freedom use *reference capabilities* to describe what a reference is *allowed*

Agere 2015

Pony types reflect execution

- What may I do with my reference?
- What if I alias my reference?
- What if I un-alias my reference?
- What if I read a field from my reference?
- What if I extract a field from my reference?

Pony types reflect execution

- What may I do with my reference?

reference capabilities: κ

- What if I alias my reference?

- What if I un-alias my reference?

- What if I read a field from my reference?

- What if I extract a field from my reference?

Pony types reflect execution

- What may I do with my reference?

reference capabilities: ***κ***

- What if I alias my reference?

aliasing capabilities: ***κ!***

- What if I un-alias my reference?

- What if I read a field from my reference?

- What if I extract a field from my reference?

Pony types reflect execution

- What may I do with my reference?

  reference capabilities: κ

- What if I alias my reference?

  aliasing capabilities: $\kappa!$

- What if I un-alias my reference?

  unaliasing capabilities: $\kappa^\wedge$

- What if I read a field from my reference?

- What if I extract a field from my reference?

Pony types reflect execution

- What may I do with my reference?

  reference capabilities: ***κ***

- What if I alias my reference?

  aliasing capabilities: ***κ!***

- What if I un-alias my reference?

  unaliasing capabilities: ***κ^***

- What if I read a field from my reference?

  viewpoint adaptation: ***κ->κ'***

- What if I extract a field from my reference?

Pony types reflect execution

- What may I do with my reference?

  reference capabilities: ***κ***

- What if I alias my reference?

  aliasing capabilities: ***κ!***

- What if I un-alias my reference?

  unaliasing capabilities: ***κ^***

- What if I read a field from my reference?

  viewpoint adaptation: ***κ->κ'***

- What if I extract a field from my reference?

  extracting adaptation: ***κ^->κ'***

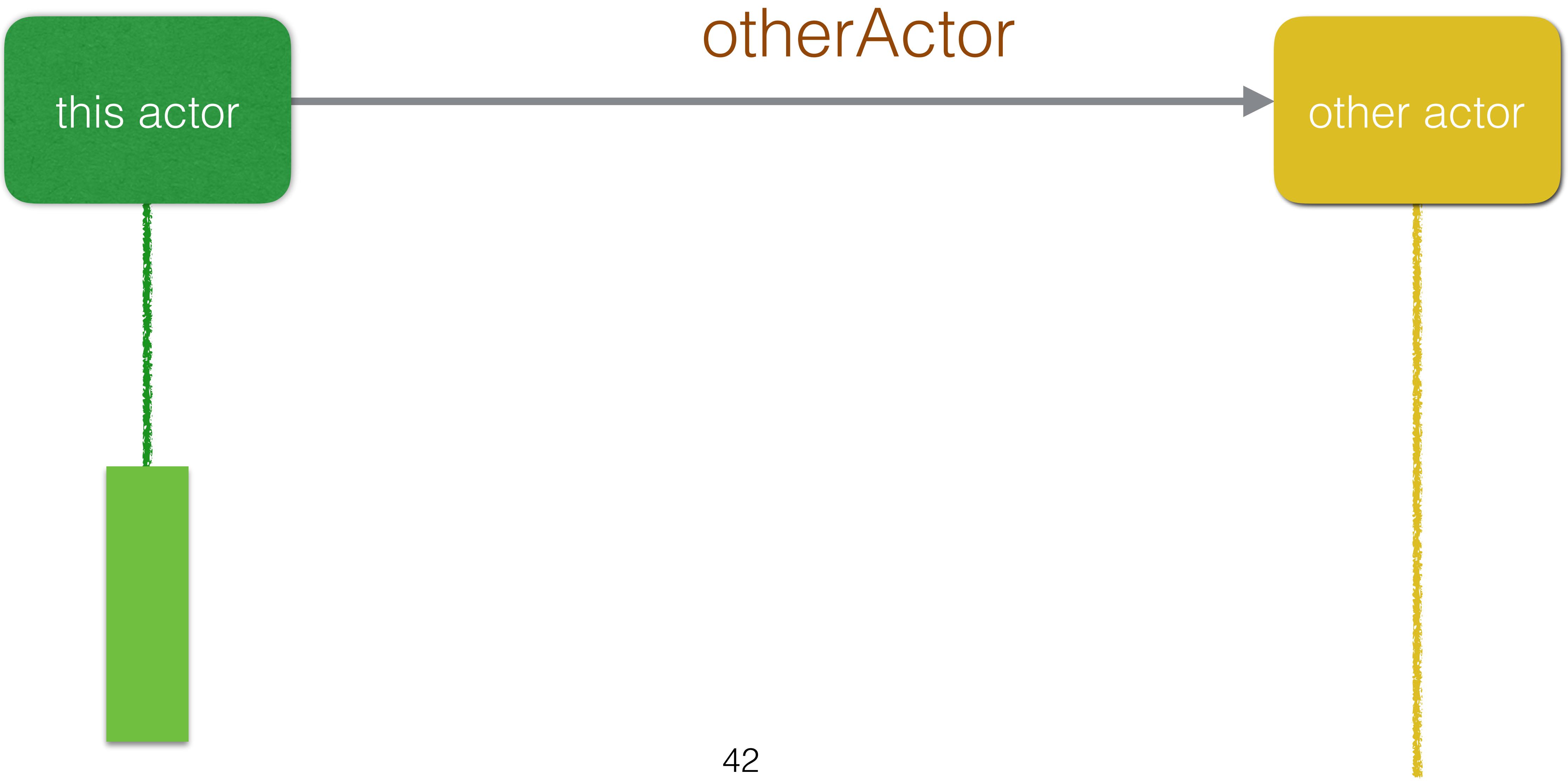
Pony types - 5 novel ingredients

- **reference capabilities: κ**
- aliasing a capability: κ**!**
- consuming (unaliasing) capability: κ**^**
- viewpoint adaptation: κ **->** κ'
- extracting adaptation: κ ^**->** κ'

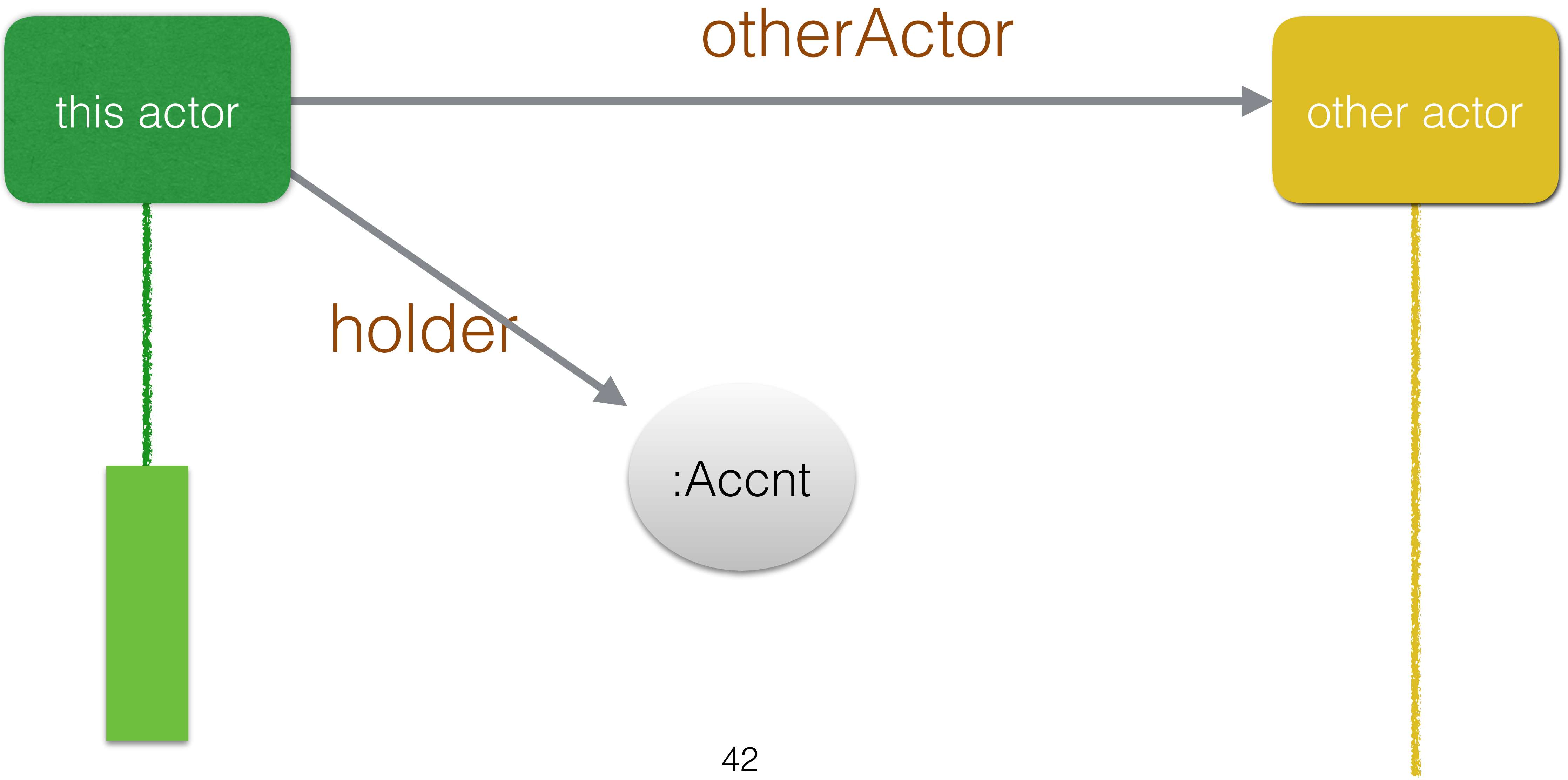
Reference capabilities κ

- attached to references (ie paths).
- express whether *holder* of a reference to an object is allowed to read or write into the object
- also express whether *other aliases* to the object *might* read or write into the object

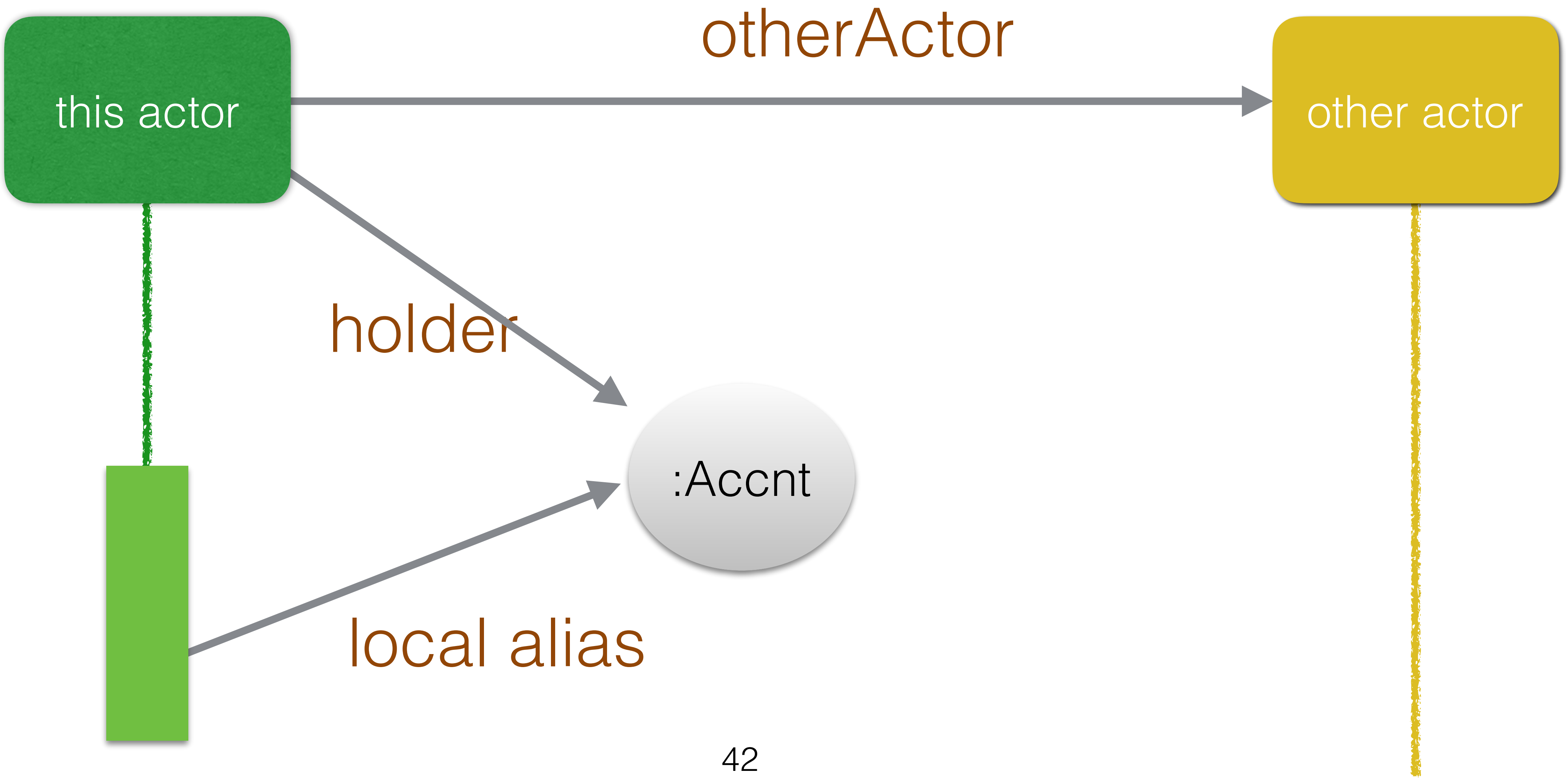
```
actor ThisActor
  let otherActor: OtherActor
```



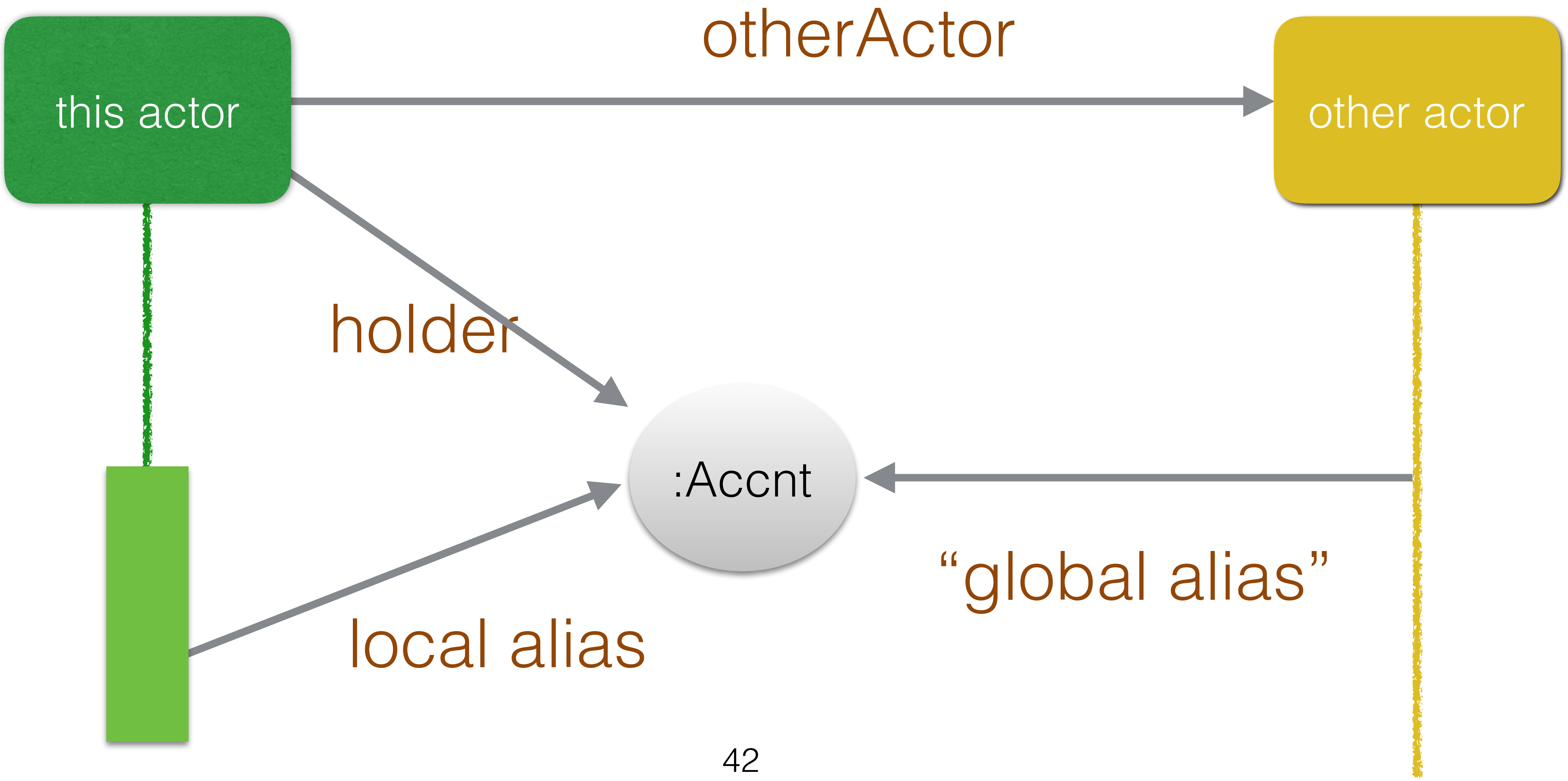
```
actor ThisActor
  let otherActor: OtherActor
  let holder: … = new Account
```



```
actor ThisActor
   let otherActor: OtherActor
   let holder: … = new Account

    fun … =>
        var localAlias = holder
```



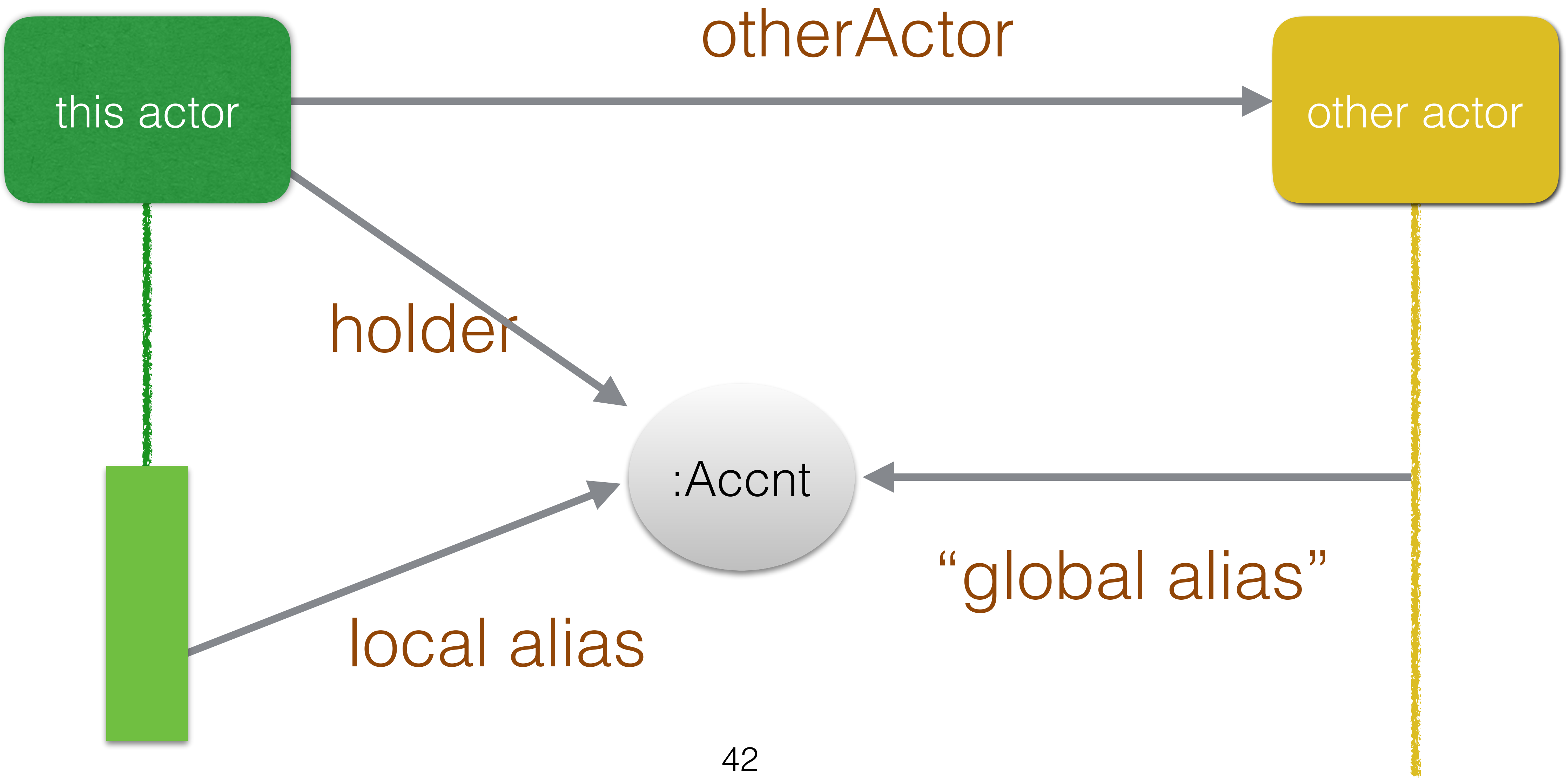
```
actor ThisActor
   let otherActor: OtherActor
   let holder: … = new Account

    fun … =>
        var localAlias = holder
        otherAcror.take(holder)
```



```
actor ThisActor
```

Holder’s capability has to be compatible with

- possible actions of the local alias,
- possible actions of the global alias



reference capabilities (adapted from morning paper) - and omitting **trn** -

reference capabilities (adapted from morning paper) - and omitting **trn** -

iso	
ref	
val	
box	
tag	

reference capabilities (adapted from morning paper) - and omitting **trn** -

	holder may Read, Write?
iso	RD, WR
ref	RD, WR
val	RD
box	RD
tag	—

reference capabilities (adapted from morning paper) - and omitting **trn** -

	holder may Read, Write?	local alias might
iso	RD, WR	—
ref	RD, WR	RD, WR
val	RD	RD
box	RD	RD, WR
tag	—	RD, WR

reference capabilities (adapted from morning paper) - and omitting **trn** -

	holder may Read, Write?	local alias might	global alias might
iso	RD, WR	—	—
ref	RD, WR	RD, WR	—
val	RD	RD	RD
box	RD	RD, WR	RD
tag	—	RD, WR	RD, WR

reference capabilities (adapted from morning paper) - and omitting **trn** -

	holder may Read, Write?	local alias might	global alias might	holder may send?
iso	RD, WR	—	—	✅
ref	RD, WR	RD, WR	—	
val	RD	RD	RD	✅
box	RD	RD, WR	RD	
tag	—	RD, WR	RD, WR	✅

Reference capabilities κ

- attached to references (ie paths, eg x, x.f, x.f.g).
- express whether holder of a reference to an object is allowed to read or write into the object
- also express whether other aliases to the object are denied to read or write to the object
- The type of the receiver is part of function signature

```
fun ref eat(food: Food box) =>
    this.strength = this.strength + food.take_a_bite()
```

capabilities - find the type errors!

Code in `4a_EatingSequential/Eating.pony`

capabilities - find the type errors!

```
class Person
  let id: IdentityData val
  var strength: U64

  fun ref eat(food: Food box) =>
    strength = strength +
            food.take_a_bite()
```

Code in `4a_EatingSequential/Eating.pony`

capabilities - find the type errors!

```
class Person
  let id: IdentityData val
  var strength: U64

  fun ref eat(food: Food box) =>
    strength = strength +
            food.take_a_bite()


class Food
  var calories: U64

  fun box take_a_bite( ):U64 =>
    calories = calories/2
    calories/3
```

Code in `4a_EatingSequential/Eating.pony`

capabilities - find the type errors!

```
class Person
  let id: IdentityData val
  var strength: U64

  fun ref eat(food: Food box) =>
    strength = strength +
            food.take_a_bite()


class Food
  var calories: U64

  fun box take_a_bite( ):U64 =>
    calories = calories/2
    calories/3
```

```
actor Main
  let apple: Food ref

  new create(env':Env) =>
    apple = Food("apple",50)
    run()

  be run( )  =>
    let pear: Food ref = Food("pear",160)
    let laurie: Person ref =
                  Person("Laurie",400)
    let jan: Person ref =
                  Person("Jan",300)
    jan.eat(apple)
    laurie.eat(pear)
    jan.eat(pear)
    laurie.eat(apple)
```

Code in `4a_EatingSequential/Eating.pony`

capabilities - correct the type errors!

```
class Person
  let id: IdentityData val
  var strength: U64

  fun ref eat(food: Food box) =>
    strength = strength +
            food.take_a_bite()


class Food
  var calories: U64

  fun box take_a_bite( ):U64 =>
    calories = calories/2
    calories/3
```

```
actor Main
  let apple: Food ref

  new create(env':Env) =>
    apple = Food("apple",50)
    run()

  be run( )  =>
    let pear: Food ref = Food("pear",160)
    let laurie: Person ref =
                  Person("Laurie",400)
    let jan: Person ref =
                  Person("Jan",300)
    jan.eat(apple)
    laurie.eat(pear)
    jan.eat(pear)
    laurie.eat(apple)
```

valid local aliases - find the type errors!

```
class Person
  let id: IdentityData val
  var strength: U64

  fun ref eat(food: Food box) =>
    strength = strength +
            food.take_a_bite()


class Food
  var calories: U64

  fun ref take_a_bite( ):U64 =>
    calories = calories/2
    calories/3
```

```
actor Main
  let apple: Food ref

  new create(env':Env) =>
    apple = Food("apple",50)
    run()

  be run( )  =>
    let pear: Food ref = Food("pear",160)
    let laurie: Person ref =
                 Person("Laurie",400)
    let jan: Person ref =
                 Person("Jan",300)
    jan.eat(apple)
    laurie.eat(pear)
    jan.eat(pear)
    laurie.eat(apple)
```

valid local aliases - correct the type errors!

valid local aliases - correct the type errors!

```
class Person
  let id: IdentityData val
  var strength: U64

  fun ref eat(food: Food box) =>
    strength = strength +
          food.take_a_bite()


class Food
  var calories: U64

  fun ref take_a_bite( ):U64 =>
    calories = calories/2
    calories/3
```

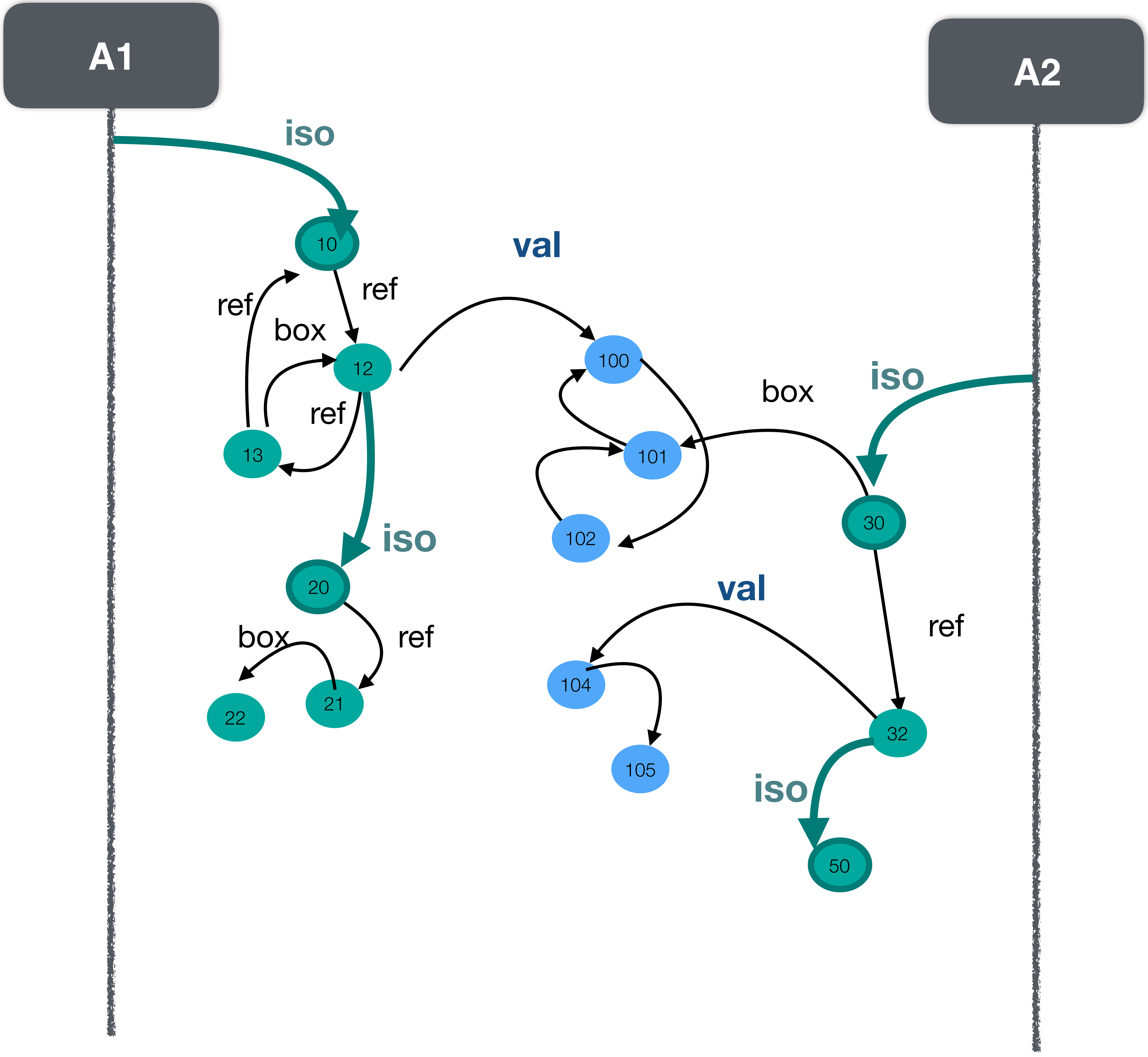
```
actor Main
  let apple: Food ref

  new create(env':Env) =>
    apple = Food("apple",50)
    run()

  be run( )  =>
    let pear: Food ref = Food("pear",160)
    let laurie: Person ref =
                   Person("Laurie",400)
    let jan: Person ref =
                   Person("Jan",300)
    jan.eat(apple)
    laurie.eat(pear)
    jan.eat(pear)
    laurie.eat(apple)
```

valid local aliases - type errors corrected

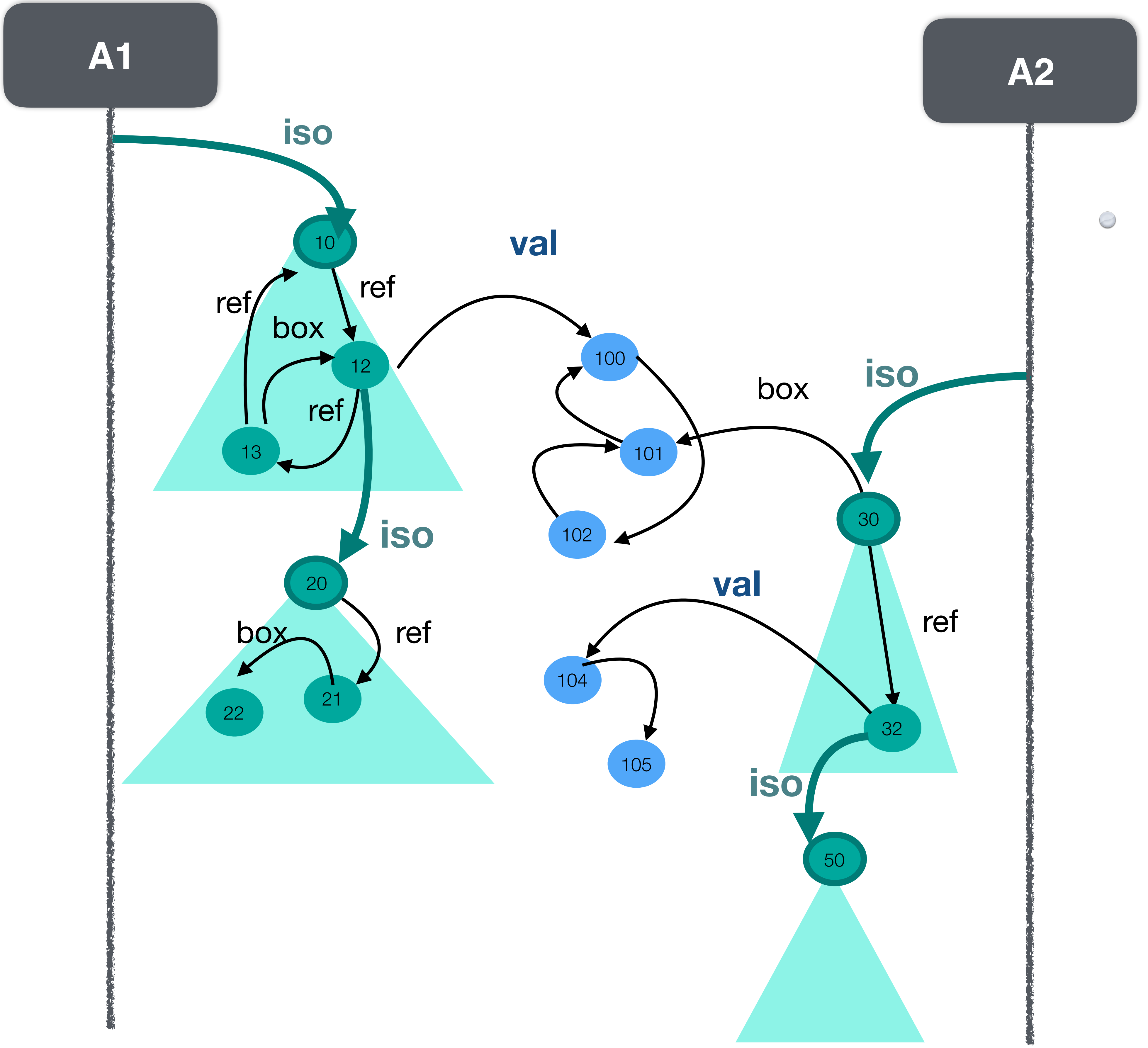
```
class Person
  let id: IdentityData val
  var strength: U64

  fun ref eat(food: Food ref) =>
    strength = strength +
            food.take_a_bite()


class Food
  var calories: U64

  fun ref take_a_bite( ): U64 =>
    calories = calories/2
    calories/3
```

```
actor Main
  let apple: Food ref

  new create(env':Env) =>
    apple = Food("apple",50)
    run()

  be run( )  =>
    let pear: Food ref = Food("pear",160)
    let laurie: Person ref =
                 Person("Laurie",400)
    let jan: Person ref =
                 Person("Jan",300)
    jan.eat(apple)
    laurie.eat(pear)
    jan.eat(pear)
    laurie.eat(apple)
```

Capabilities and Object Heap (excl. tag)

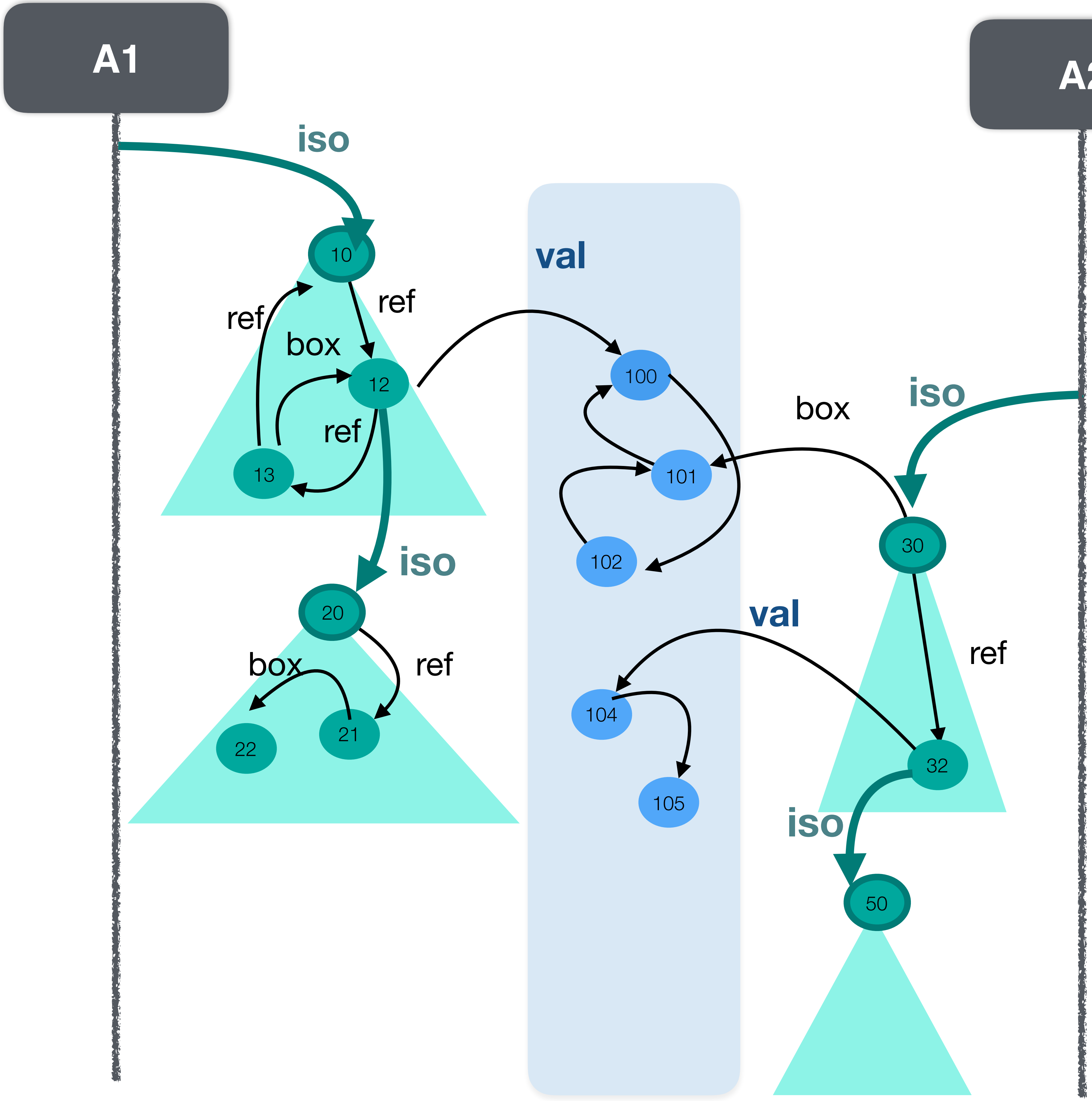


Capabilities and Object Heap (excl. tag)



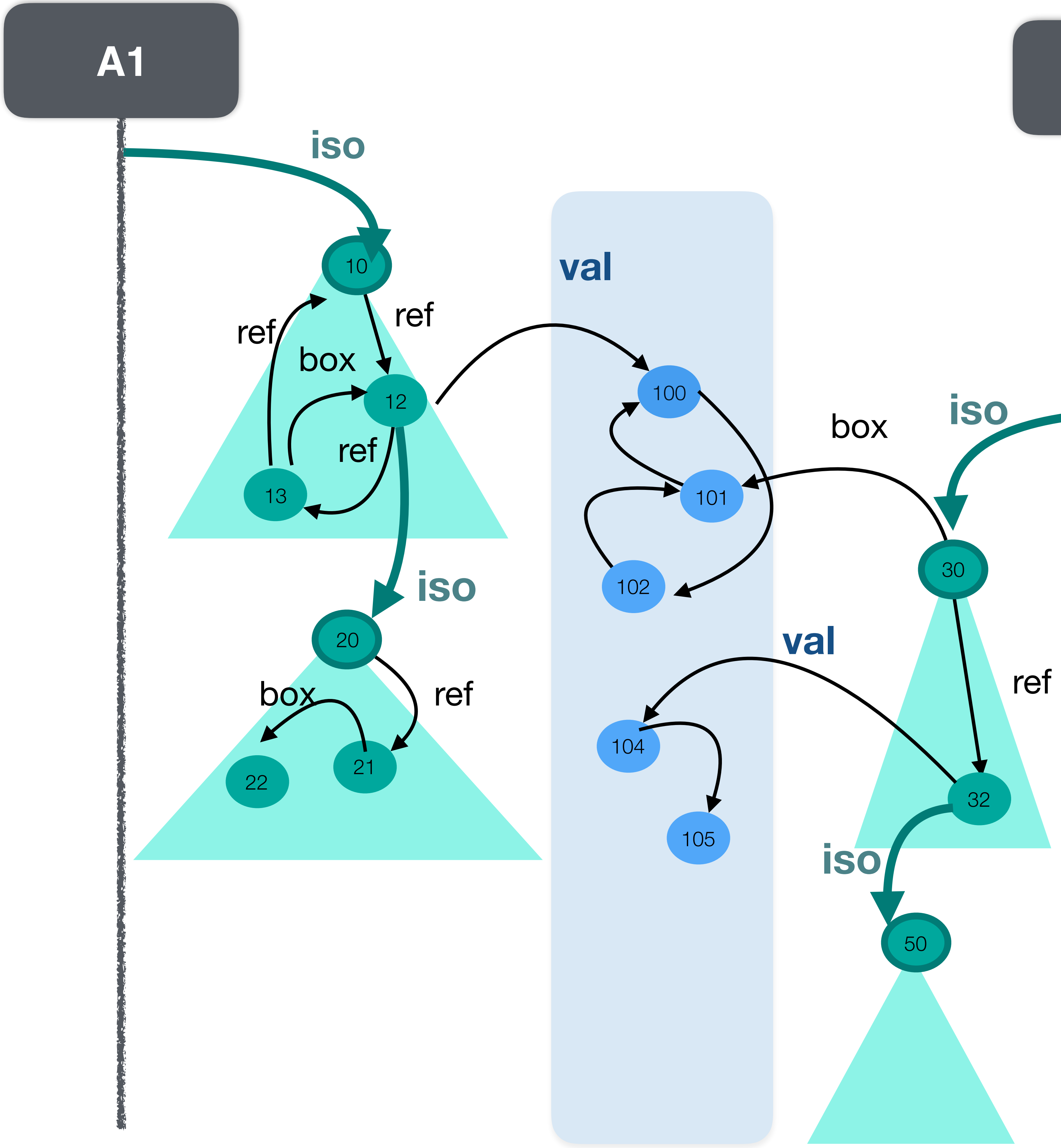
- **iso**: disjoint, mutable regions ▲

Capabilities and Object Heap (excl. tag)



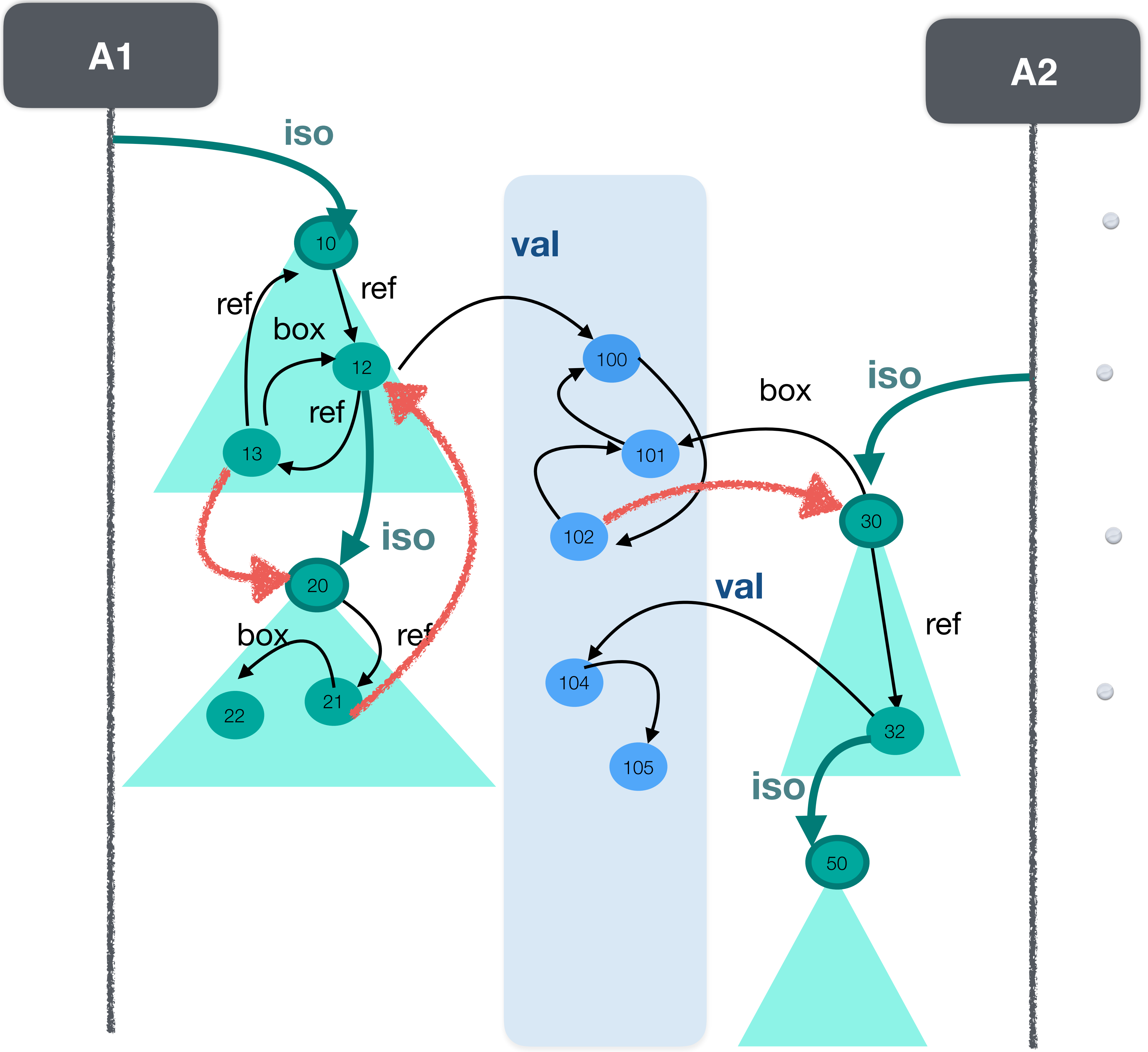
- **iso**: disjoint, mutable regions ▲
- one immutable region ▮

Capabilities and Object Heap (excl. tag)



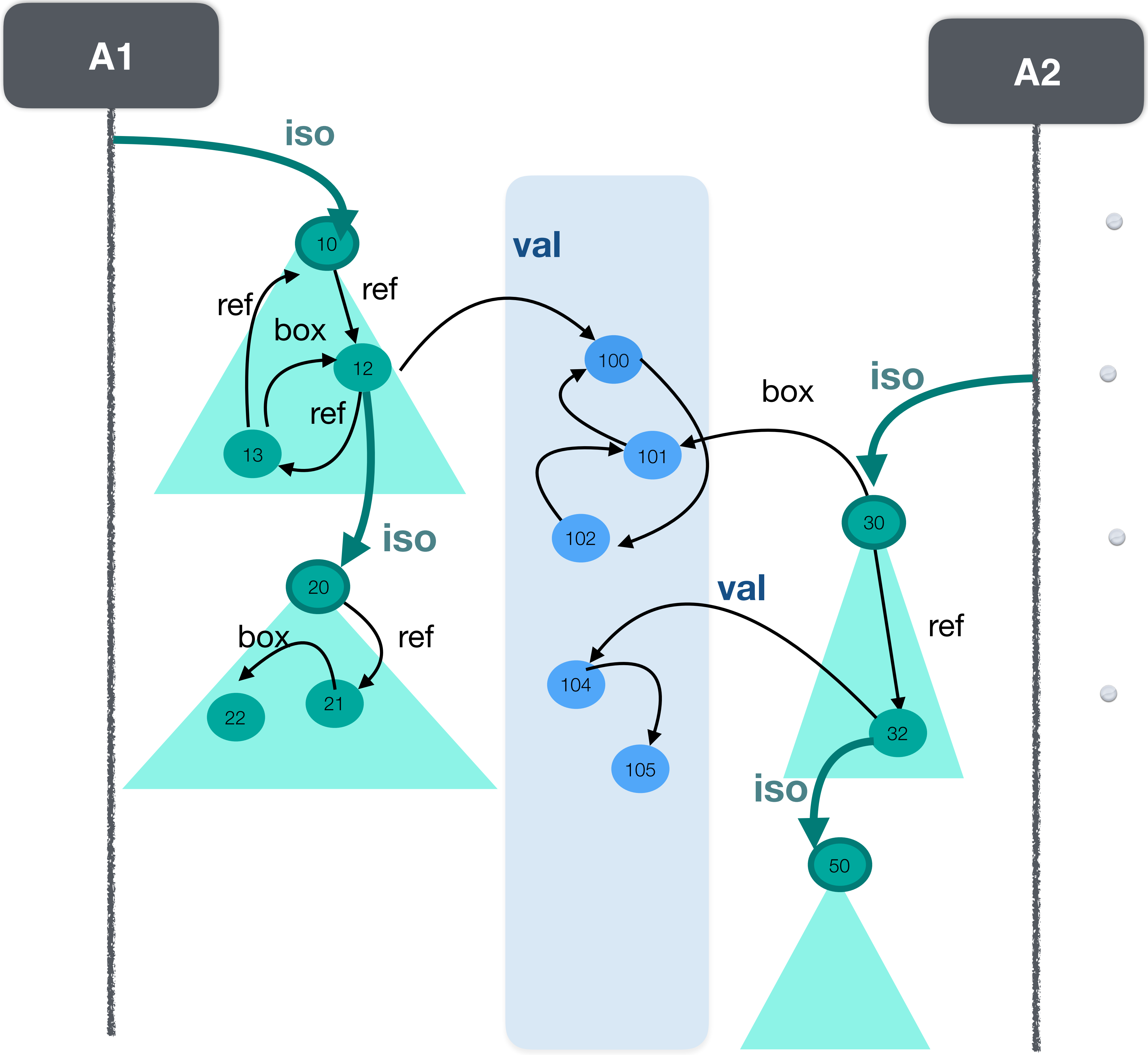
- **iso**: disjoint, mutable regions
- one immutable region
- cycles possible (cf Rust)

Capabilities and Object Heap (excl. tag)



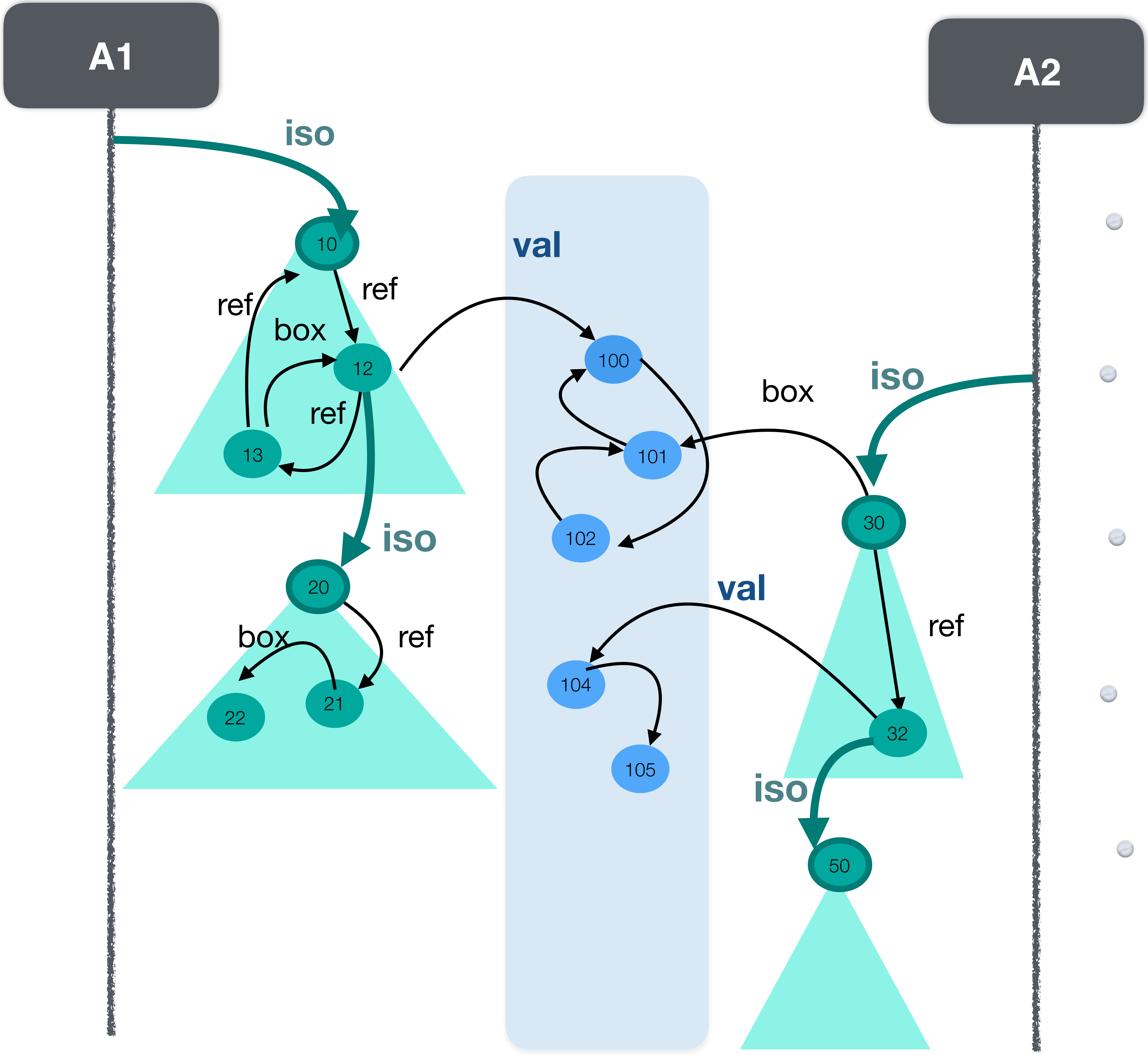
- **iso**: disjoint, mutable regions ▲
- one immutable region ▮
- cycles possible (cf Rust)
- no incoming references into mutable regions

Capabilities and Object Heap (excl. tag)



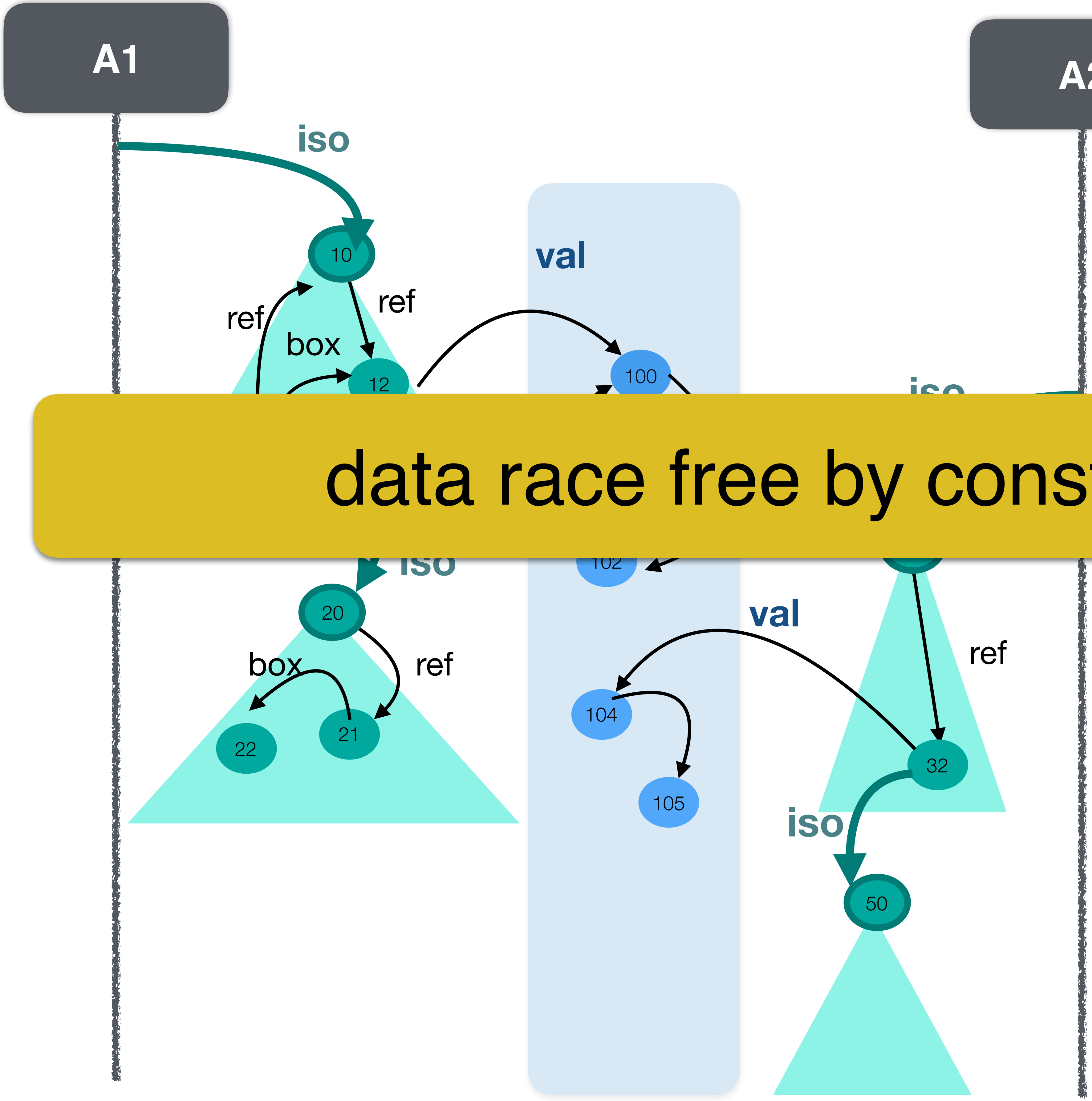
- **iso**: disjoint, mutable regions ▲
- one immutable region ▮
- cycles possible (cf Rust)
- no incoming references into mutable regions

Capabilities and Object Heap (excl. tag)



- **iso**: disjoint, mutable regions ▲
- one immutable region ▯
- cycles possible (cf Rust)
- no incoming references into mutable regions
- at most one actor at a time has access to mutable region

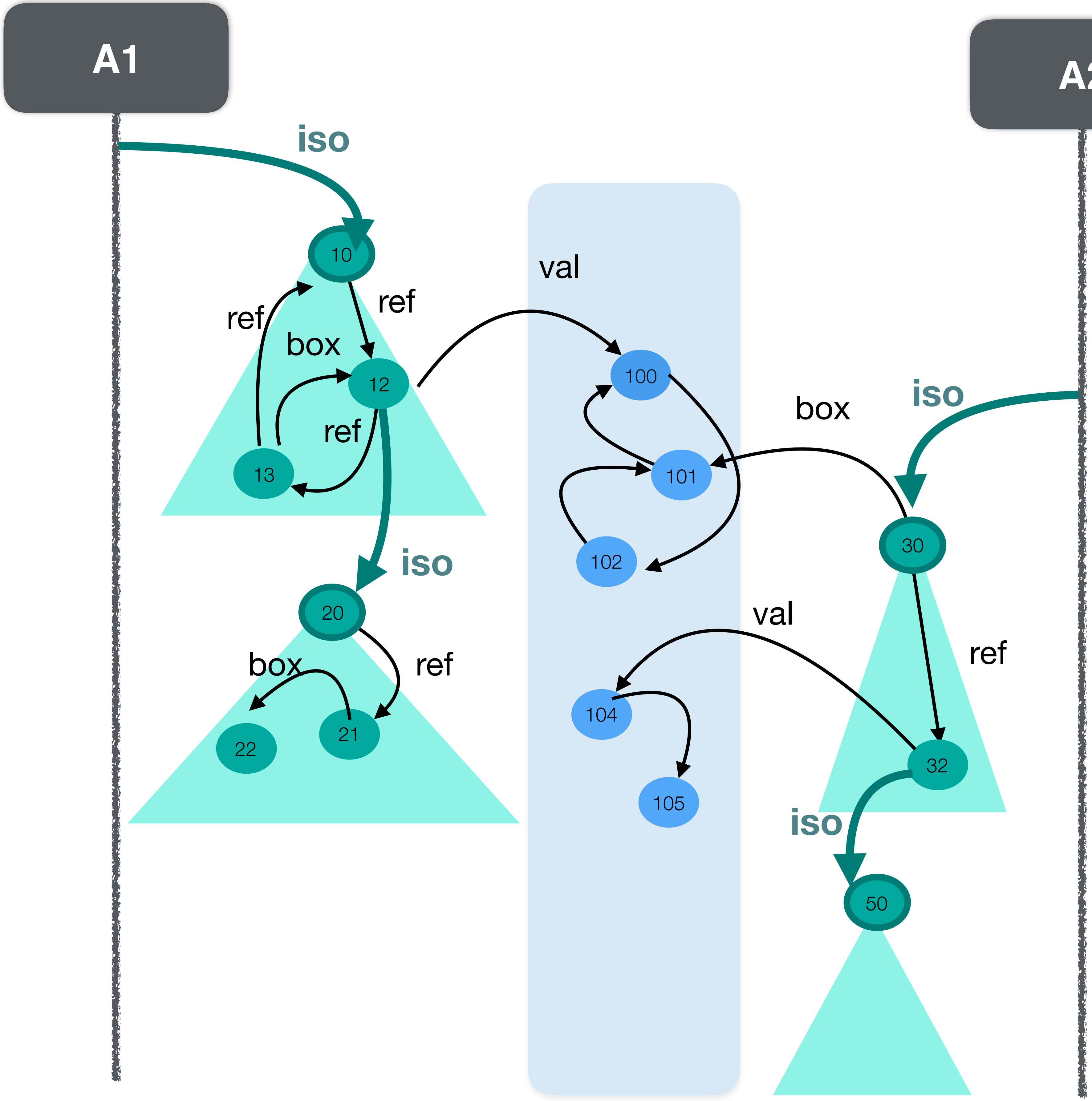
Capabilities and Object Heap (excl. tag)



- **iso**: disjoint, mutable regions
- one
- cycles possible (cf Rust)
- no incoming references into mutable regions
- at most one actor at a time has access to mutable region

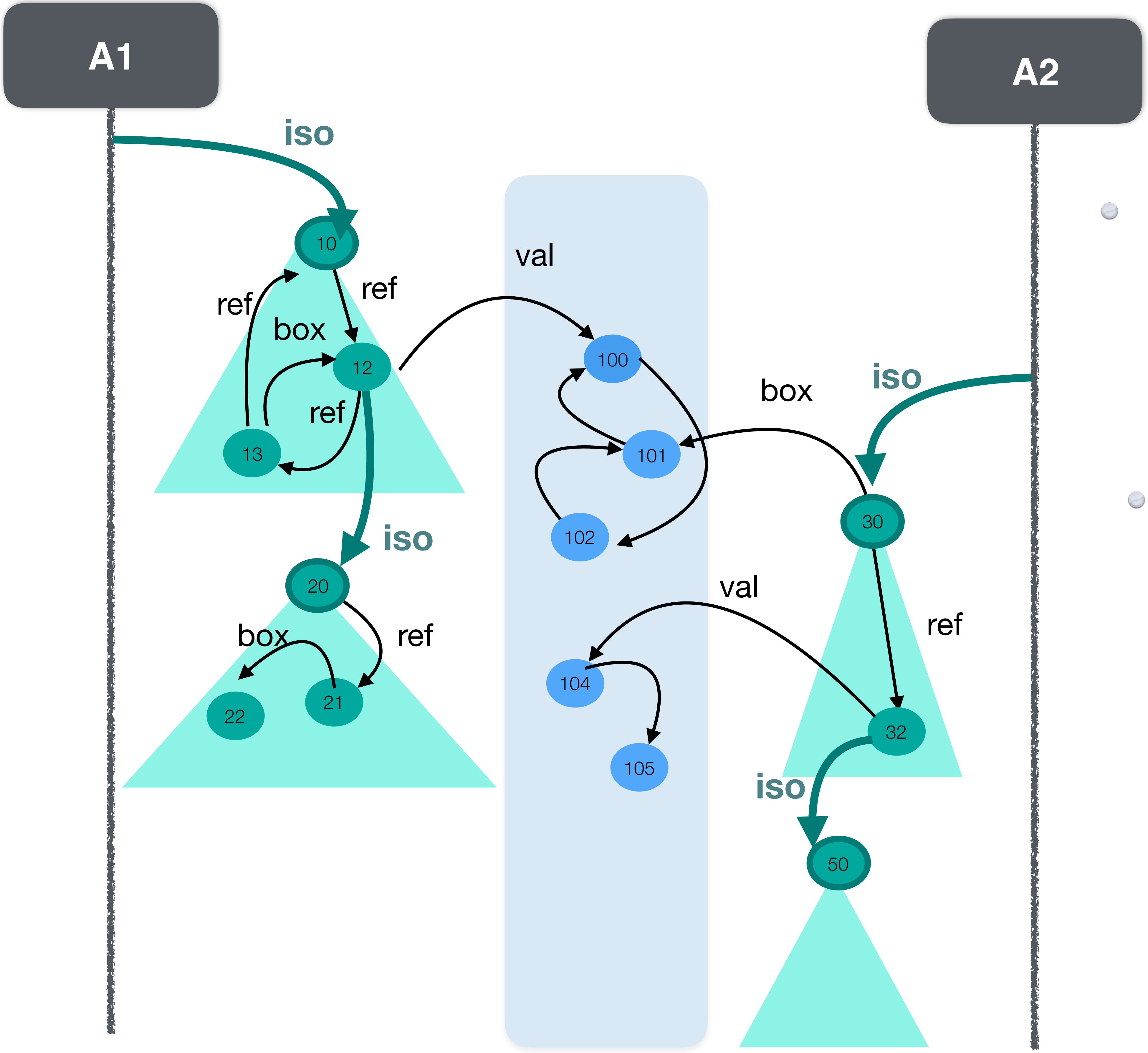
data race free by construction

Capabilities and Safe Communication



- at most one actor at a time has access to mutable region

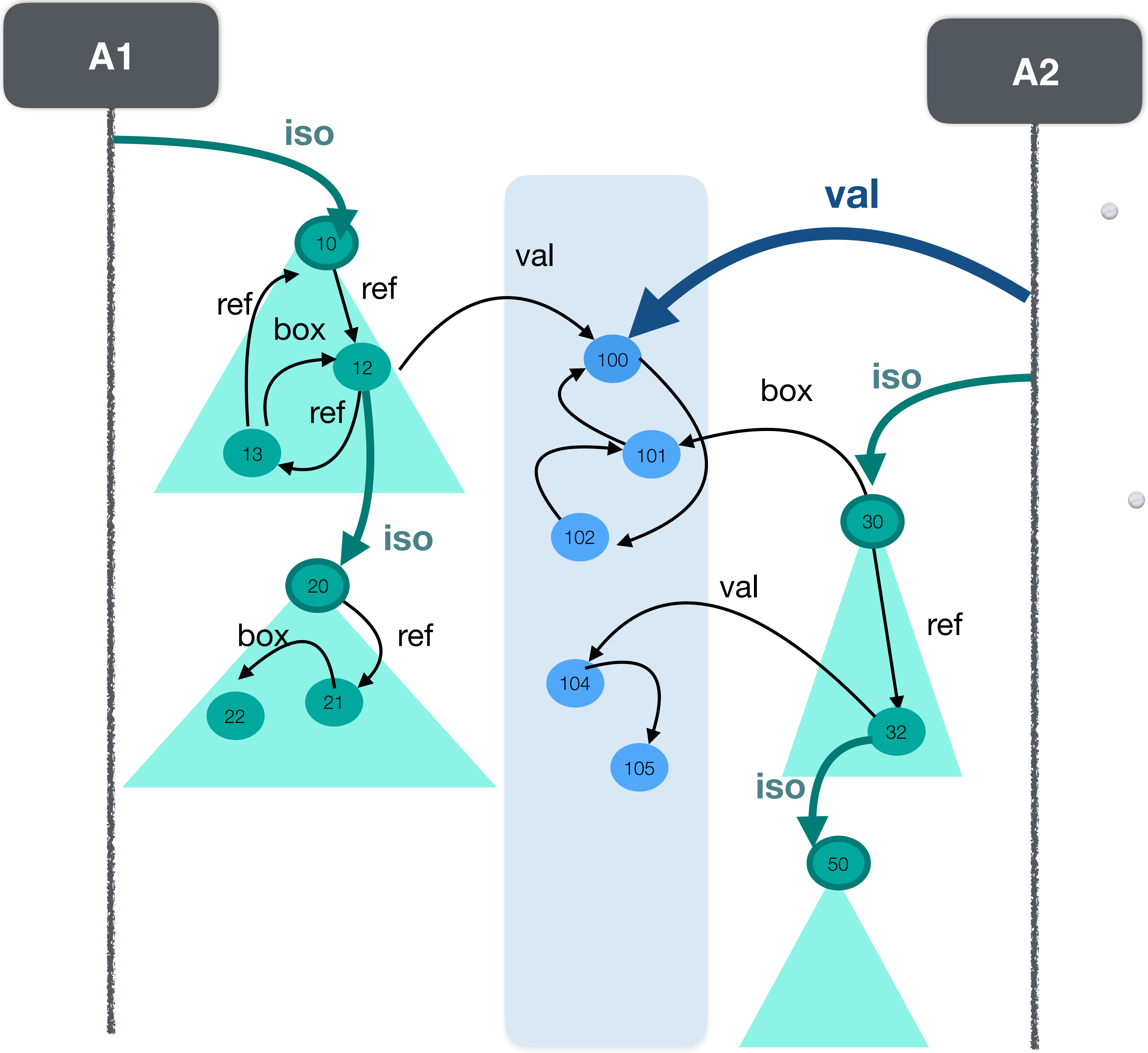
Capabilities and Safe Communication



- at most one actor at a time has access to mutable region

- **val**-reference may be sent to other actor

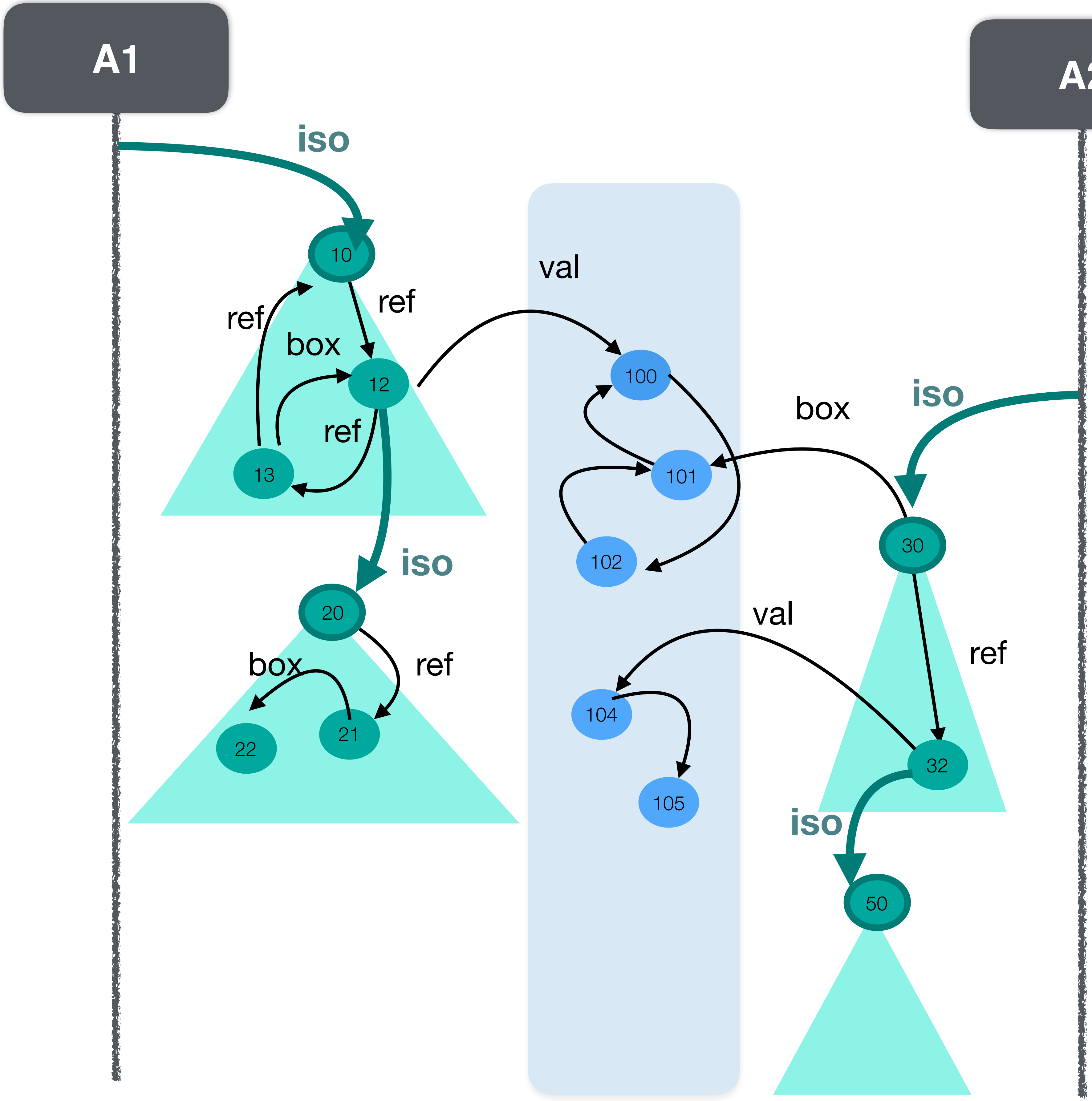
Capabilities and Safe Communication



- at most one actor at a time has access to mutable region

- **val**-reference may be sent to other actor

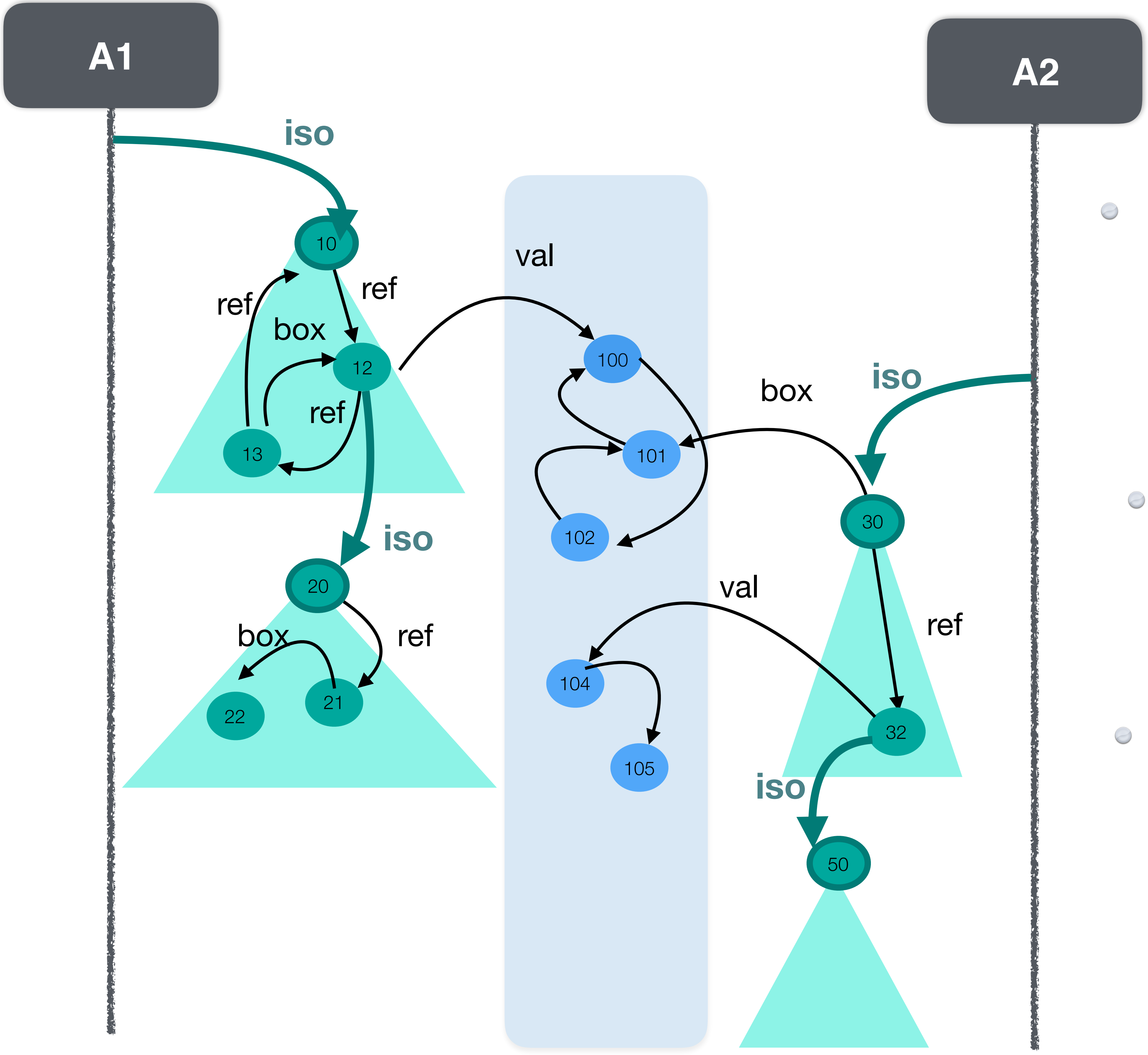
Capabilities and Safe Communication



- at most one actor at a time has access to mutable region

- **val**-reference may be sent to other actor

Capabilities and Safe Communication

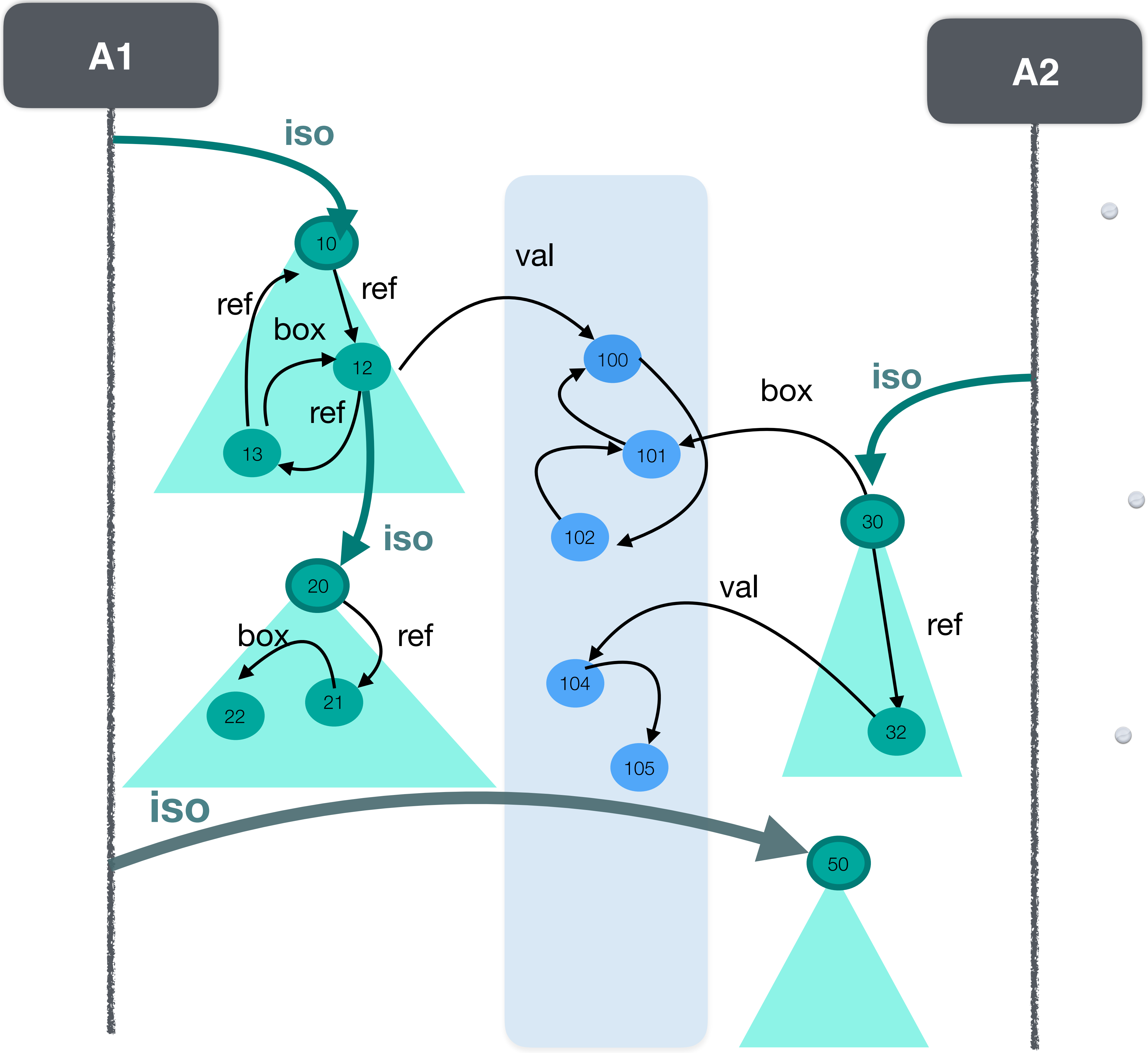


- at most one actor at a time has access to mutable region

- **val**-reference may be sent to other actor

- **iso**:-reference may be given up, and sent to other actor

Capabilities and Safe Communication

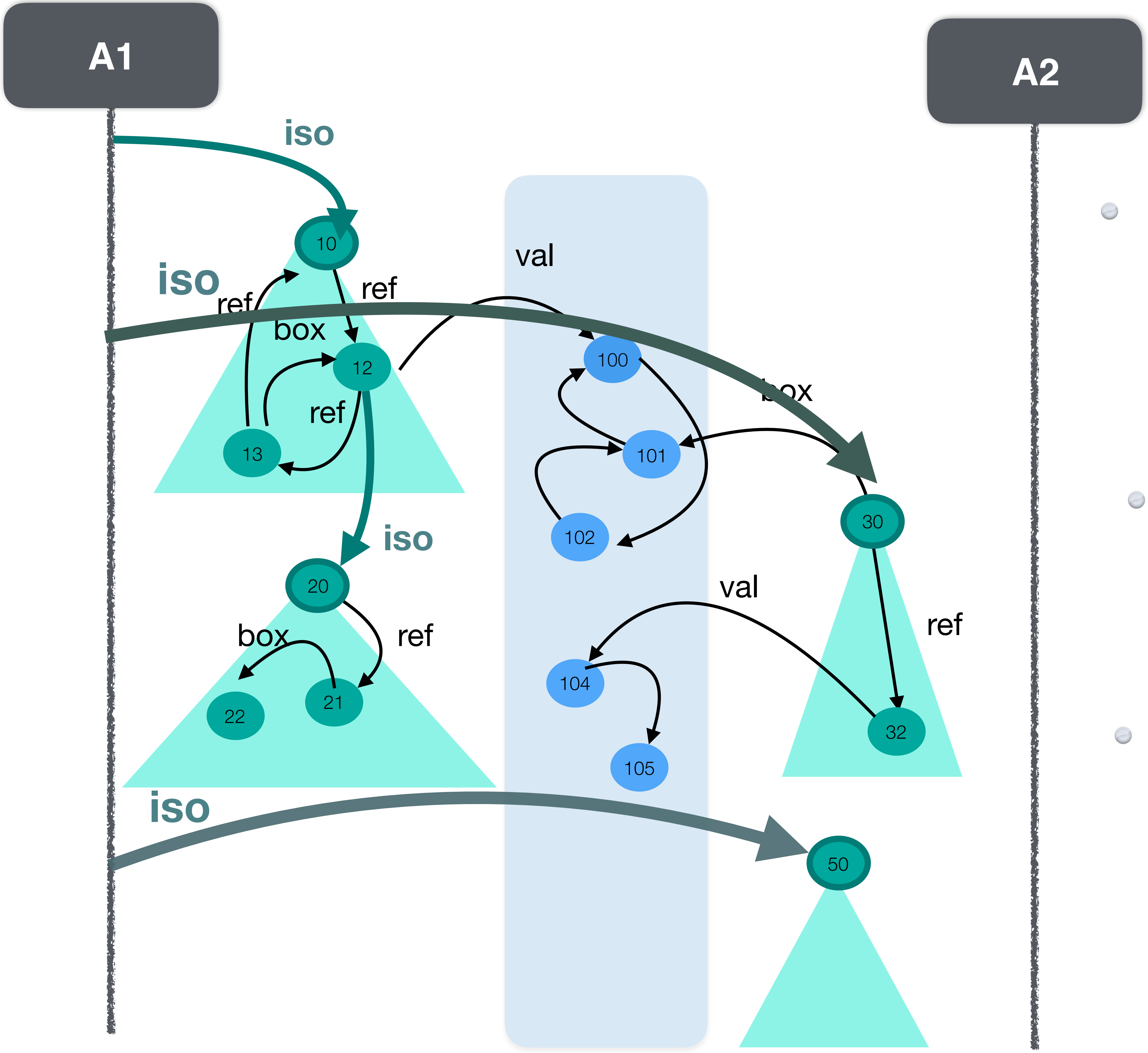


- at most one actor at a time has access to mutable region

- **val**-reference may be sent to other actor

- **iso**:-reference may be given up, and sent to other actor

Capabilities and Safe Communication

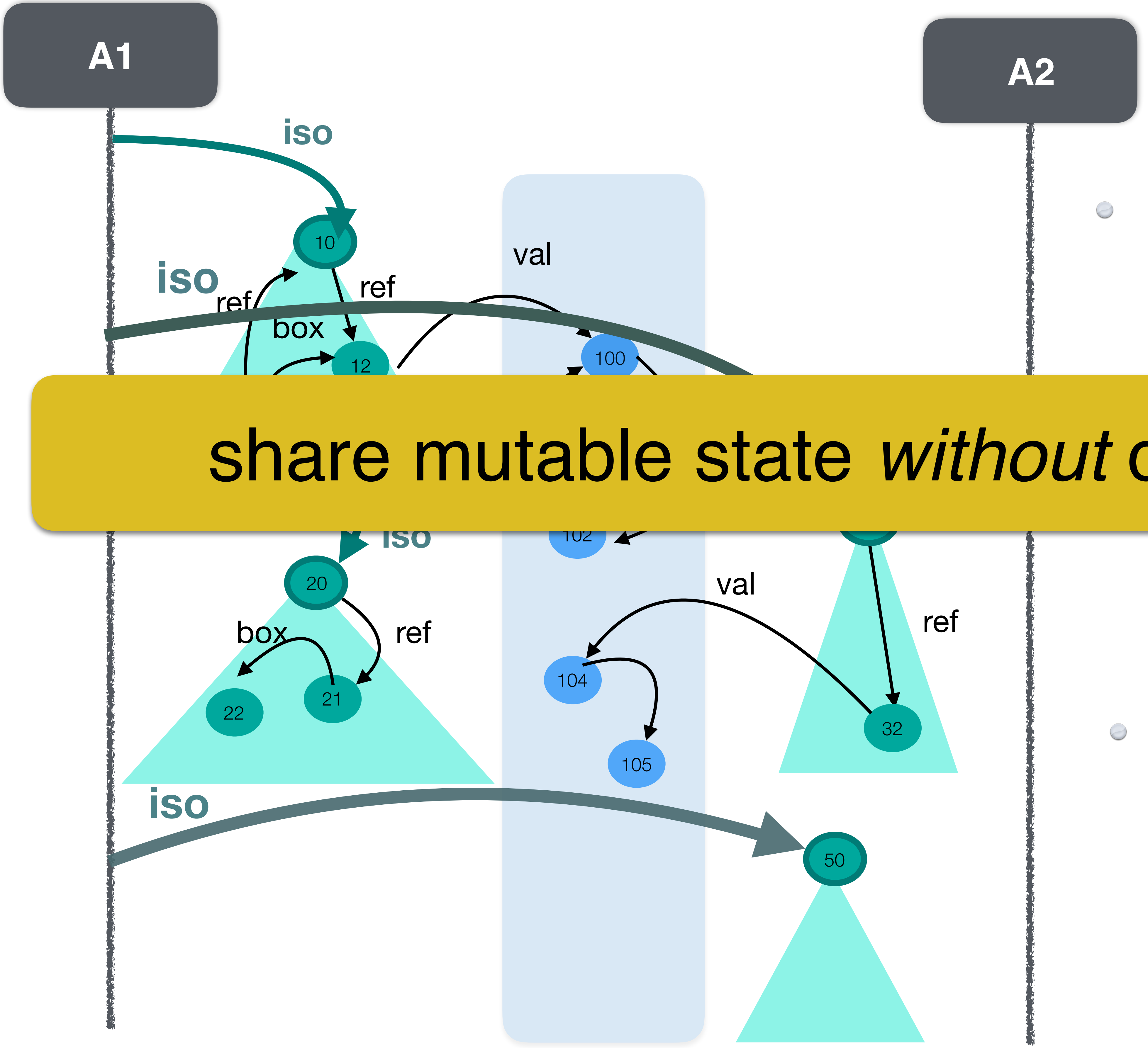


- at most one actor at a time has access to mutable region

- **val**-reference may be sent to other actor

- **iso**:-reference may be given up, and sent to other actor

Capabilities and Safe Communication



share mutable state *without* copying

- at most one actor at a time has access to mutable region

be sent to other actor

- **iso**:-reference may be given up, and sent to other actor

Guarantees of the type system

Guarantees

Guarantees

- **No data-races** At most one actor at a time has access to a mutable region.

Guarantees

- **No data-races** At most one actor at a time has access to a mutable region.
- **Immutability is deep and permanent** Everything that is in the immutable region remains immutable from now on

Guarantees - 2

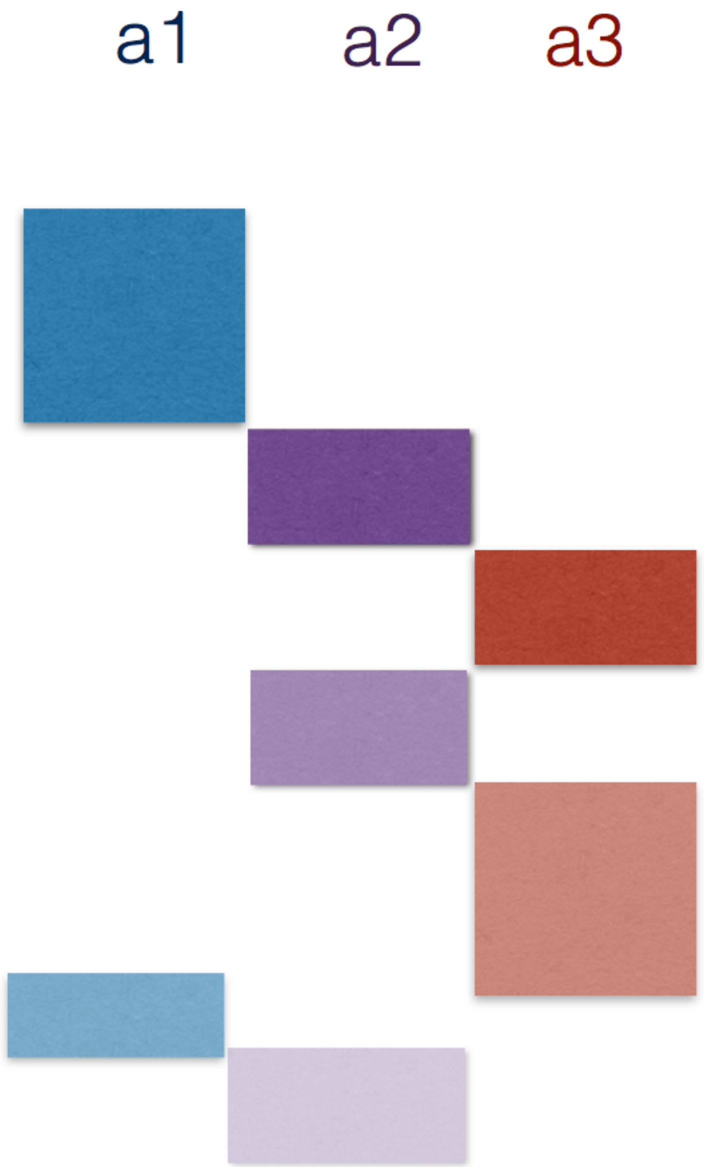
- **Capabilities weaker with distance** If path p.f has capability κ, then p has same or “stronger” capability.

- **Figment of atomicity** If configuration *C’* arises from *C* without message receipts at actor α, and if α sees o at non-tag capability at *C’*, and the contents of o at *C* different from contents at *C’*, then either α created o, or α caused the change.

-

the figment of atomicity

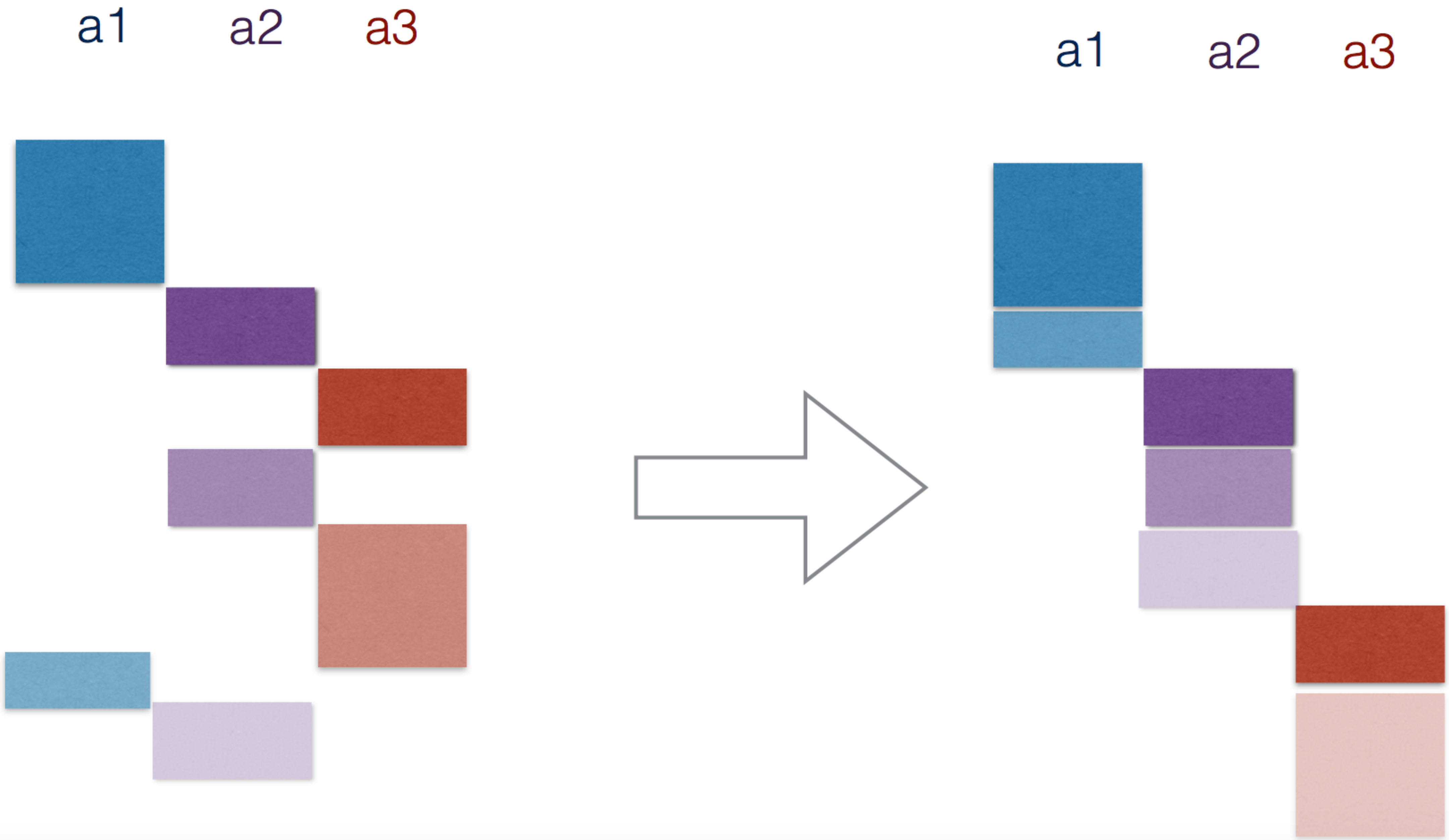
interleaved semantics



the figment of atomicity

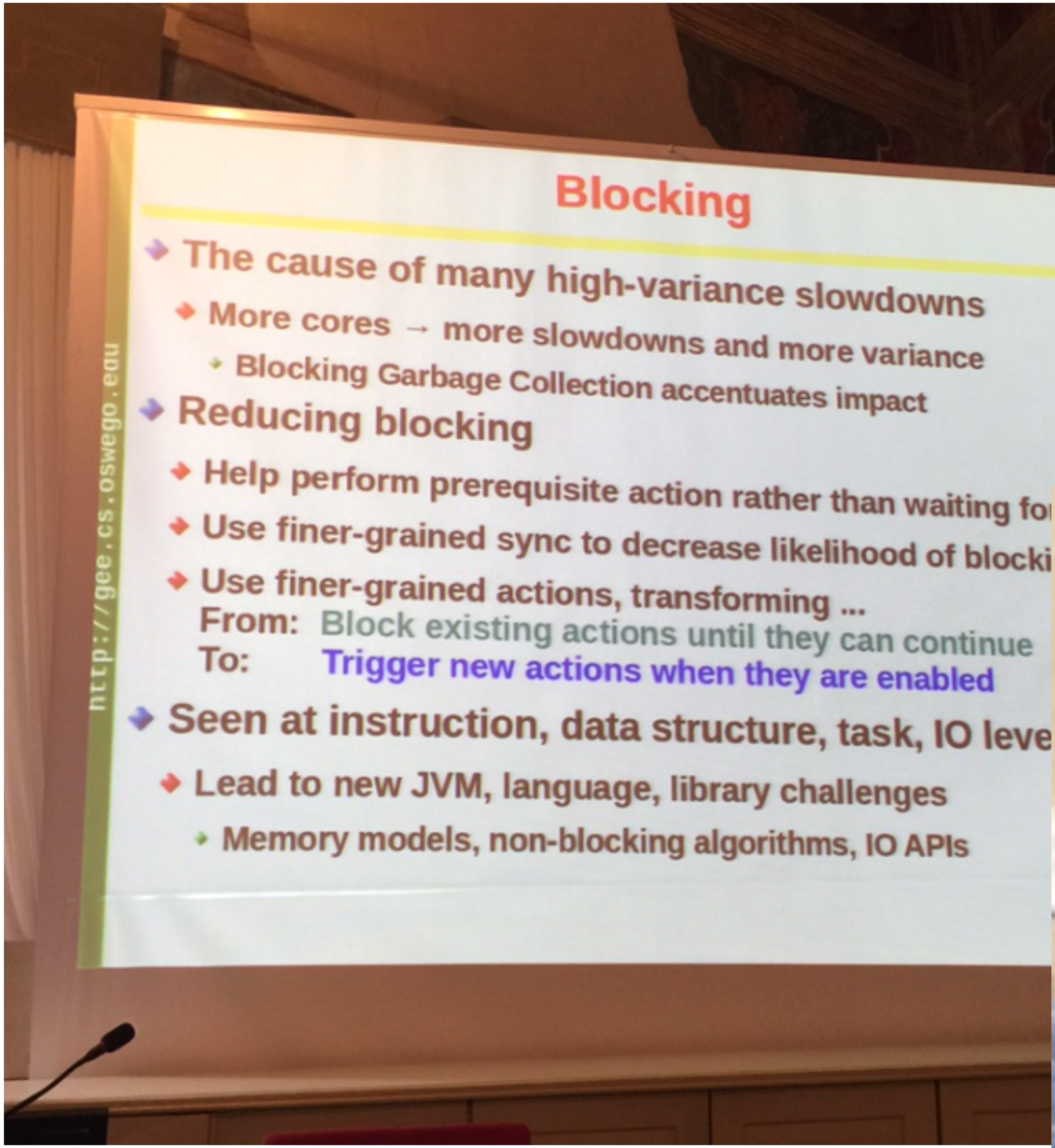
interleaved semantics

how programmers think



Today's Talk

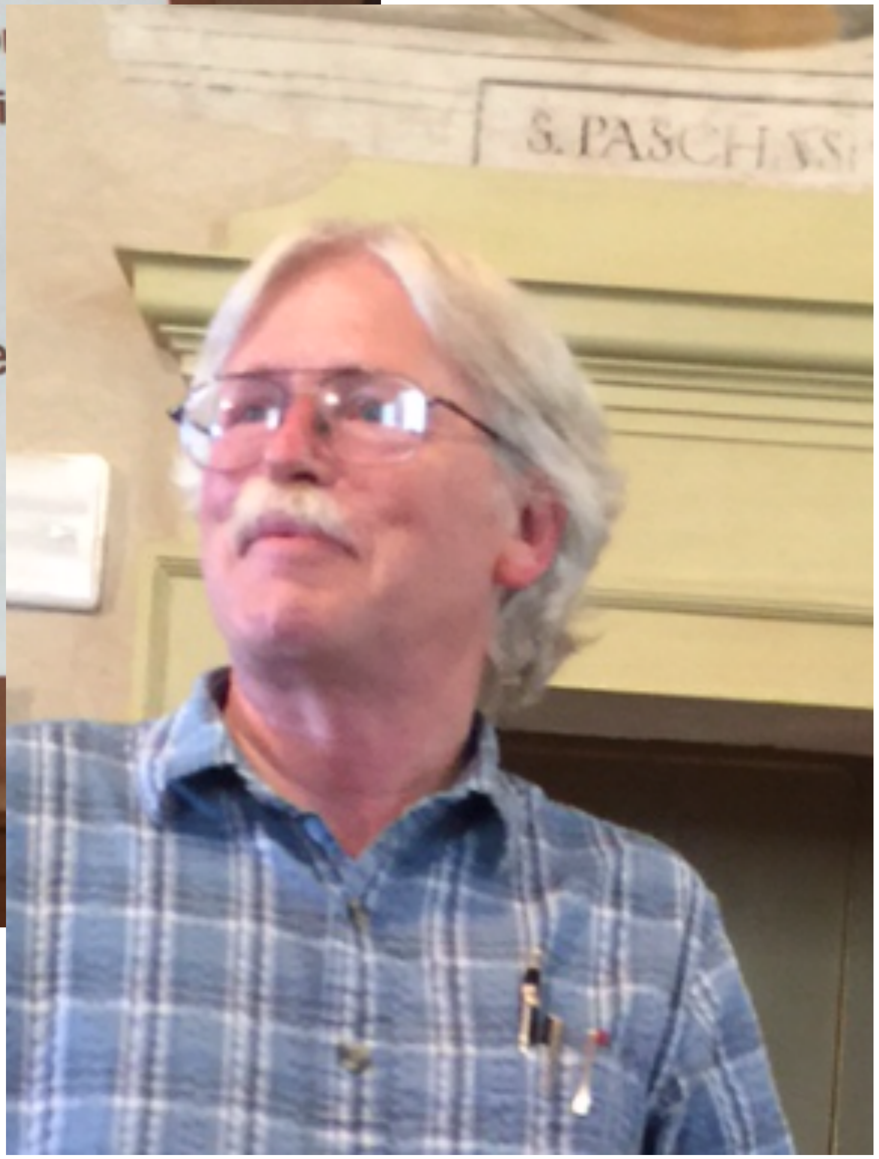
- Pony - the language and its design
  - Actors
  - Causality
  - The Type System
  - **Garbage Collection**



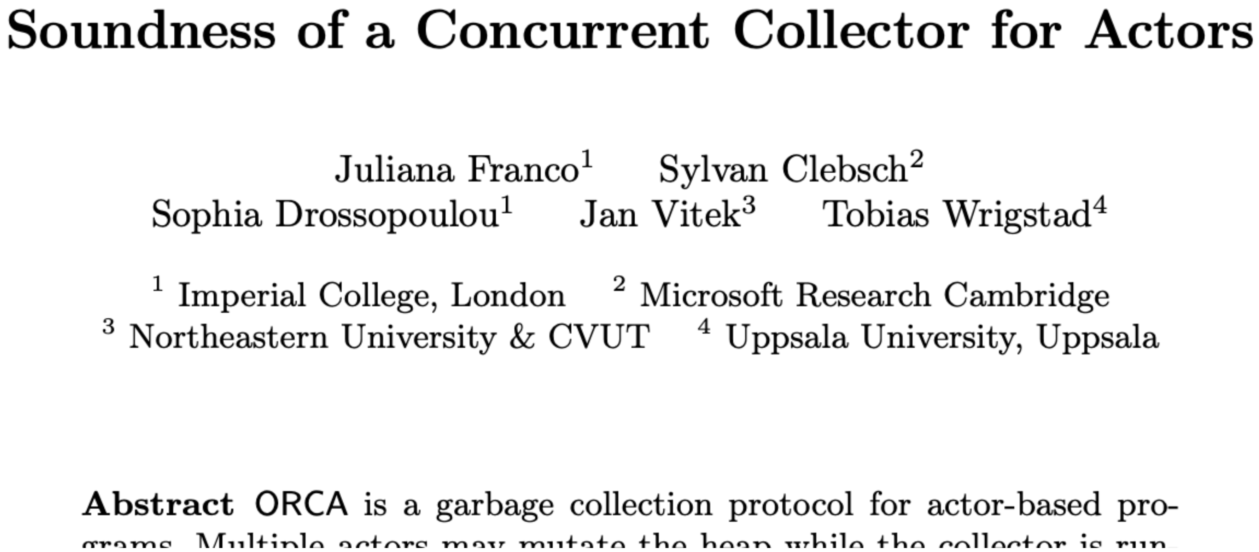
Blocking

- The cause of many high-variance slowdowns
  - More cores → more slowdowns and more variance
    - Blocking Garbage Collection accentuates impact
- Reducing blocking
  - Help perform prerequisite action rather than waiting fo
  - Use finer-grained sync to decrease likelihood of blocki
  - Use finer-grained actions, transforming ...
    From: Block existing actions until they can continue
    To: Trigger new actions when they are enabled
- Seen at instruction, data structure, task, IO leve
  - Lead to new JVM, language, library challenges
    - Memory models, non-blocking algorithms, IO APIs

http://gee.cs.oswego.edu



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



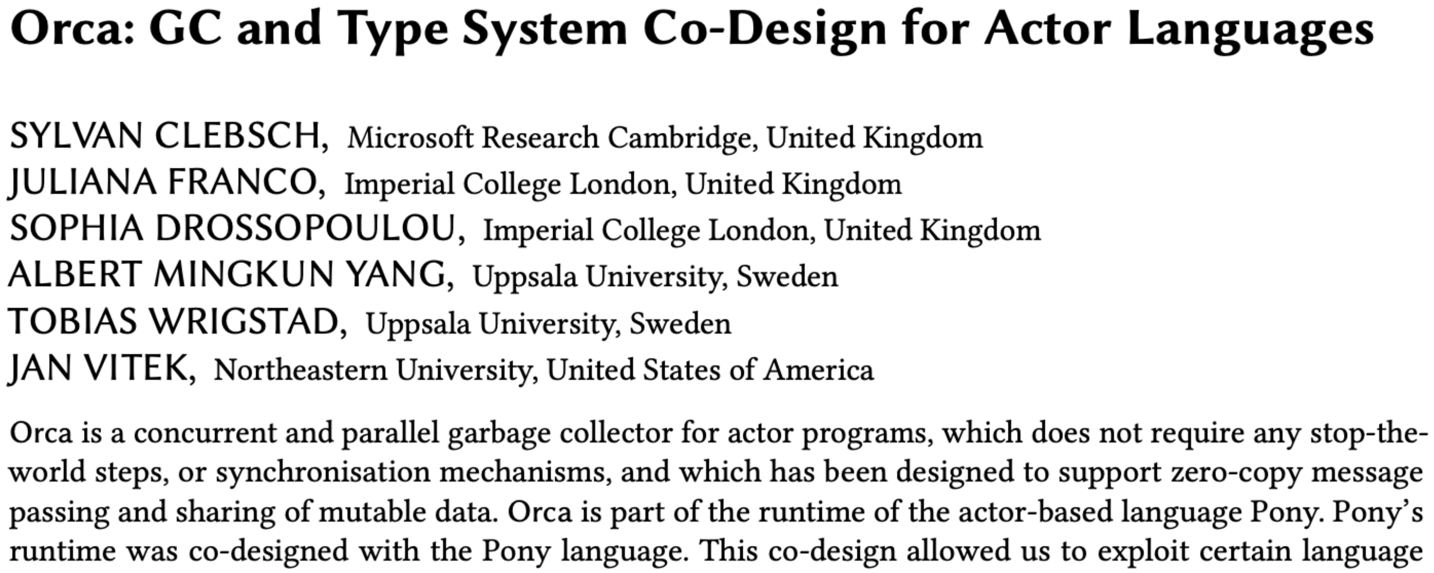
Soundness of a Concurrent Collector for Actors

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 programs. Multiple actors may mutate the heap while the collector is run

ESOP’18

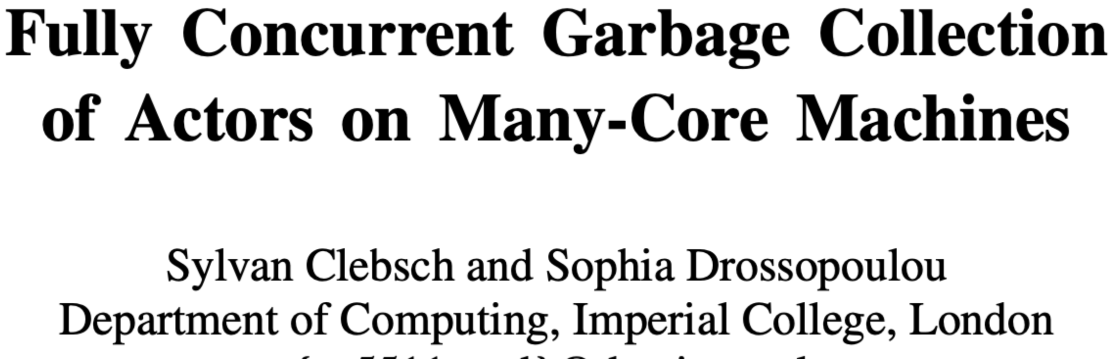


Orca: GC and Type System Co-Design for Actor Languages

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



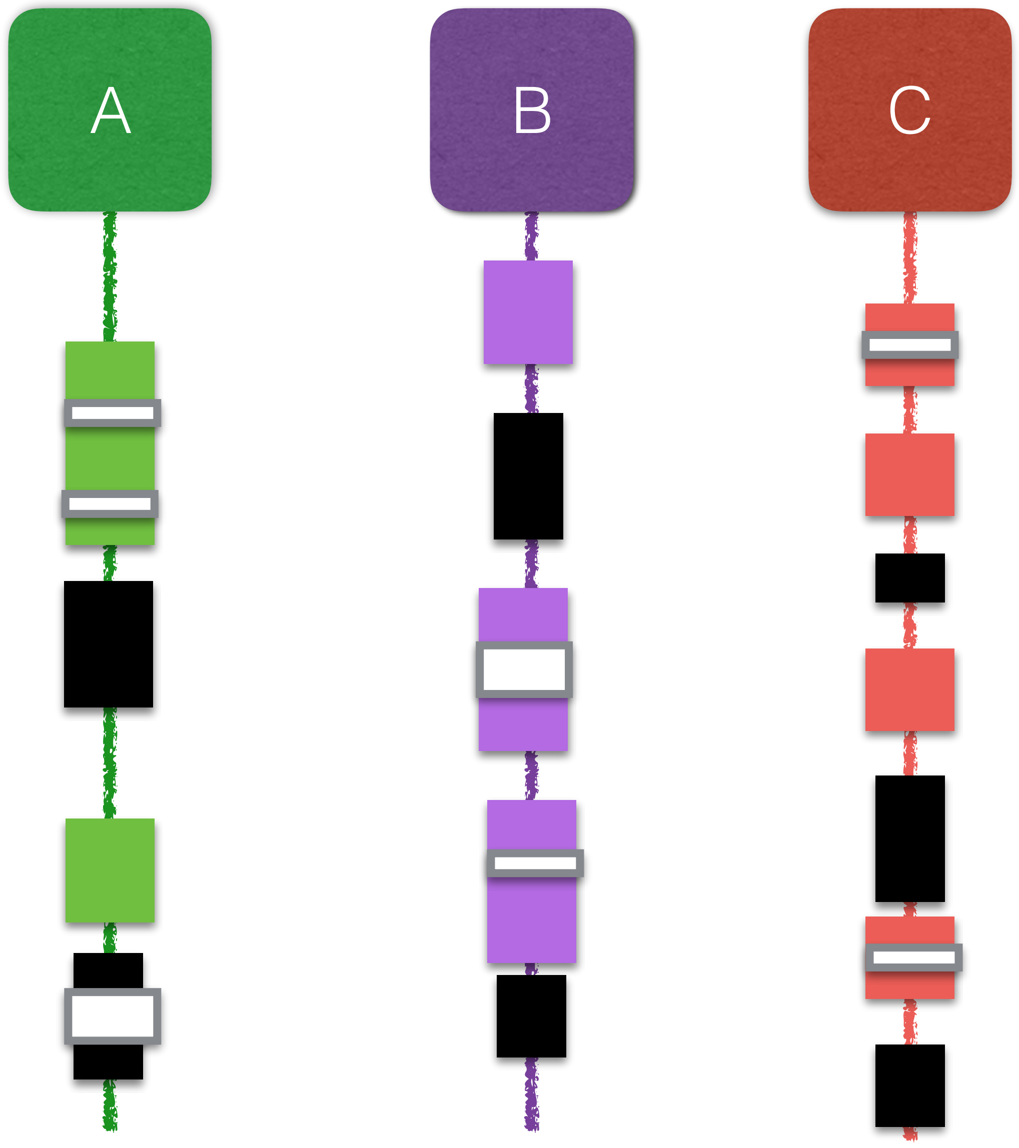
Fully Concurrent Garbage Collection of Actors on Many-Core Machines

Sylvan Clebsch and Sophia Drossopoulou
Department of Computing, Imperial College, London

OOPSLA’13

Pony Garbage Collection
is *fully concurrent*

ie no synchronization, lo locks, no barrier, no stop the world step.



A

B

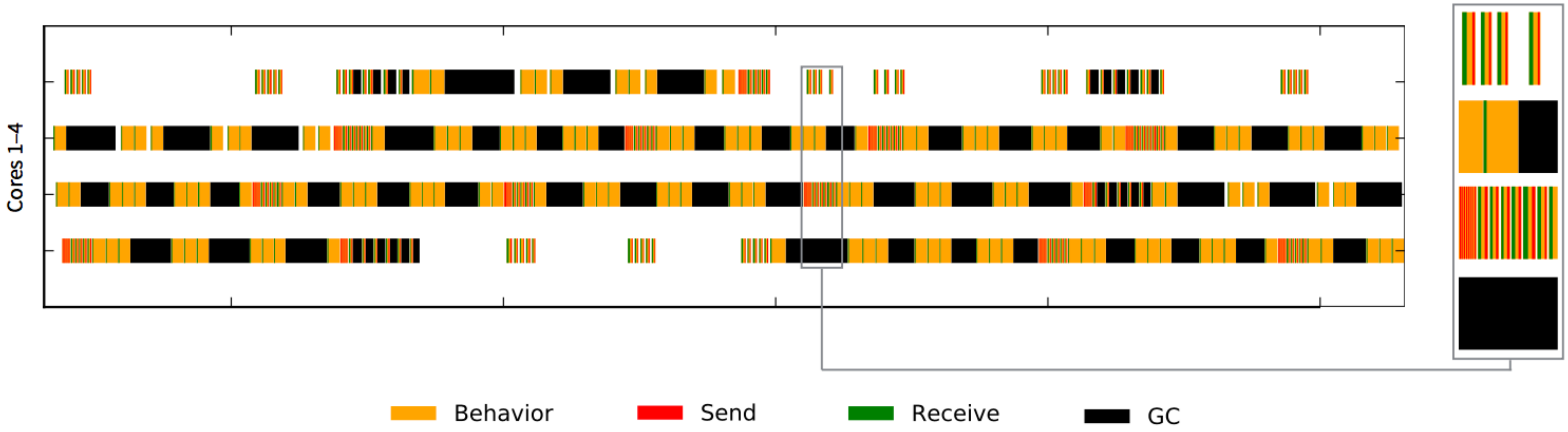
C

⬛ GC

🟩 🟪 🟥 behaviour

□ idle

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 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 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 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

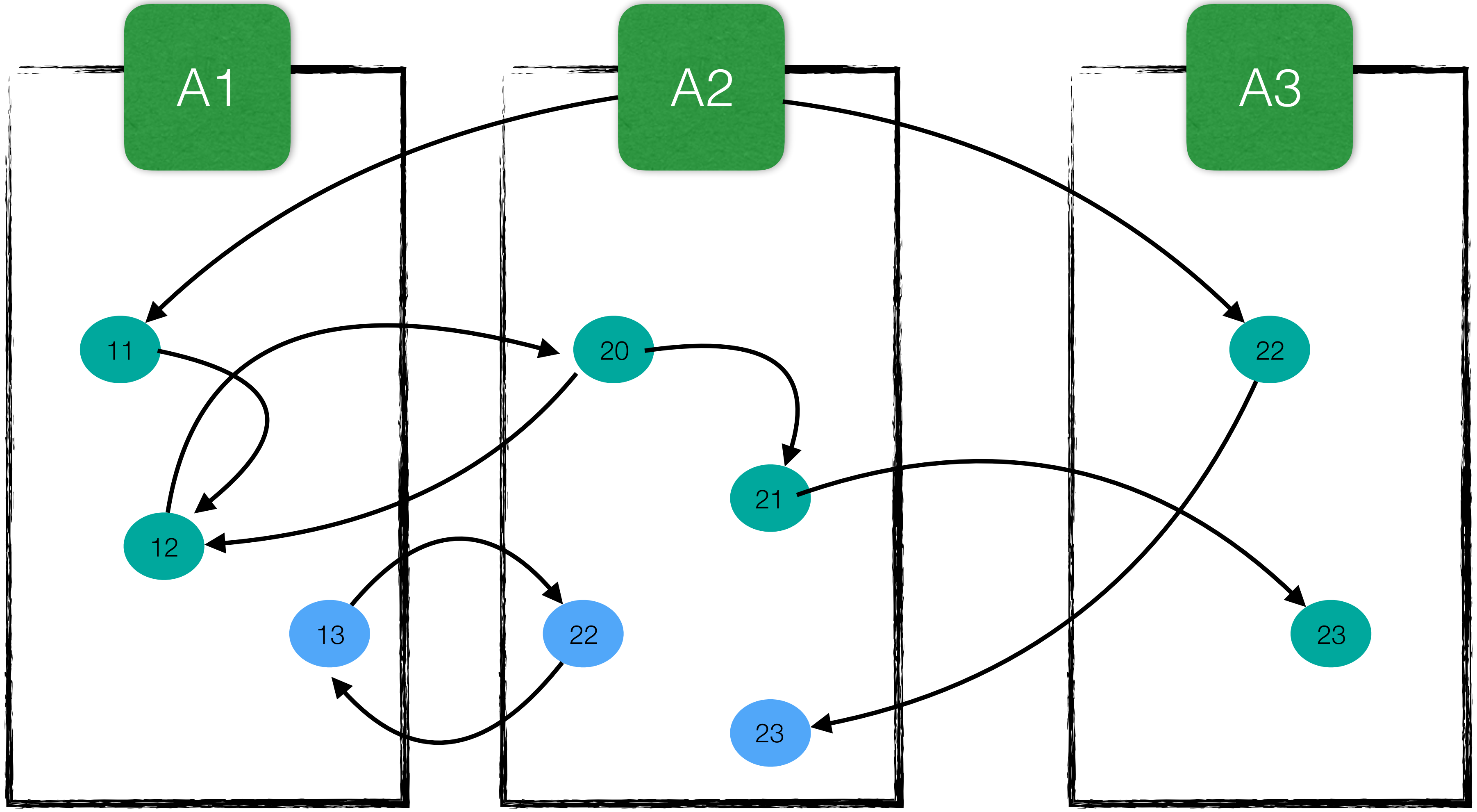
Challenge_3: How does the “owner” know whether there are foreign references to its owned objects?

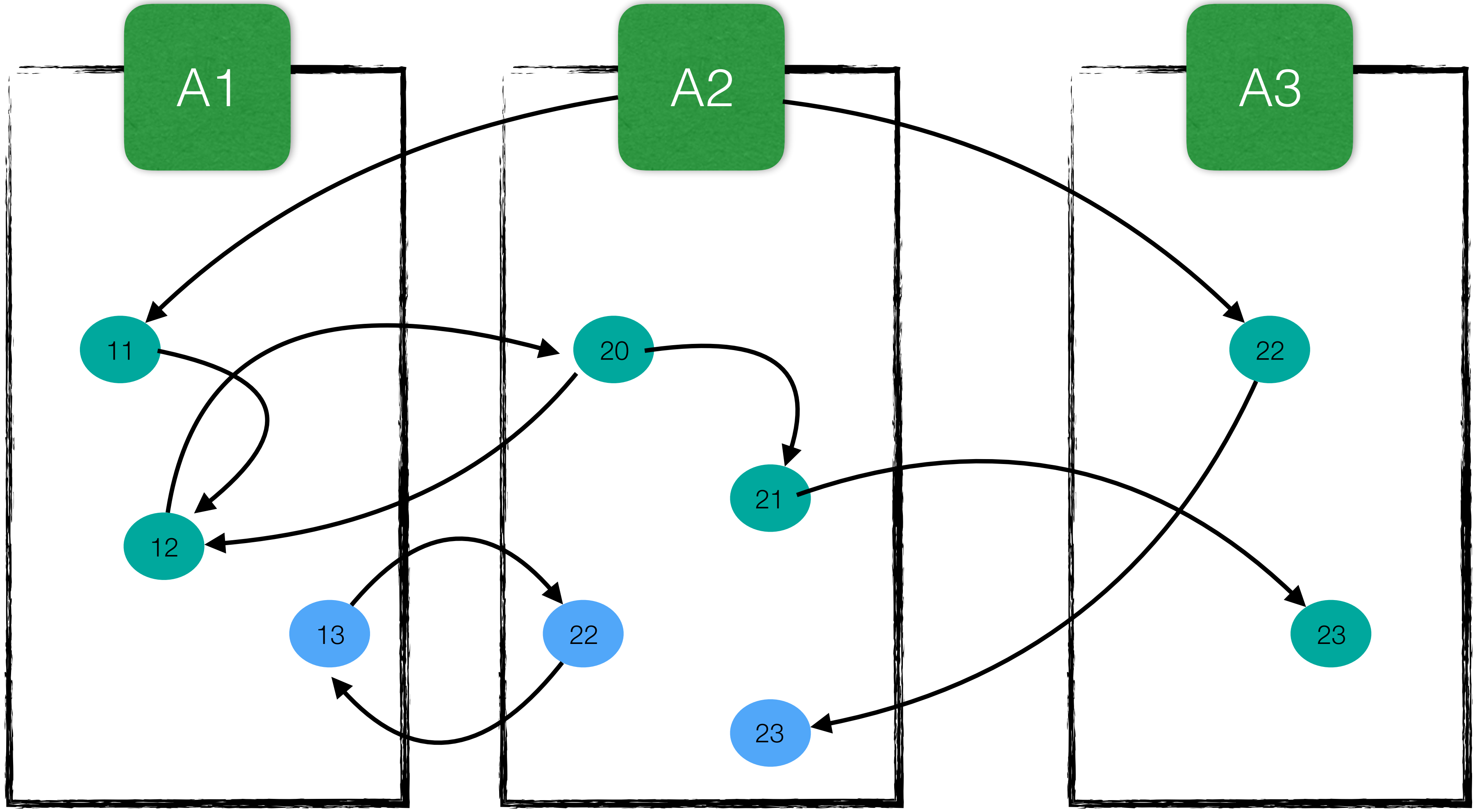
use Deferred Reference Counts and Messaging Mechanism

Challenge_4: How deal with uncertainty in message delivery

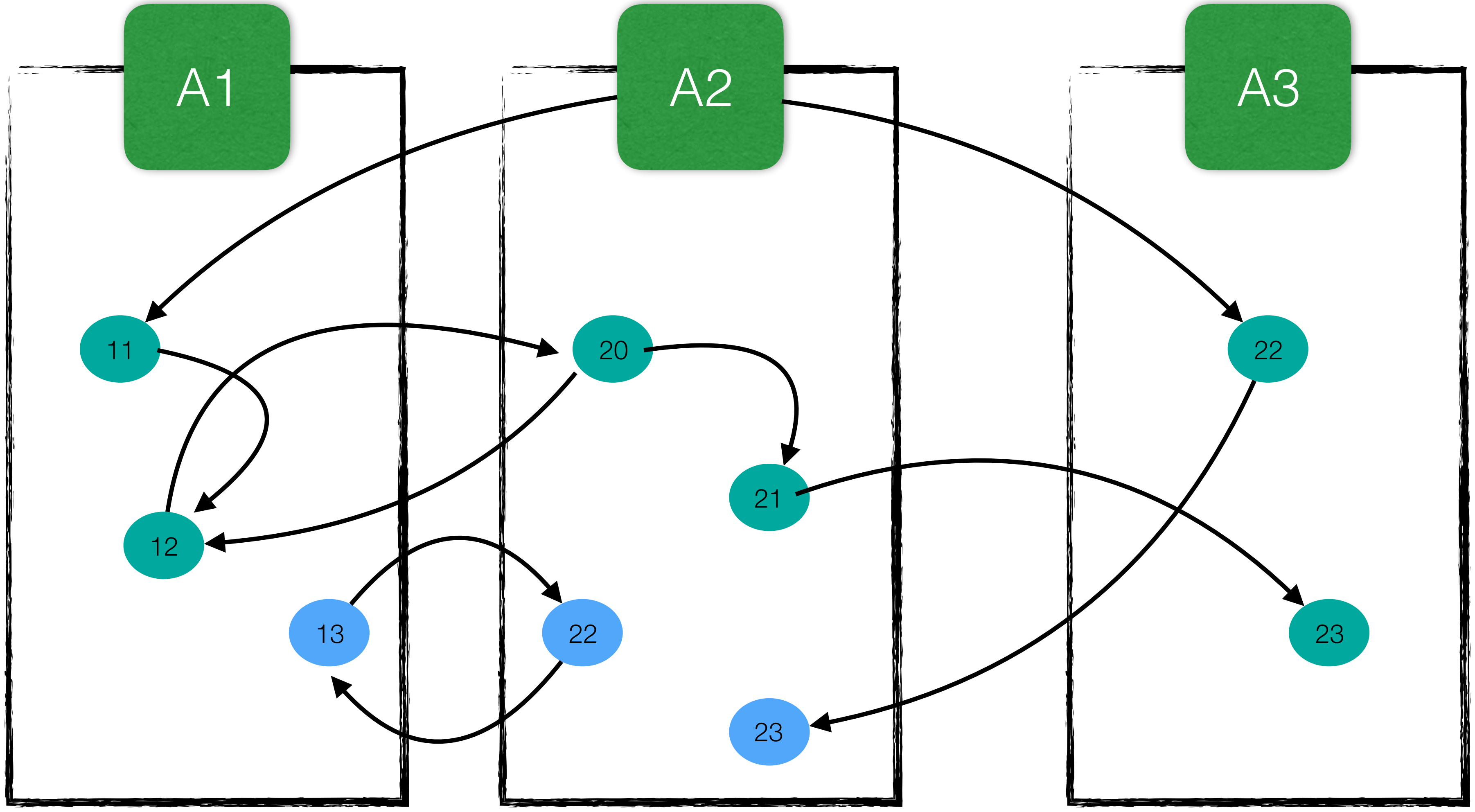
rely on Causal Message Delivery

Tight Connection between Language and Runtime Design


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

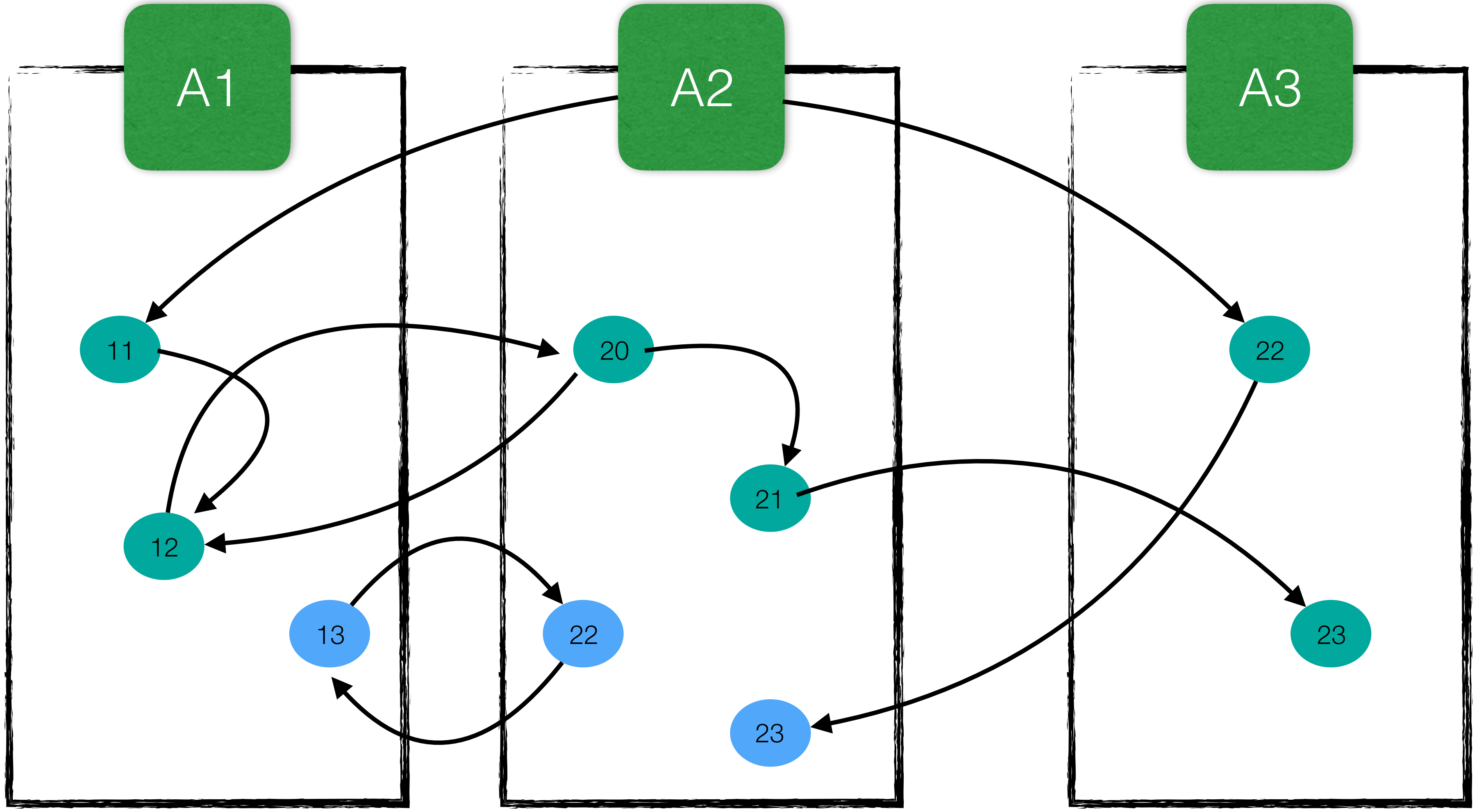


Challenges



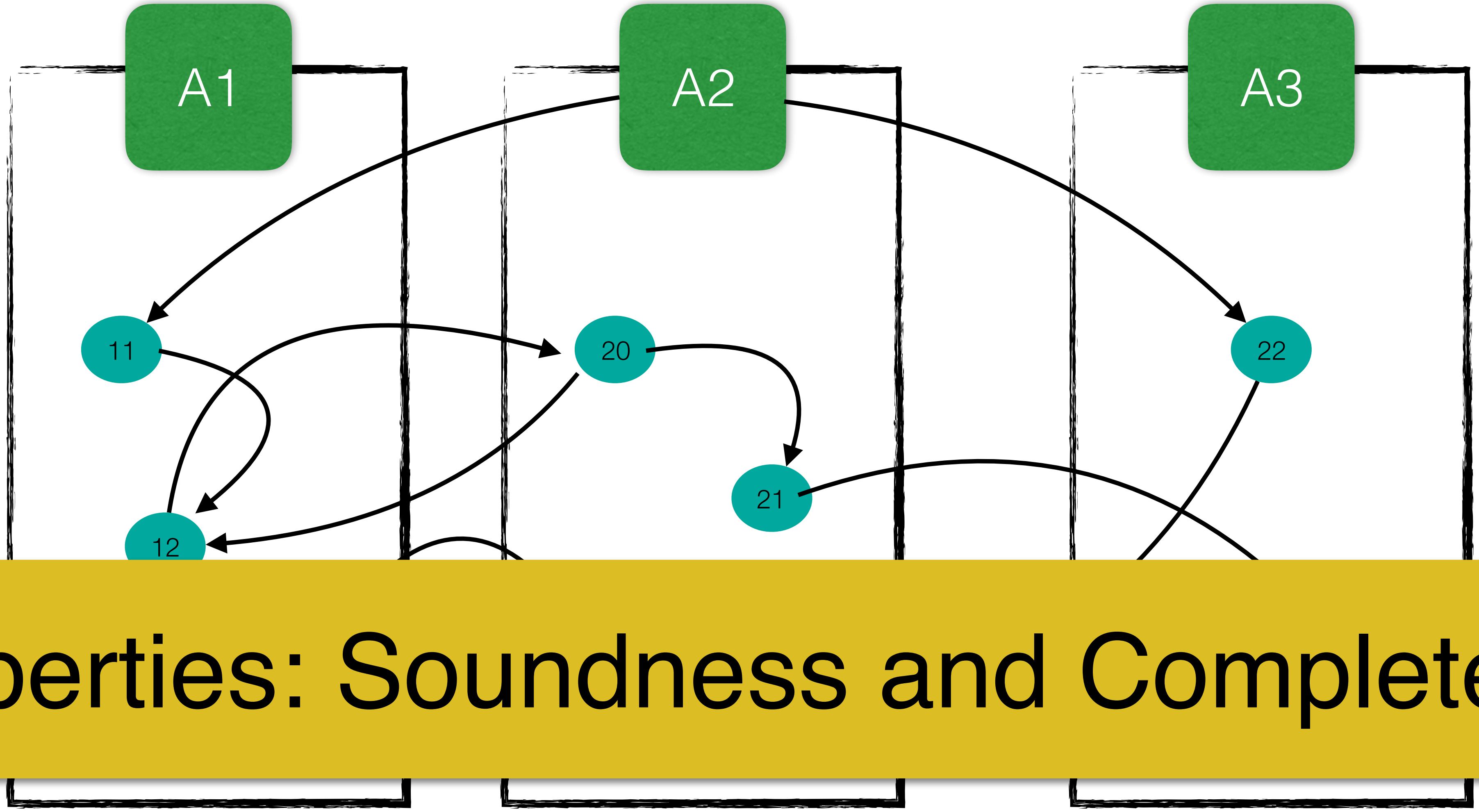
Challenges

- 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)

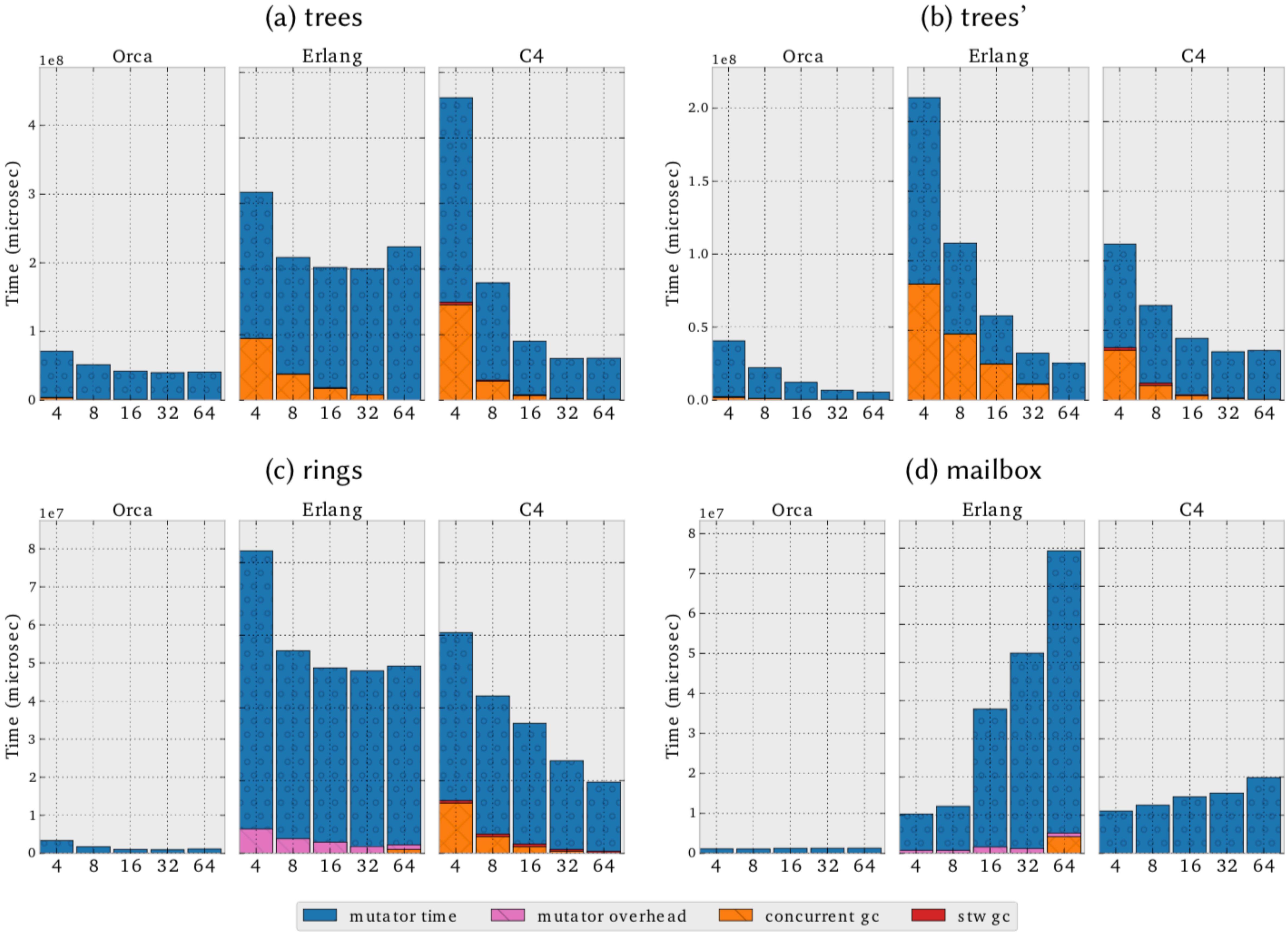


Properties: Soundness and Completeness

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



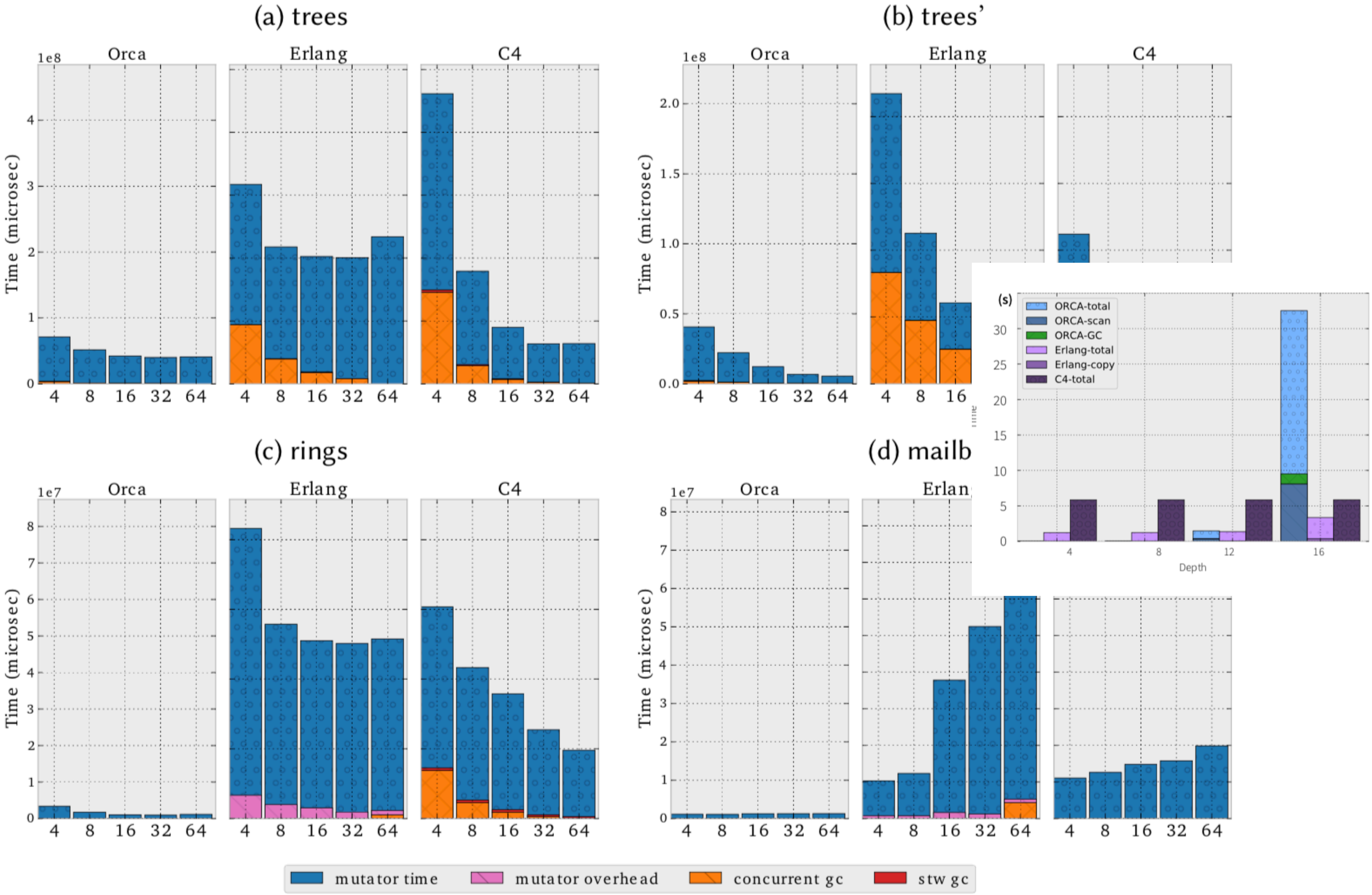
(a) trees

(b) trees’

(c) rings

(d) mailbox

Pony vs Erlang vs Java



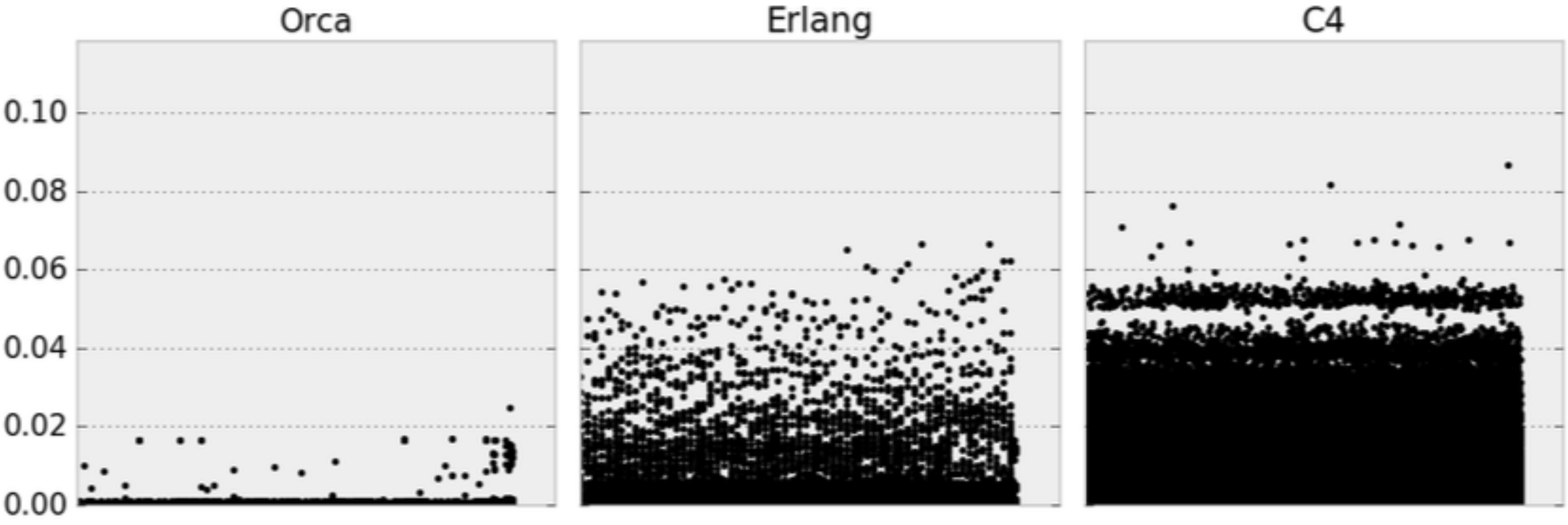
(a) trees

(b) trees'

(c) rings

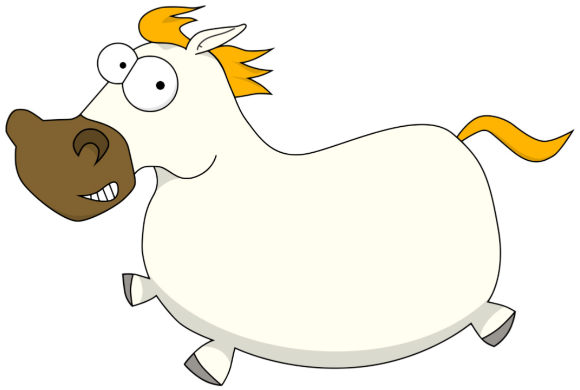
(d) mailb

Responsiveness



Pony features

- actors, objects
- pass mutable state without copying
- static types, type safe
- no Null values
- capabilities
- checked exceptions
- pattern marching
- lambda-s and partial applications
- causality
- traits and interfaces (nominal and structural types)
- union and intersection types
- generics ala f-bounded polymorphism
- consuming and destructive read
- alias/unalias and viewpoints in types
- C ffi
- small library



GC in the middle of behaviour
send
local
foreign
mutable
lrc++ trace
frc-- trace
immutable
lrc++ trace
frc-- trace
was mutable now immutable this.imm(o):=true
lrc++ trace
frc-- trace
?
IMM
recv
local
foreign
mutable
lrc-- trace
frc++ trace
immutable
lrc-- trace
frc++ trace
O: unshared → α traces O
passed → α does not trace O
shared → α traces O
OC o: traced by only one actor in mark phase
reachable
local
foreign
mutable
mark + trace
mark + trace
immutable
mark + trace
mark
trace
unreachable ∧ rc=0
local
foreign
mutable
collect
—
immutable
collect
—
unreachable ∧ rc>0
local
foreign
mutable
mark
DEC
immutable
mark + trace [as shared]
DEC
ORCA −
ORCA +
root+
roots={o|O(o)∈this ∧ o.rc>0} val
α1
α2
α3
mut
O1
O2
O3
o

Conclu
lrc++
trace
mutable
immutable
x1.iso
x2.iso