Concurrency in Verona

PLISS 2023

SOPHIA DROSSOPOULOU, IMPERIAL COLLEGE LONDON

Splash Cascais 2023

Sun 22 - Fri 27 October 2023 Cascais, Portugal

Attending ▾ Tracks ▾ Organization ▾ Search Series ▾

When Concurrency Matters

Behaviour-Oriented Concurrency

LUKE CHEESEMAN, Imperial College London
MATTHEW J. PARKINSON, Azure Research, Microsoft
SYLVAN CLEBSCH, Azure Research, Microsoft
MARIOS KOGIAS, Imperial College London
SOPHIA DROSSOUPOLOU, Imperial College London
DAVID CHISNALL, Azure Research, Microsoft
TOBIAS WRIGSTAD, Uppsala University
[illegible]ETAR, Microsoft

[illegible]ordination is central for modern concurrent programming. Many m
[illegible] coordination. However, the design decisions for these two
[illegible]ardependence of these two mechanisms should b
[illegible]We are not the first to realise this: the p
[illegible]h fine-grained decomposition
[illegible]wever, actor model pro
[illegible]hallenging t

SPLASH 2023 (series) /

OOPSLA

Accepted Papers | Information for Reviewers | Call for Papers

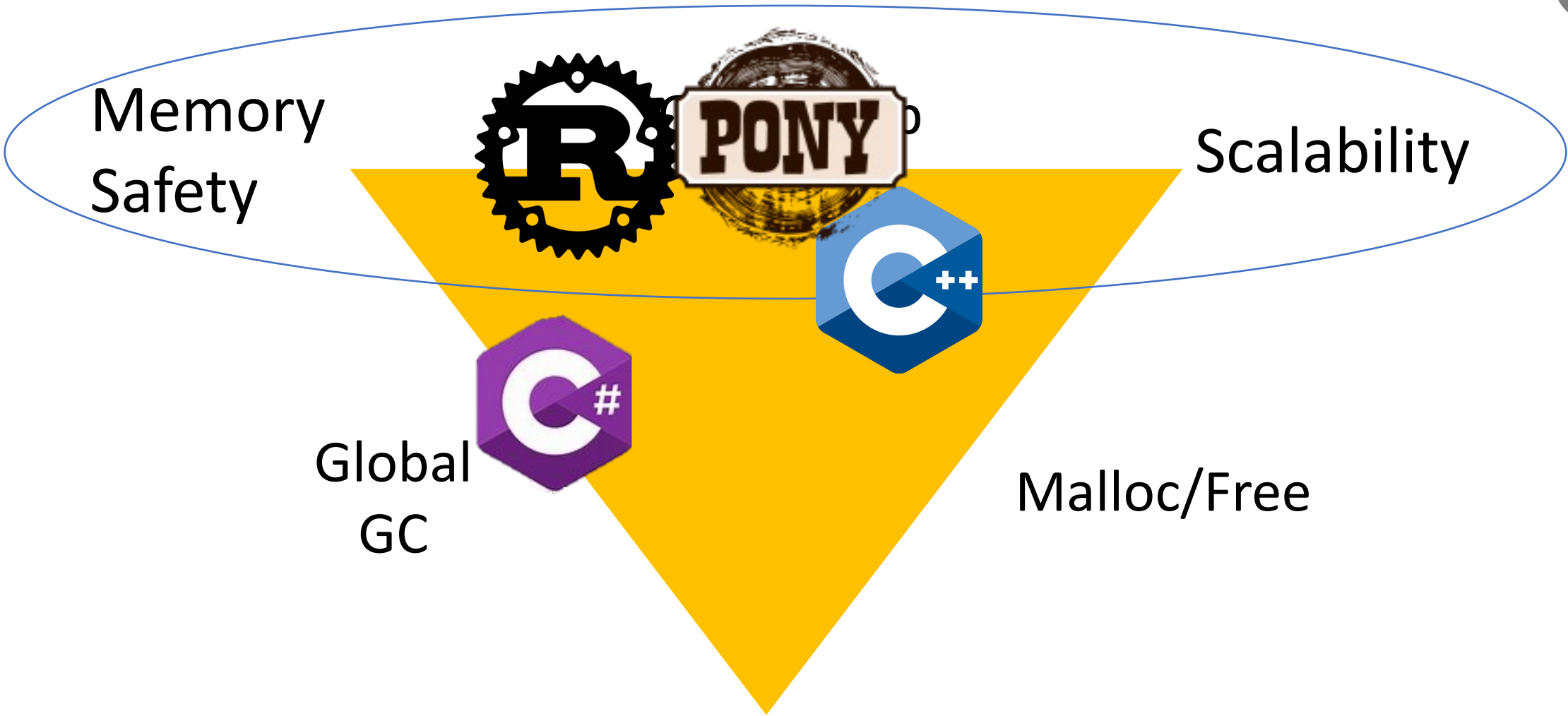







History: Project Snowflake

Manual Memory Management for .NET

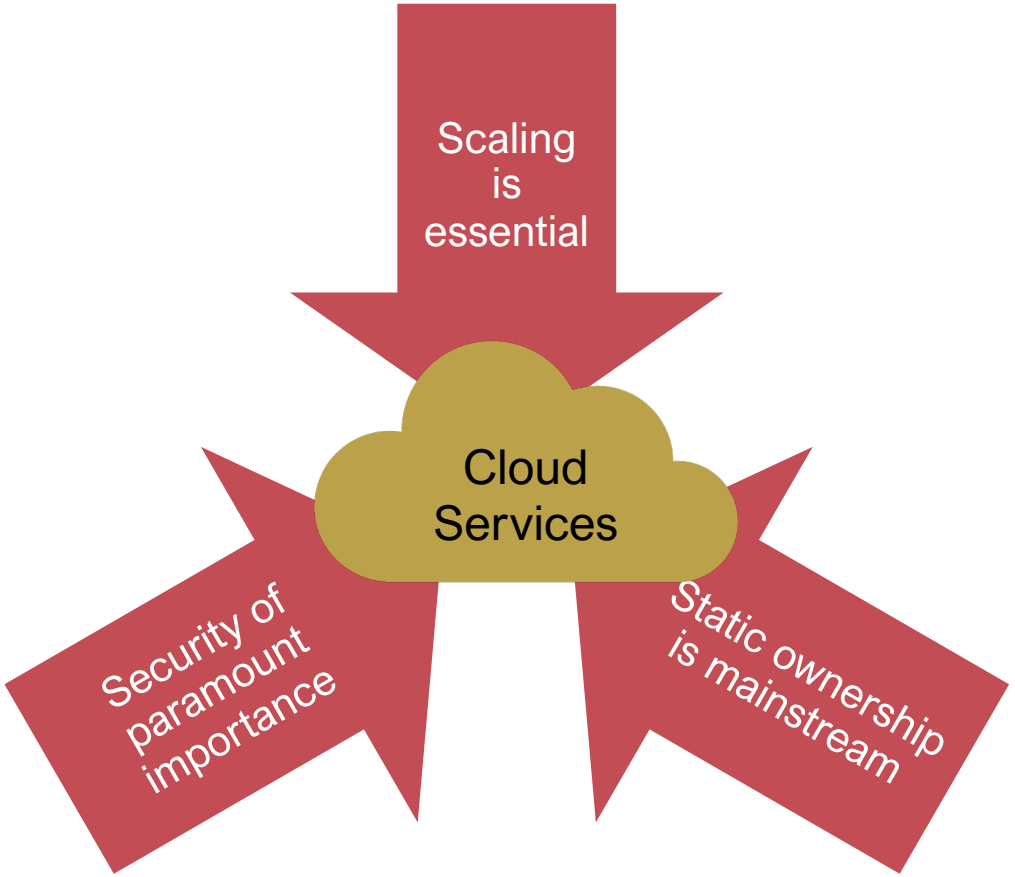

Memory
Safety
PONY
Scalability
Global
GC
Malloc/Free
Concurrent
Mutation

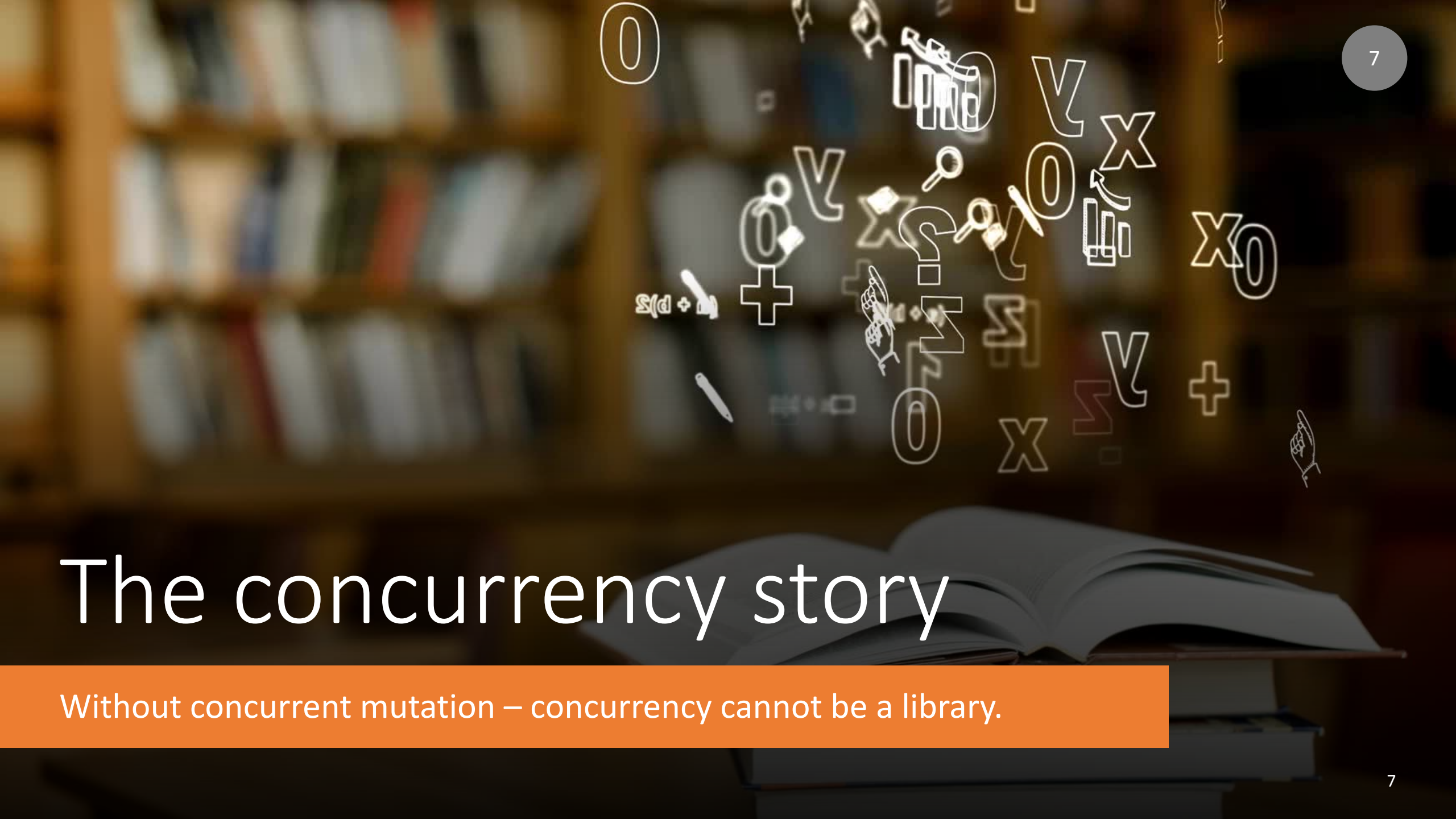
What is Project Verona?

Language design project

Key aim:

Provide safe, **local** memory management.



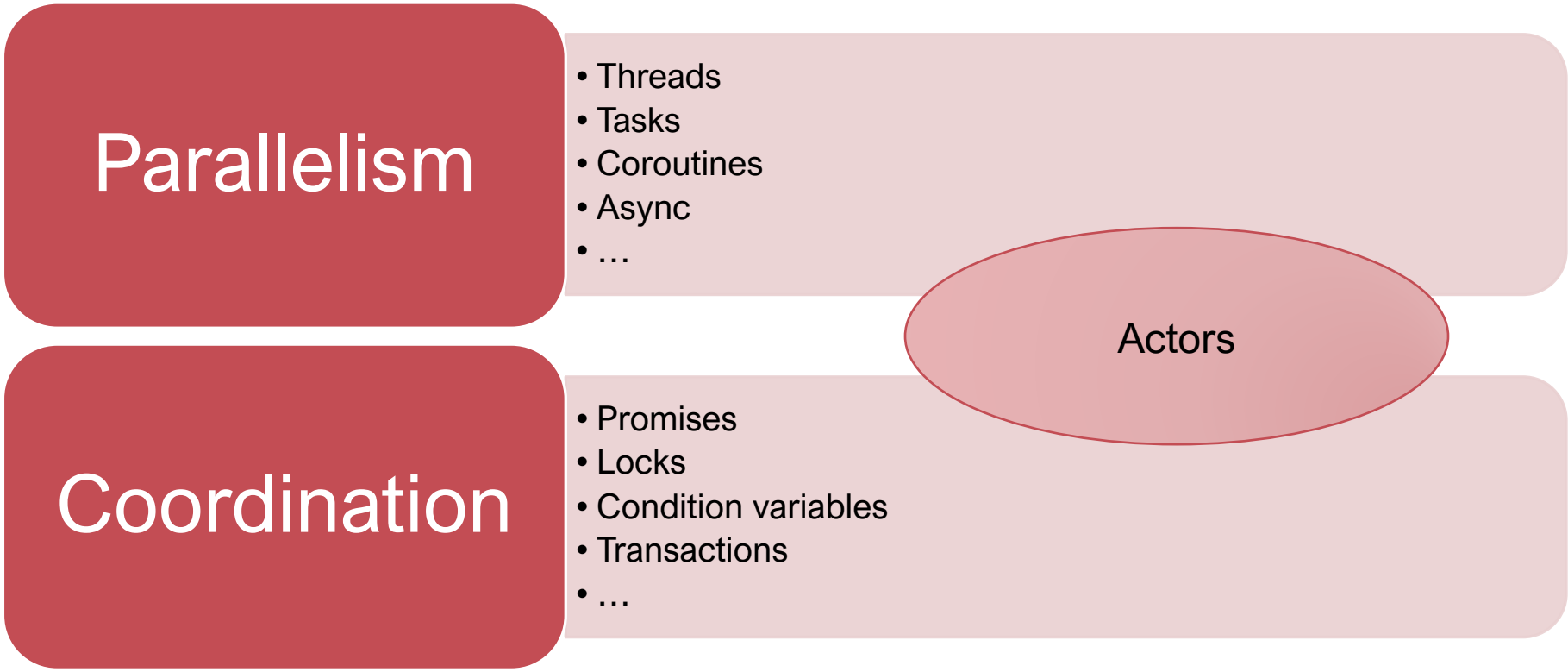


The concurrency story

Without concurrent mutation – concurrency cannot be a library.

Why revisit concurrency?

Without concurrent mutation – concurrency cannot be a library.



The origins
Why revisit concurrency?

- Matt: So how would you build a database with Actor Model Programming
- Sylvan: What do you mean?
- Matt: How do you represent a transaction over multiple tables with Actors?
- Sylvan: Yeah, that's hard!

- And thus Verona Concurrency was born

Today’s talk

- BOC in a nutshell
- BOC by example
- Semantics
- Implementation
- Evaluation

Behaviour-oriented concurrency – in a nutshell --

Cown - protects data

Behaviour - sole unit of execution; concurrent

Order - of behaviour execution

Behaviour-oriented concurrency – in a nutshell --

Cown – protects piece of separated data, ie provides unique external access to that data
- is *unavailable* (ie acquired by exactly one behaviour), or *available.*
- a behaviour may read/write only data protected by one of its acquired cowns.

Behaviour – the unit of (concurrent) execution.
- may be *spawned* with a list of required cowns.
- once spawned, a behaviour may *run* when all its required cowns are available
- while behaviour is running all its required cowns are un-available.
- when behaviour terminates, all it required cowns become available.
- cows are acquired/released atomically

Order – a behaviour may run only after all previously spawned behaviours with overlapping cowns have terminated.

Behaviour-oriented concurrency is PL parametric

Cown – protects piece of separated data, ie provides unique external access to that data

- is *unavailable* (ie acquired by exactly one behaviour), or *available.*
- a behaviour may read/write only data protected by one of its acquired cowns.

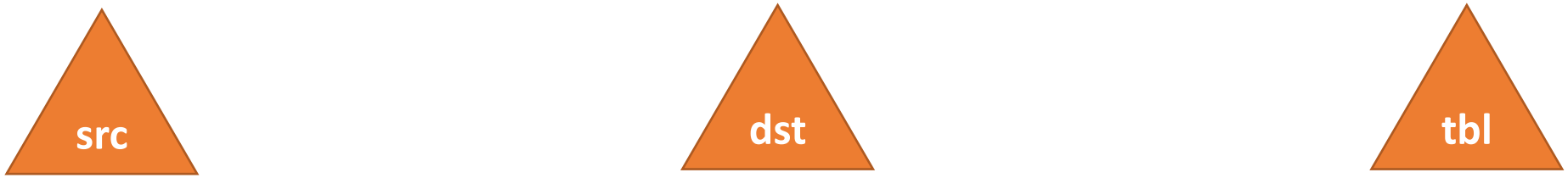
Behaviour – the unit of (concurrent) execution.

- may be *spawned* with a list of required cowns.
- once spawned, a behaviour may *run* when all its required cowns are available
- while behaviour is running all its required cowns are un-available.
- when behaviour terminates, all it required cowns become available.
- cows are acquired/released atomically

Order – a behaviour may run only after all previously spawned behaviours with overlapping cowns have terminated.

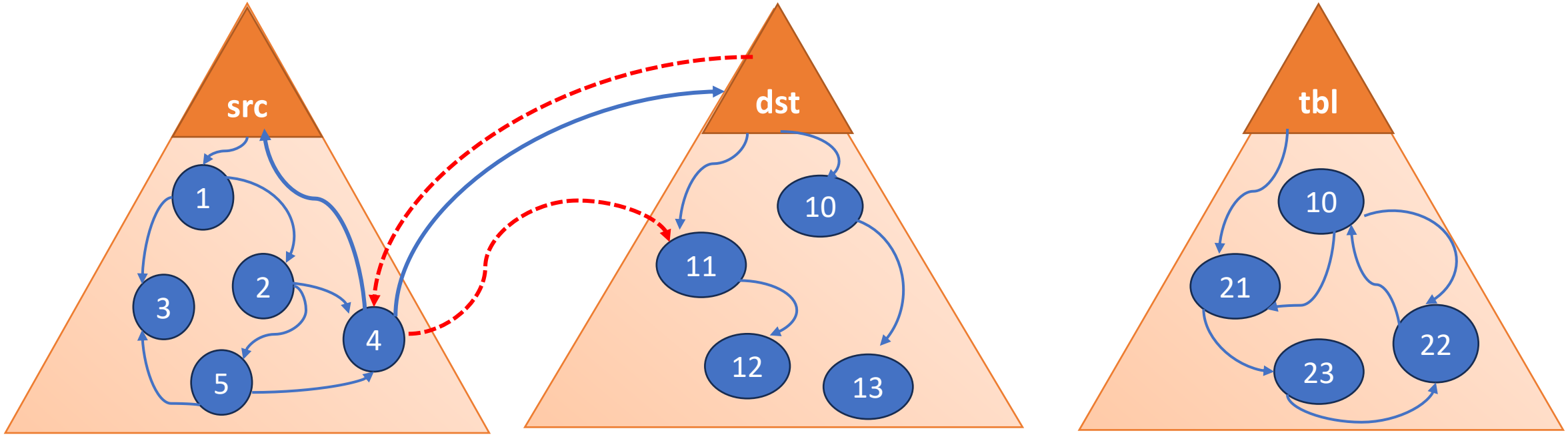
Cowns and protection

Cown – protects data, providing unique external read/write access to that data.



Cowns and protection

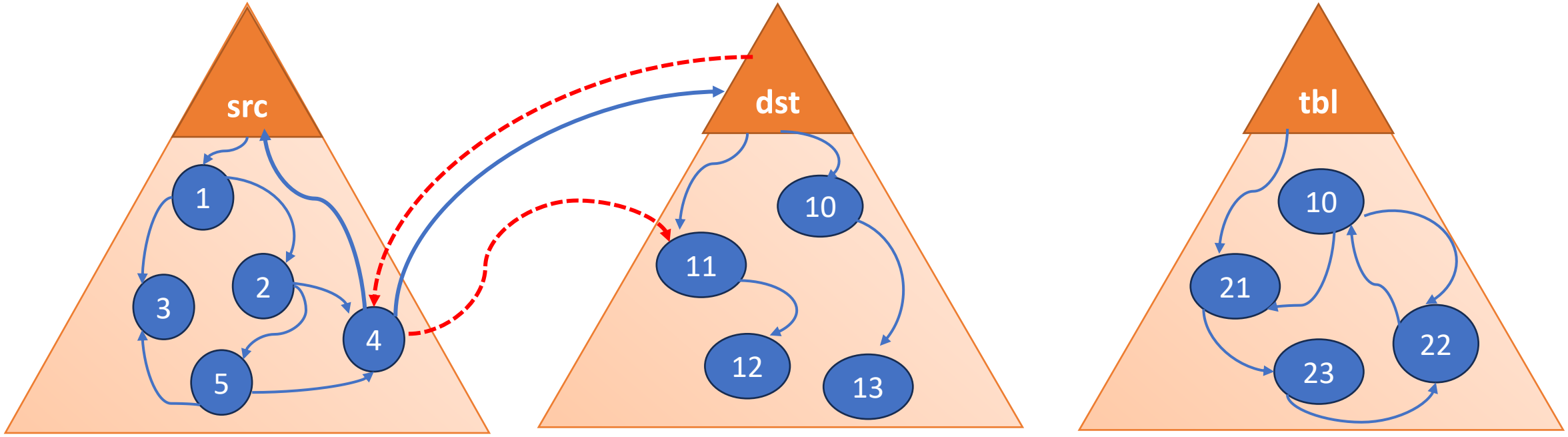
Cown – protects data, providing unique external read/write access to that data.



… the remit of the underlying PL's static/dynamic type system

Cowns and protection

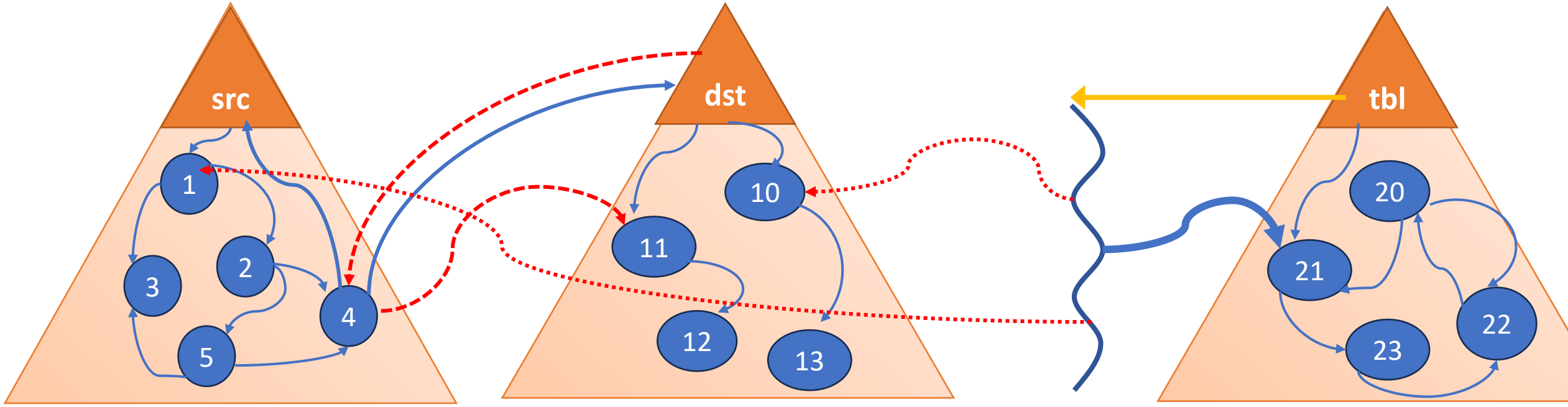
Cown – protects data, providing unique external read/write access to that data.



… the remit of the underlying PL's static/dynamic type system

Cowns and protection

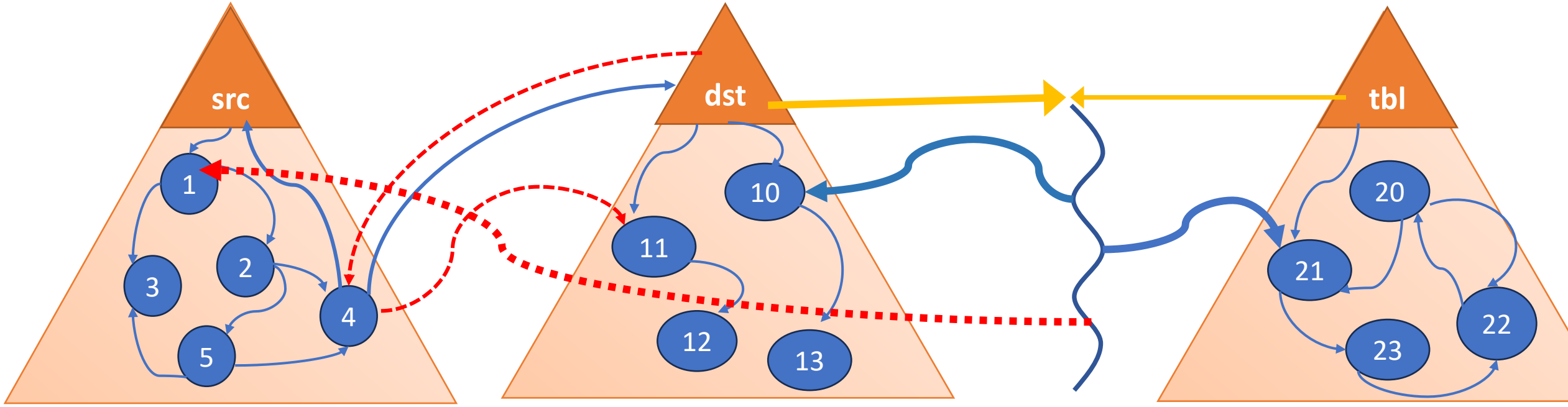
Cown – a behaviour may read/write only data protected by cowns it has acquired.



… a behaviour which has acquired tbl, may read/write 20, 21, … but *not* 10, 11, .. 1, 2, …

Cowns and protection

Cown – a behaviour may read/write only data protected by cowns it has acquired.



… a behaviour which has acquired tbl and dst may read/write 20, 21, … 10, 11 but *not* 1, 2, …

Cowns and protection

1. Cown protects piece of separated data, ie provides unique external access to that data.

2. Behaviour may only read/write data protected by its cowns.

3. Cown is *unavailable* (ie acquired by exactly one behaviour), or available (not acquired).

Therefore, Behaviour Oriented Concurrency is **free from data-races.**

1, 2 is the remit of the underlying PL's static/dynamic type system.

3 is the remit of the Behaviour Oriented Concurrency extension.

Behaviour execution

Behaviour – the unit of (concurrent) execution.

- may be *spawned* with a list of required cowns.
- once spawned, a behaviour may *run* when all its required cowns are available
- while behaviour is running all its required cowns are un-available.
- when behaviour terminates, all it required cowns become available.
- cows are acquired/released atomically

Order – a behaviour may run only after all previously spawned behaviours with overlapping cowns have terminated.

Therefore, Behaviour Oriented Concurrency is **free from deadlocks**.

Behaviour oriented programming by example

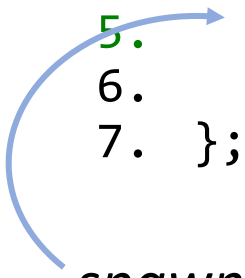
Creating Concurrent Owners (Cowns)

```
1.    var src = cown.create(Account.create(0));
2.    src.balance += 50; // illegal access
3.
4.    when (src) {
5.        src.balance -= amount;
6.  };
```

spawn behaviour

Creating Concurrent Owners (Cowns)

```
1.    var src = cown.create(Account.create(0));
2.    src.balance += 50; // illegal access
3.
4.    when (src /* src has type cown[Account] */ ) {
5.       // src has type Account
6.       src.balance -= amount;
7.  };
```



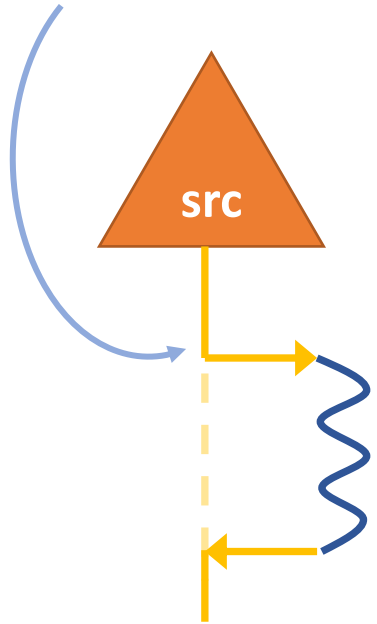
spawn behaviour

Creating Concurrent Owners (Cowns)

```
1.    var src = cown.create(Account.create(0));
2.    src.balance += 50; // illegal access
3.
4.    when (src /* src has type cown[Account] */ ) {
5.       // src has type Account
6.       src.balance -= amount;
7.  };
```

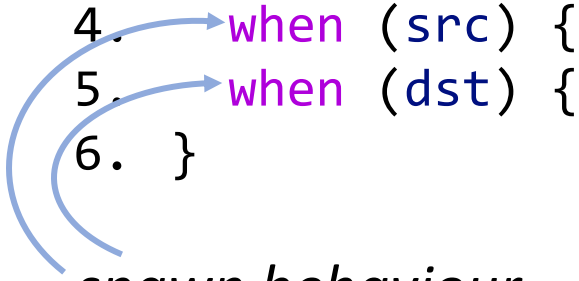
spawn behaviour

dispatch behaviour



Spawning and Dispatching behaviours

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.      when (src) { src.balance -= amount; };
5.      when (dst) { dst.balance += amount; };
6.  }
```

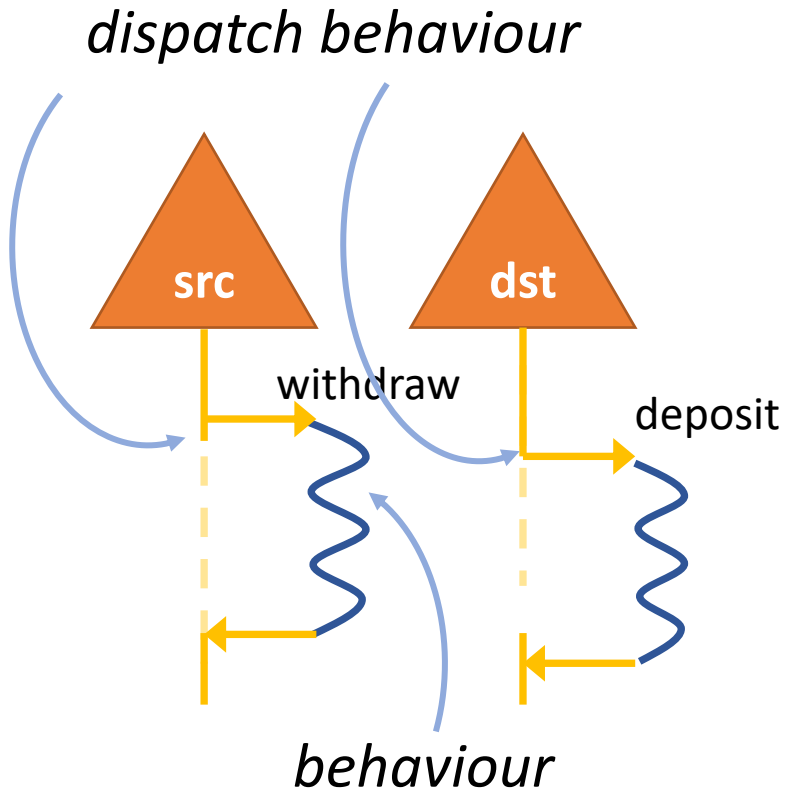


spawn behaviour

Spawning and Dispatching behaviours

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.   when (src) { src.balance -= amount; };
5.   when (dst) { dst.balance += amount; };
6.  }
```

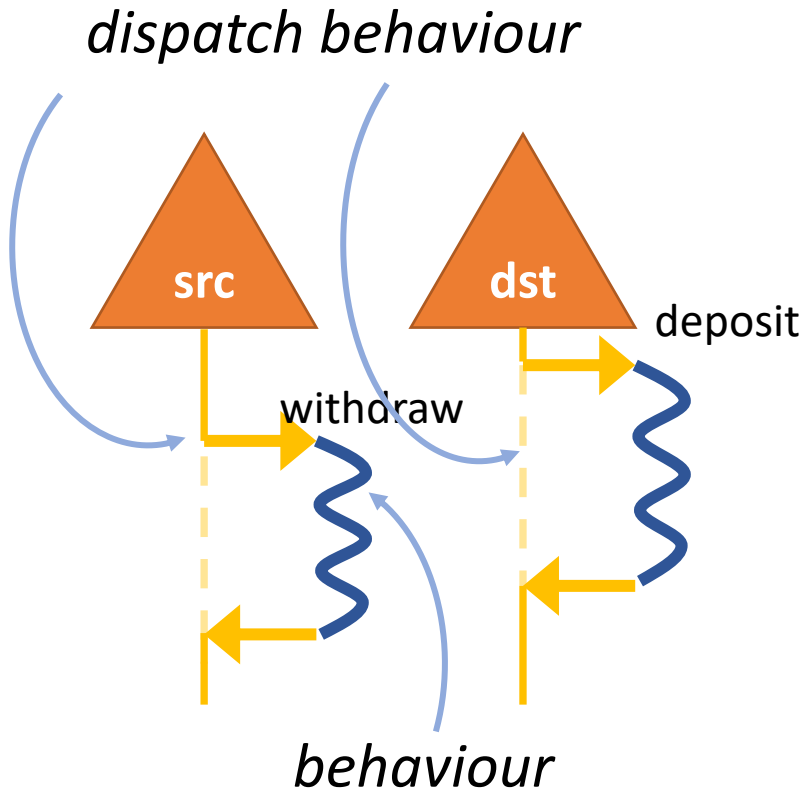
spawn behaviour



Spawning and Dispatching behaviours

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.   when (src) { src.balance -= amount; };
5.   when (dst) { dst.balance += amount; };
6.  }
```

spawn behaviour



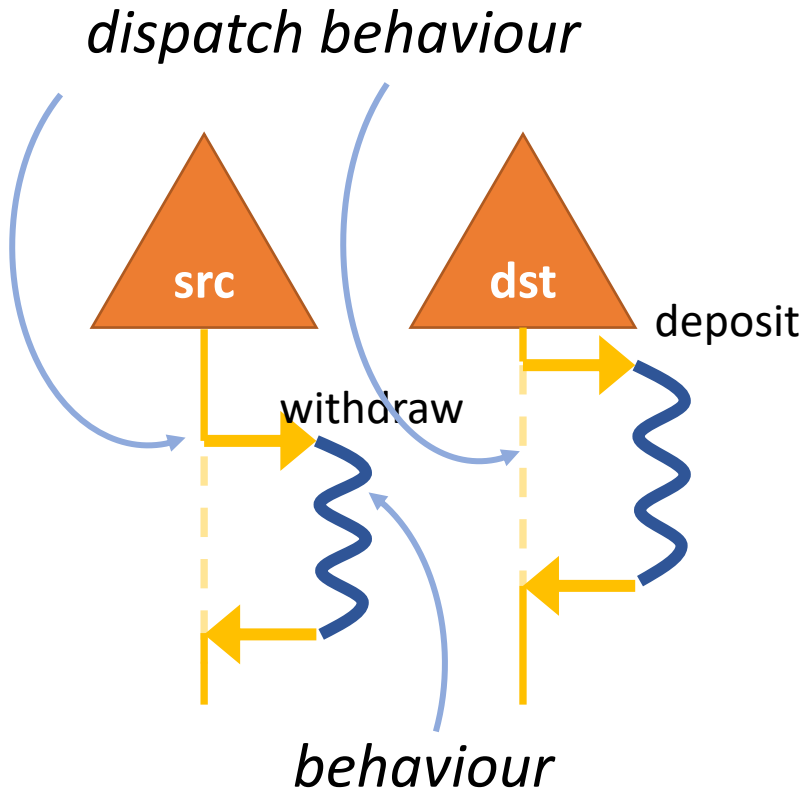
Is it possible that withdraw runs after deposit?

Order – a behaviour may run only after all previously spawned behaviours with overlapping cowns have terminated.

Spawning and Dispatching behaviours

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.      when (src) { src.balance -= amount; };
5.      when (dst) { dst.balance += amount; };
6.  }
```

spawn behaviour

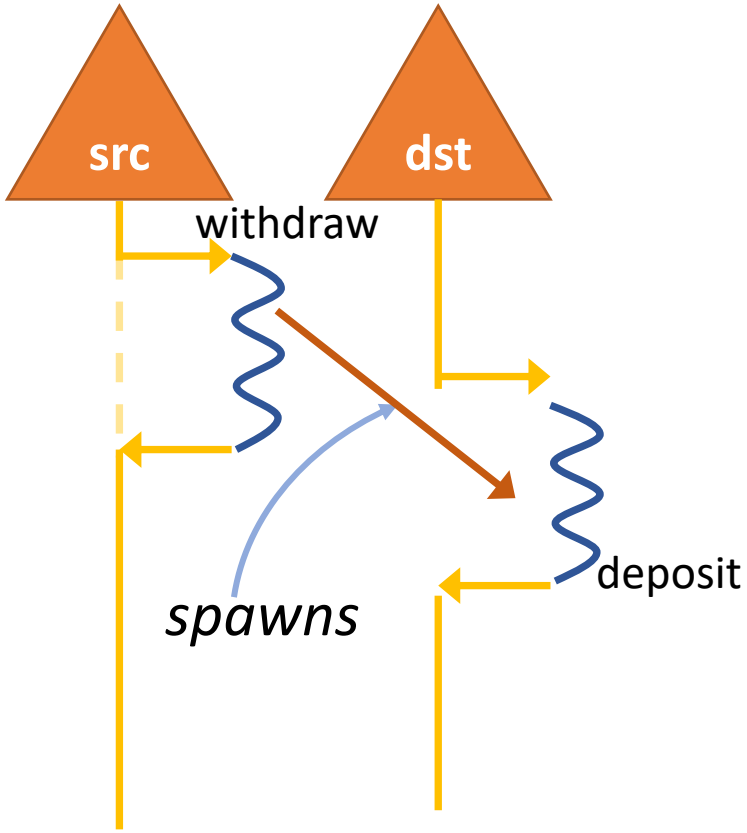


Is it possible that withdraw runs after deposit? **YES**

Spawning is synchronous, dispatching is asynchronous

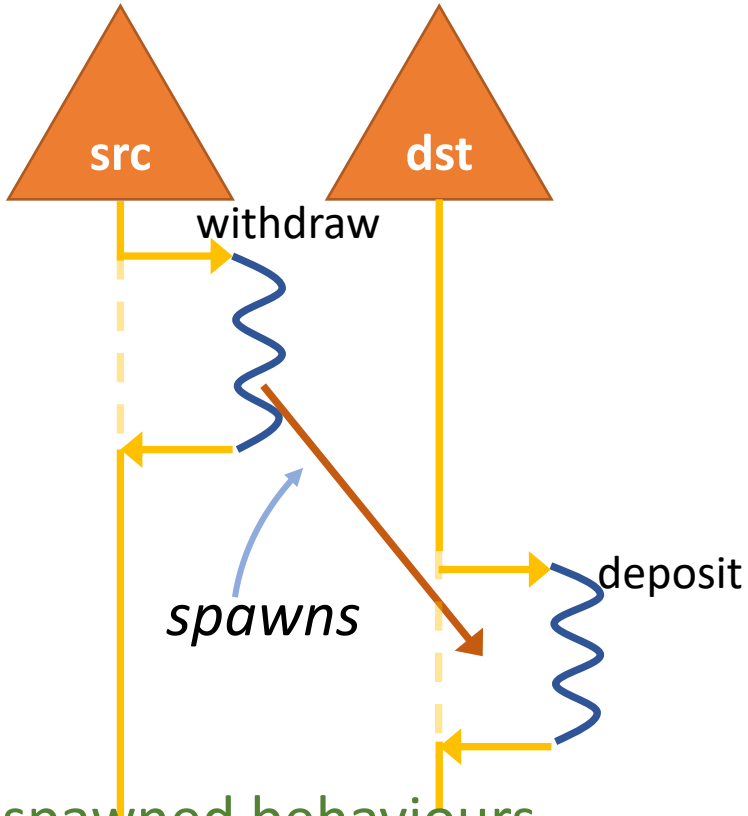
Behaviours spawning behaviours

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        when (dst) { dst.balance += amount; }
8.      }
9.    }
10.}
```



Behaviours spawning behaviours

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        when (dst) { dst.balance += amount; }
8.      }
9.    }
10.}
```

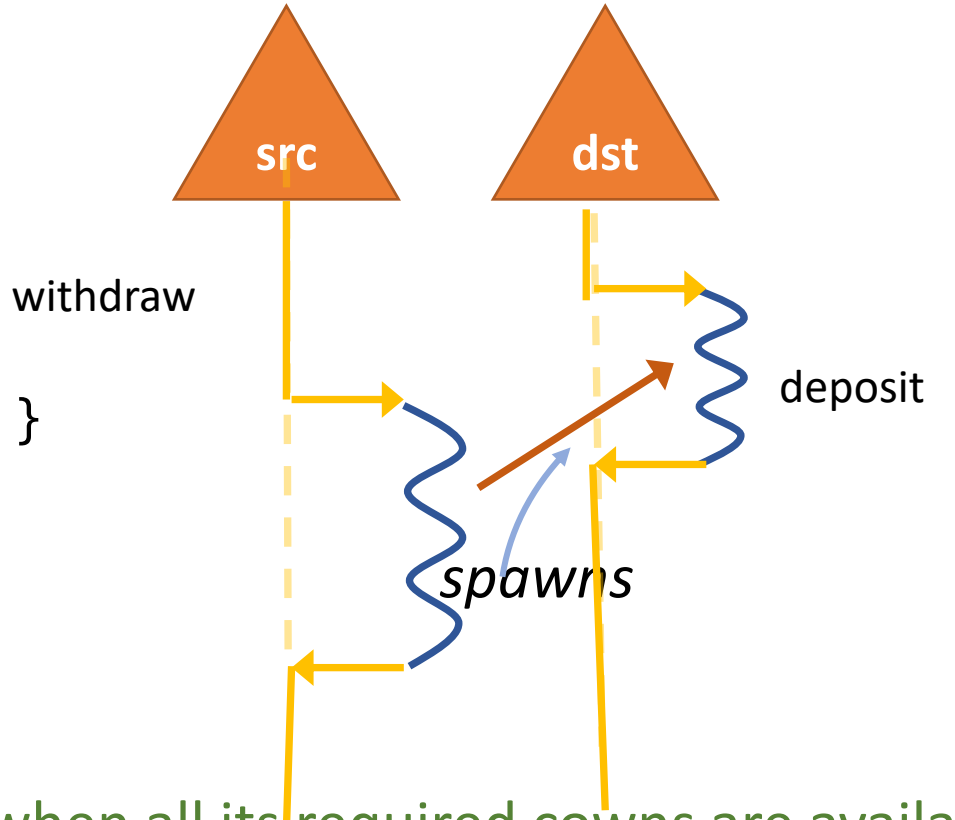


Is this execution possible? **YES**

Order – a behaviour may run only after all previously spawned behaviours with overlapping cowns have terminated.

Behaviours spawning behaviours

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        when (dst) { dst.balance += amount; }
8.      }
9.    }
10.}
```



Is this execution possible? **NO**

Behaviour - once spawned, a behaviour may *run* when all its required cowns are available.

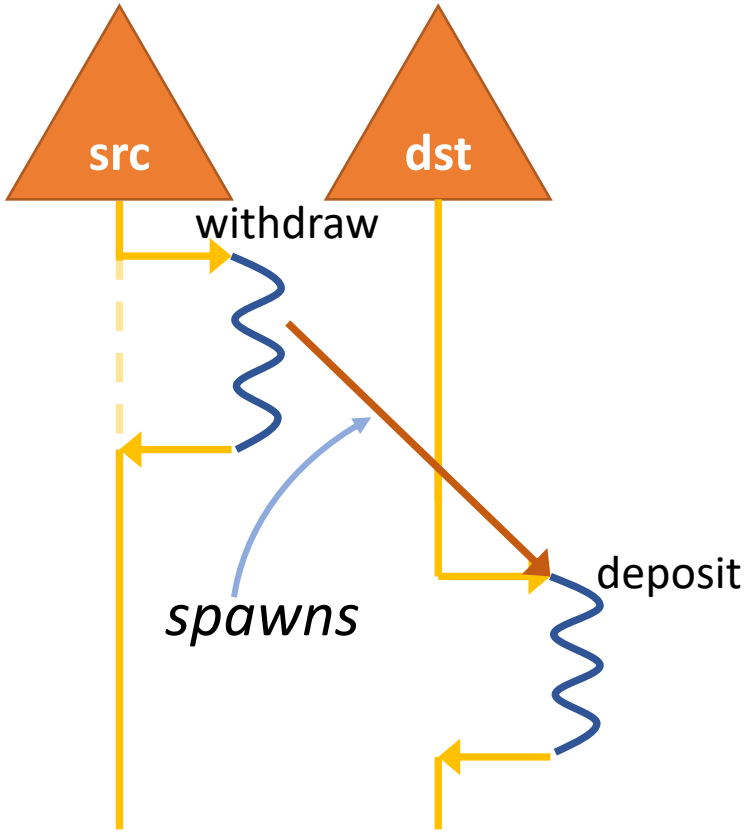
Behaviours spawning behaviours – data race freedom

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        when (dst) { dst.balance += amount; }
8.      }
9.    }
10. }
```

`src` is cown[Account] here

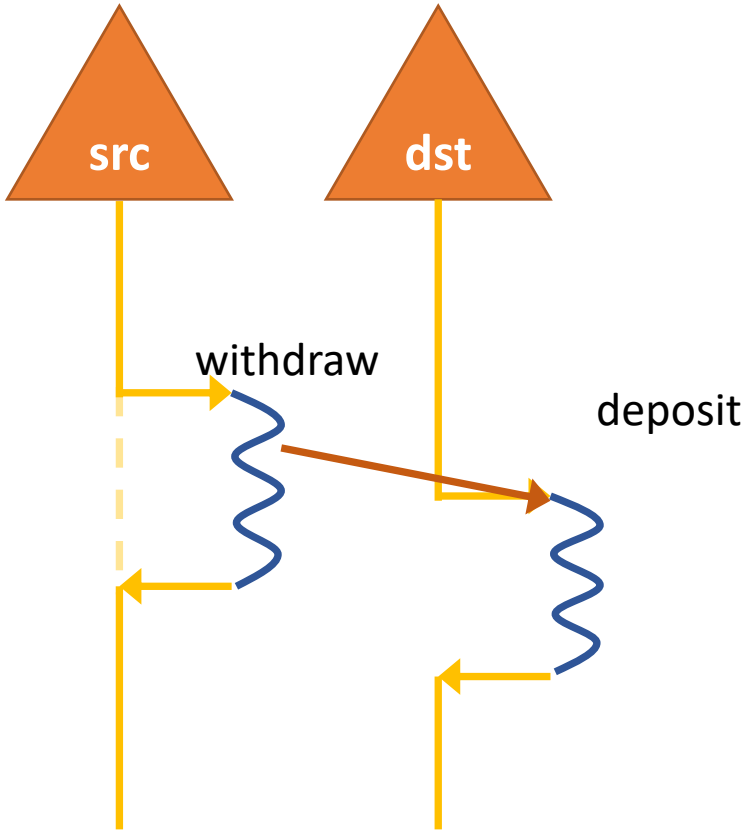
`src` Account here

`src` is cown[Account] here



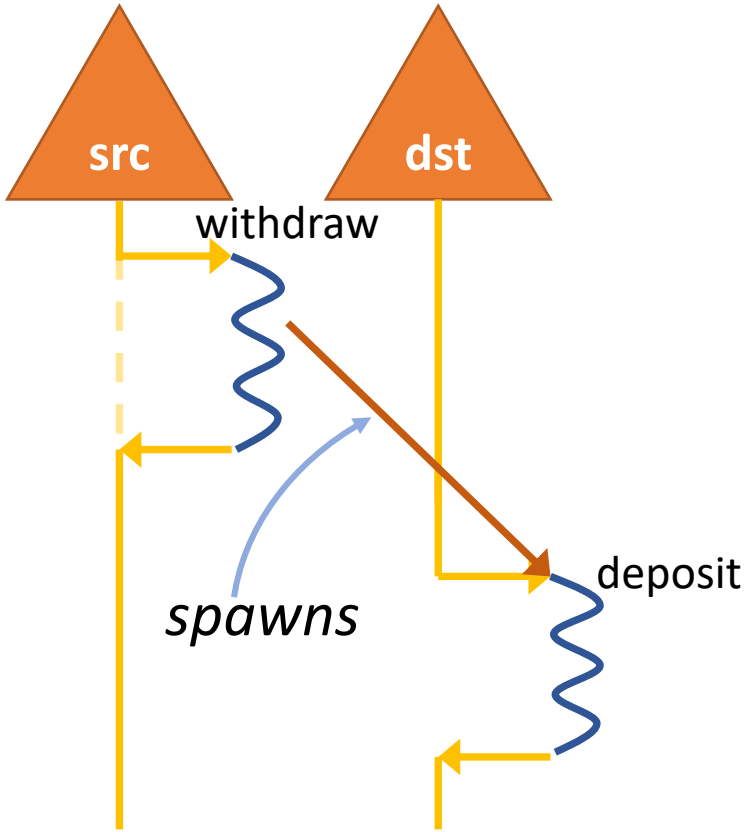
Behaviours spawning behaviours – Deadlock Freedom

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        when (dst) { dst.balance += amount; }
8.      }
9.    }
10.}
```



Behaviours spawning behaviours – Deadlock Freedom

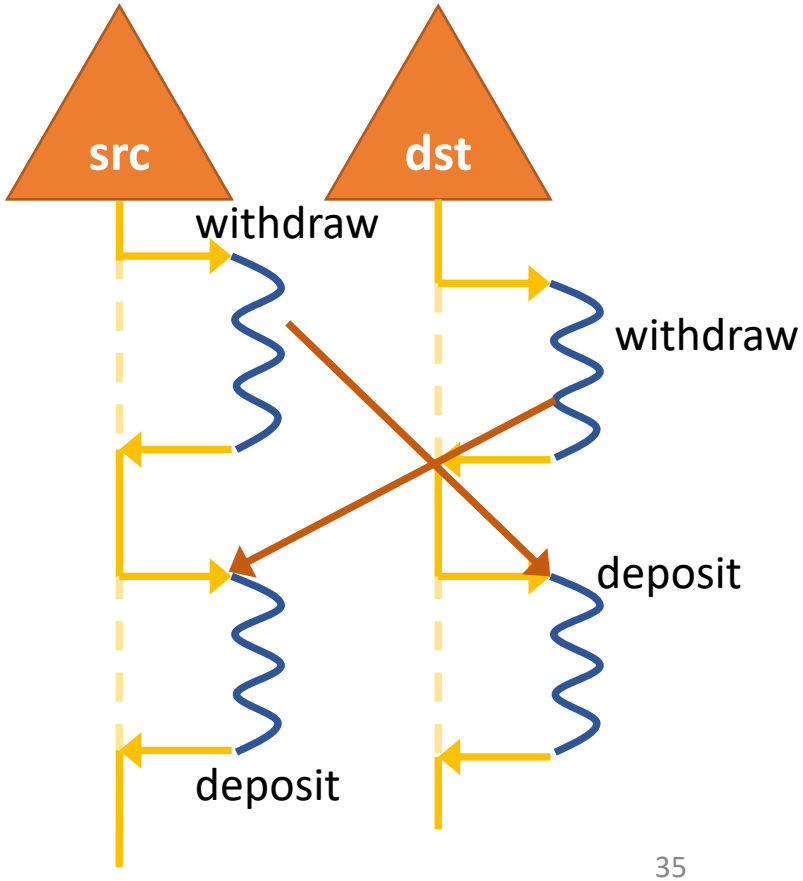
```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        when (dst) { dst.balance += amount; }
8.      }
9.    }
10.}
```



Deadlock Freedom - more

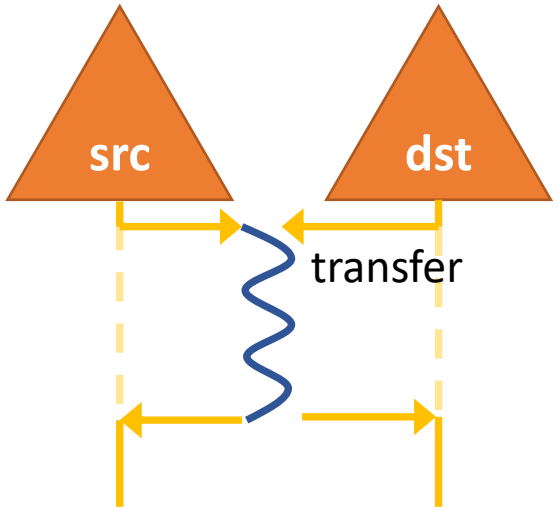
```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        when (dst) { dst.balance += amount; }
8.      }
9.    }
10.}

11.…
12.  transfer(src, dst, 10);
13.  transfer(dst, src, 20);
```



Behaviours acquiring multiple cowns -- Flexible Coordination

```
1.    transfer(src: cown[Account],
2.             dst: cown[Account],
3.             amount: U64) {
4.    when (src, dst) {
5.      if (src.balance >= amount) {
6.        src.balance -= amount;
7.        dst.balance += amount;
8.      }
9.    }
10.}
```



Order Matters

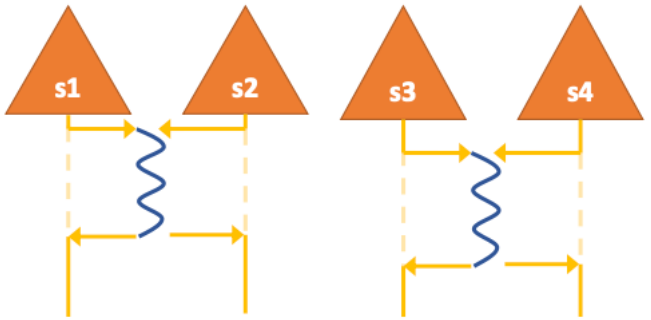
```
1.  main(a1, a2, a3: cown[Account]) {
2.    ….
3.        transfer(a1,a2,100);  // transfer_1
4.        transfer(a2,a3,300);  // transfer_2
5.    …
6.  }
```

Can transfer_1 and transfer_2 execute concurrently?	**NO**
Does it matter whether transfer_1 or transfer_2 execute first?	**YES**
May transfer_1 execute before transfer_2?	**YES**
May transfer_2 execute before transfer_1?	**NO**

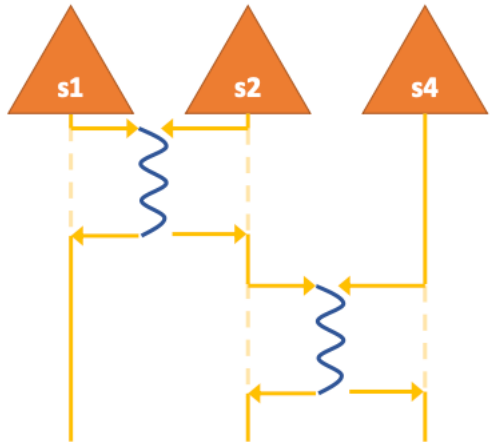
Order is dynamically determined

```
1.  main(s1, s2, s3, s4: cown[Account]) {
2.    ….
3.        transfer(s1,s2,100);  // transfer_1
4.        transfer(s3,s4,300);  // transfer_2
5.    …
6.  }
```

No aliases

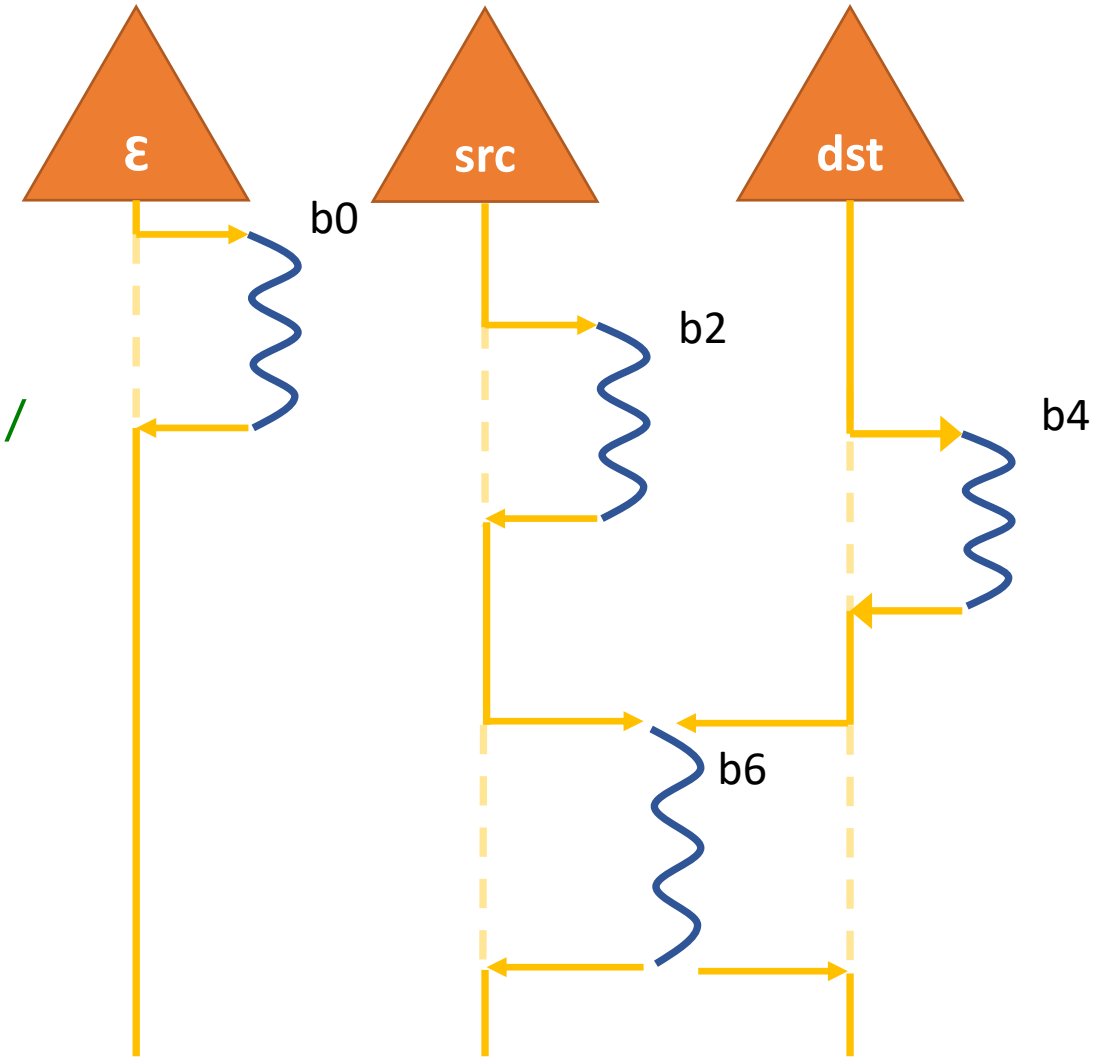


s2 and s3 are aliases, and no other aliases



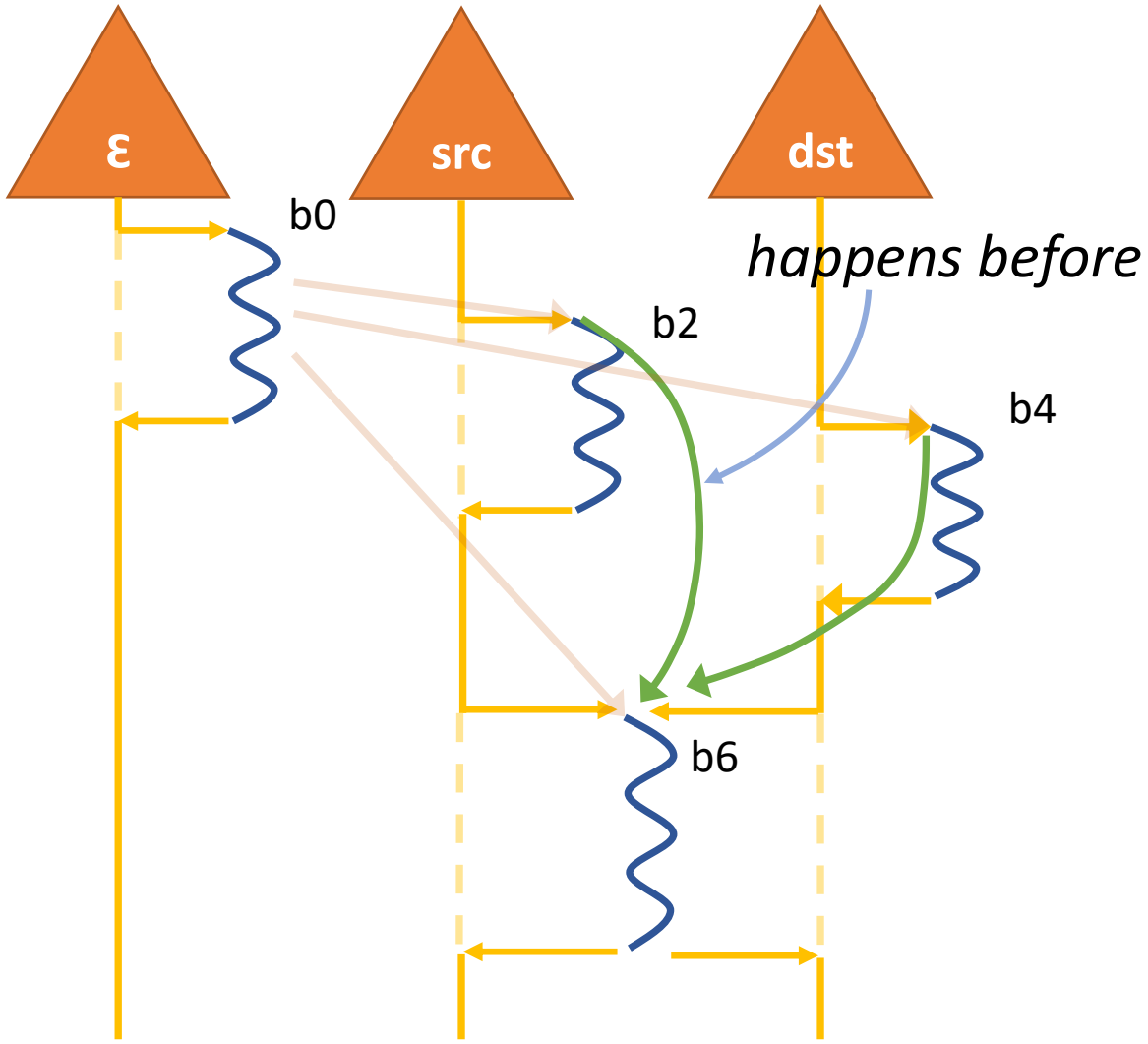
Restricted Determinism

```
1.    main(src: cown[Account],
2.         dst: cown[Account]) { /* b0 */
3.    when (src) { /* b2 */ };
4.    when (dst) { /* b4 */ };
5.    when (src, dst) { /* b6 */ }
6. }
```



Restricted Determinism

```
1.    main(src: cown[Account],
2.         dst: cown[Account]) { /* b0 */
3.    when (src) { /* b2 */ };
4.    when (dst) { /* b4 */ };
5.    when (src, dst) { /* b6 */ }
6. }
```

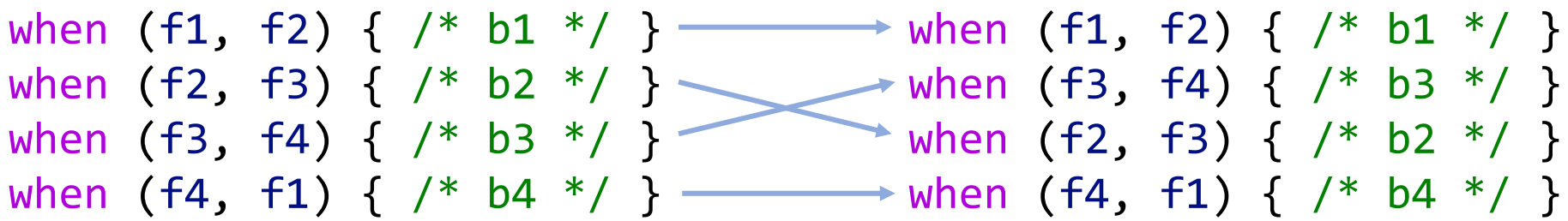


A challenge for the audience

```
when (f1, f2) { /* b1 */ }
when (f2, f3) { /* b2 */ }
when (f3, f4) { /* b3 */ }
when (f4, f1) { /* b4 */ }
```

What can execute in parallel?

An improvement to the solution



Optimal parallelism can be obtained using the paradigm


Semantics

Underlying Sequential Semantics

E Underlying sequential configuration, i.e. a closure

h Global state of the memory

$E, h \hookrightarrow E', h'$ Underlying sequential step

$finished(E)$ Predicate for termination

$E \hookrightarrow_{\text{when } (\overline{\kappa'}) \{E''\}} E'$ Trigger behaviour

Concurent Semantics

R Set of running behaviours (pairs of cowns and a closure $(\overline{\kappa}, E)$)

P List of pending behaviours (pairs of cowns and a closure $(\overline{\kappa}, E)$)

Concurrent Semantics

$$R, P, h \rightsquigarrow R, P, h$$

Concurrent Semantics

$$R, P, h \rightsquigarrow R, P, h$$

$$\textbf{STEP}\ \frac{E, h \hookrightarrow E', h'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P, h'} \qquad \textbf{SPAWN}\ \frac{E \hookrightarrow_{\text{when } (\overline{\kappa'})\ \{E''\}} E'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P : (\overline{\kappa'}, E''), h}$$

$$\textbf{RUN}\ \frac{\left(\bigcup_{(\overline{\kappa'}, \_) \in (P' \cup R)} \overline{\kappa'}\right) \cap \overline{\kappa} = \varnothing}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R \uplus (\overline{\kappa}, E), P' : P'', h} \qquad \textbf{END}\ \frac{\mathit{finished}(E)}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R, P, h}$$

Execution – Deadlock freedom

Execution – Deadlock freedom

$$\{(\varnothing, \texttt{transfer(s1, s2, 1); transfer(s2, s1, 2)})\}, [], h \rightsquigarrow^*$$

$$\varnothing, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}}), (\{s2\}, \texttt{w2; when(s1) \{ d1 \}})], h \rightsquigarrow$$

$$\{(\{s2\}, \texttt{w2; when(s1) \{ d1 \}})\}, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}})], h \rightsquigarrow$$

$$\varnothing, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}}), (\{s1\}, \texttt{d1})], h' \rightsquigarrow^*$$

$$\{(\{s1\}, \texttt{w1; when(s2) \{ d2 \}})\}, [(\{s1\}, \texttt{d1})], h' \rightsquigarrow^*$$

$$\varnothing, [(\{s1\}, \texttt{d1}), (\{s2\}, \texttt{d2})], h' \rightsquigarrow^*$$

$$\{(\{s1\}, \texttt{d1}), (\{s2\}, \texttt{d2})\}, [], h''$$


Implementing the runtime

Main implementation challenges: represent *P*

$$\textsc{step}\ \frac{E, h \hookrightarrow E', h'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P, h'} \qquad \textsc{spawn}\ \frac{E \hookrightarrow_{\text{when } (\overline{\kappa'})\ \{E''\}} E'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P : (\overline{\kappa'}, E''), h}$$

$$\textsc{run}\ \frac{\left(\bigcup_{(\overline{\kappa'}, \_) \in (P' \cup R)} \overline{\kappa'}\right) \cap \overline{\kappa} = \varnothing}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R \uplus (\overline{\kappa}, E), P' : P'', h} \qquad \textsc{end}\ \frac{\mathit{finished}(E)}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R, P, h}$$

Efficient selection of a behaviour to dispatch

Efficient appending of spawned behaviours to end of queue

Concurrent updates of *P*

Atomicity of updates of *P*

Representing *P*

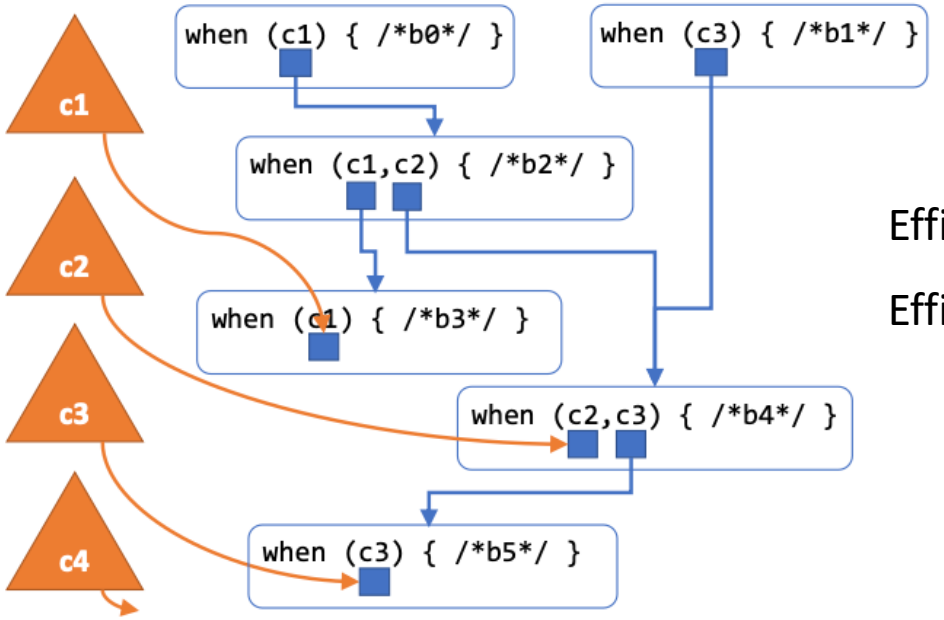
P is represented through a “dependency” graph; edges represent holding of a requested cown.

A behaviour requiring n cowns has n’ predecessors (n’<=n), meaning that it has already acquired n-n’ cowns.

We can dispatch any behaviour with no predecessors.

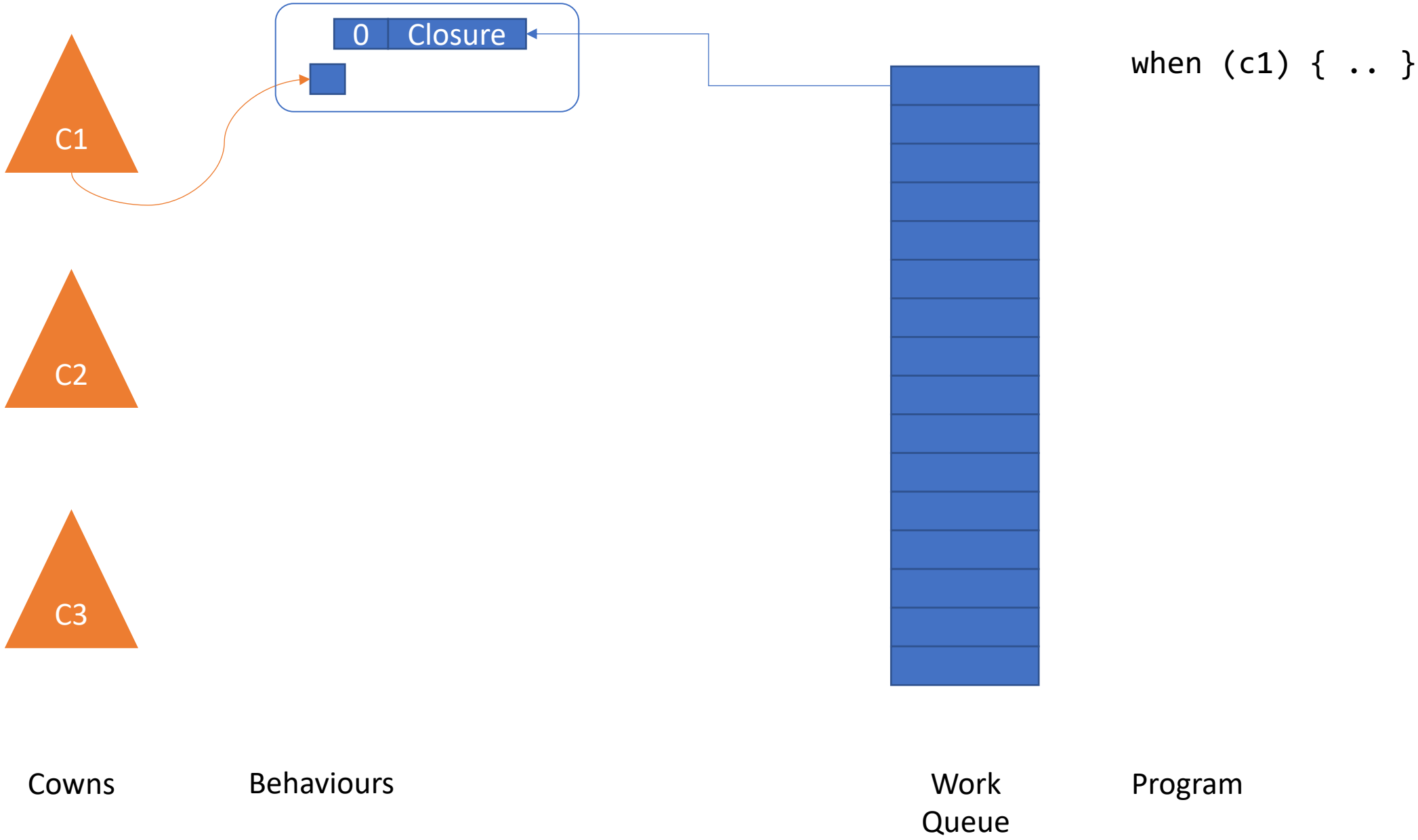
Spawning a behaviour attaches it to the ends of the queues of all its required cowns

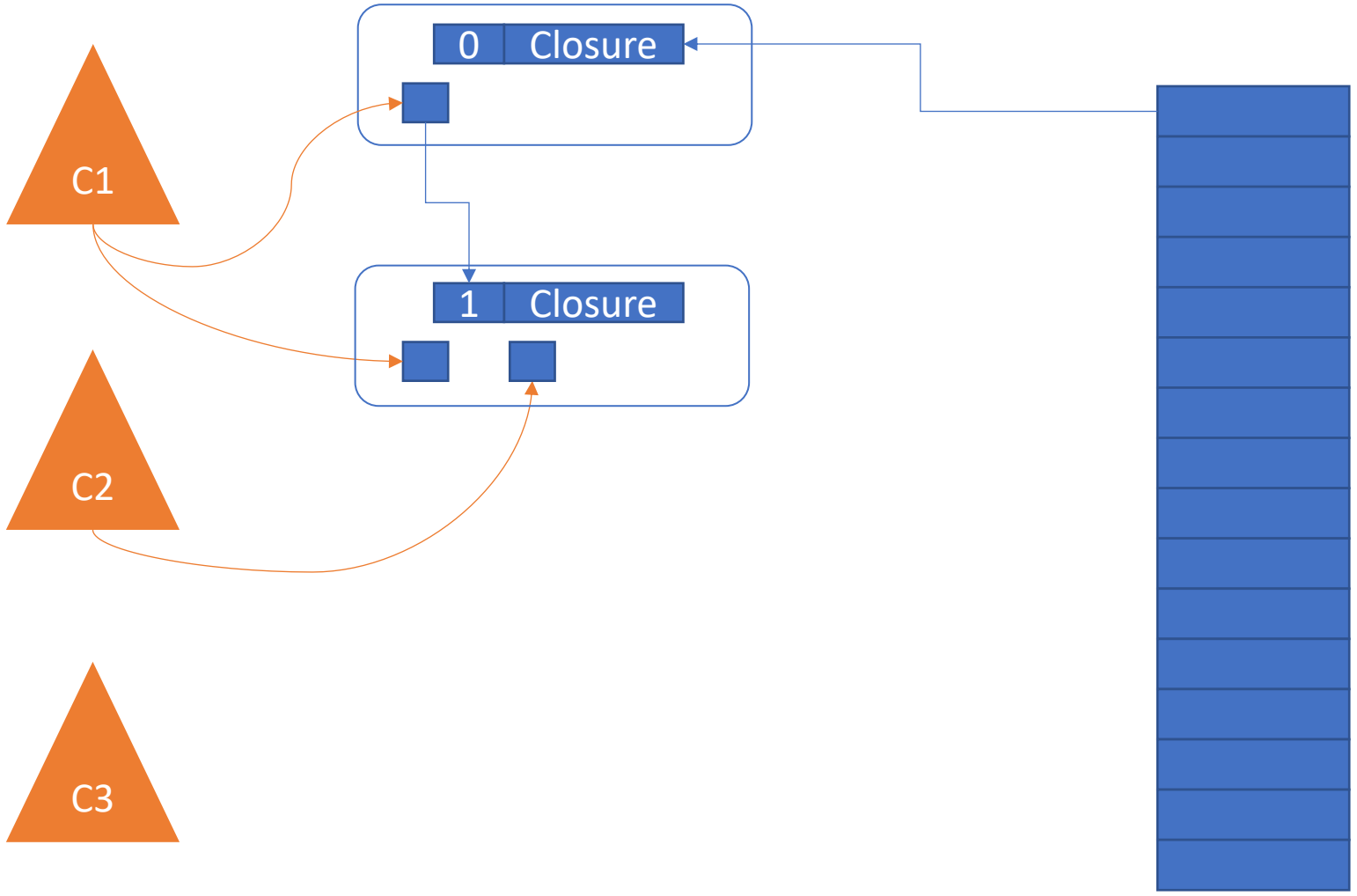
```
1
2   when (c1) { /* b0 */ }
3
4   when (c3) { /* b1 */ }
5
6   when (c1, c2) { /* b2 */ }
7
8   when (c1) { /* b3 */ }
9
10  when (c2, c3) { /* b4 */ }
11
12  when (c3) { /* b5 */ }
```



Efficient behaviour dispatch

Efficient behaviour spawning


0
Closure
C1
C2
C3
when (c1) { .. }
Cowns
Behaviours
Work Queue
Program



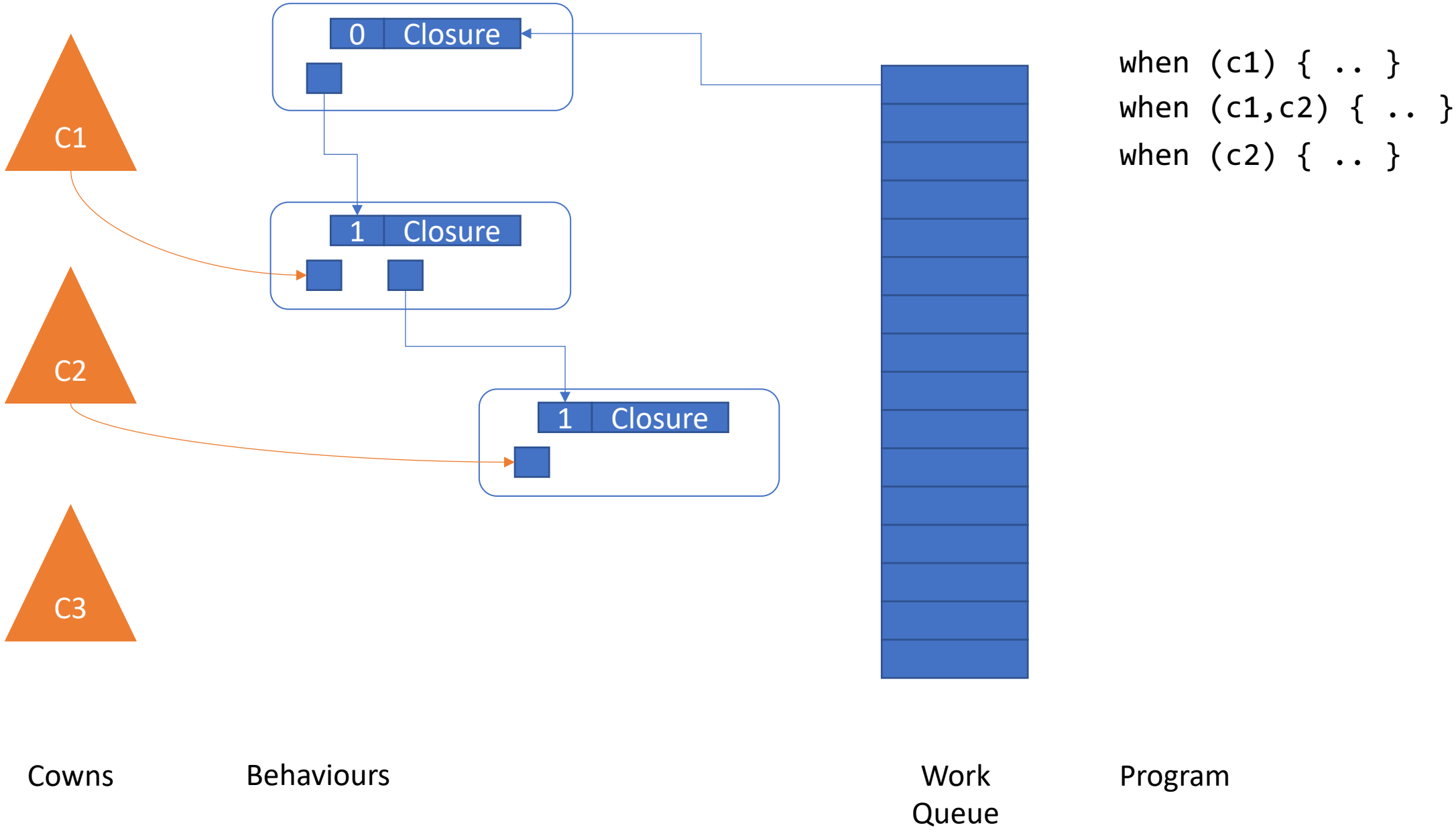

```
when (c1) { .. }
when (c1,c2) { .. }
```

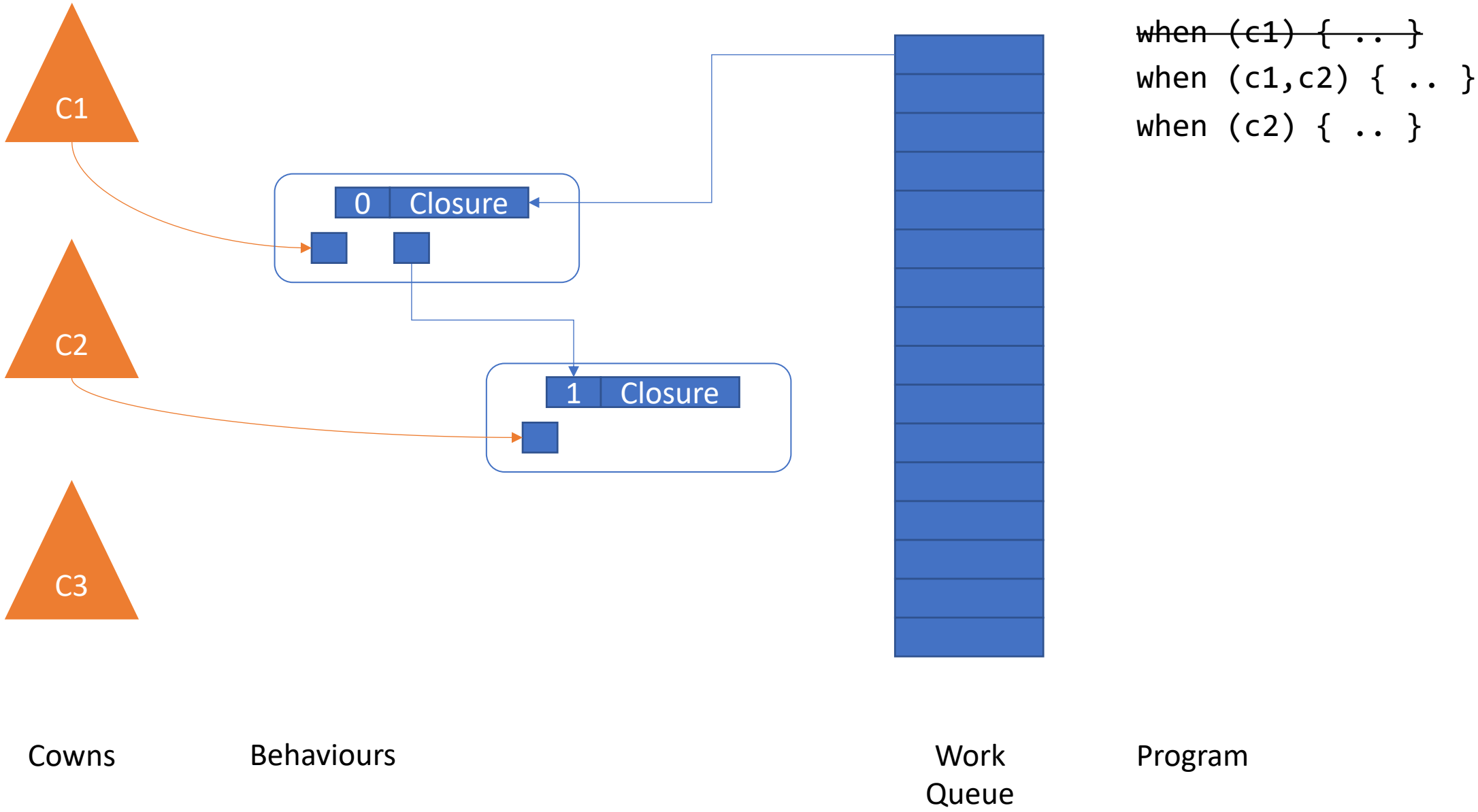
Cowns

Behaviours

Work Queue

Program


0 Closure
1 Closure
1 Closure
C1
C2
C3
when (c1) { .. }
when (c1,c2) { .. }
when (c2) { .. }
Cowns
Behaviours
Work Queue
Program



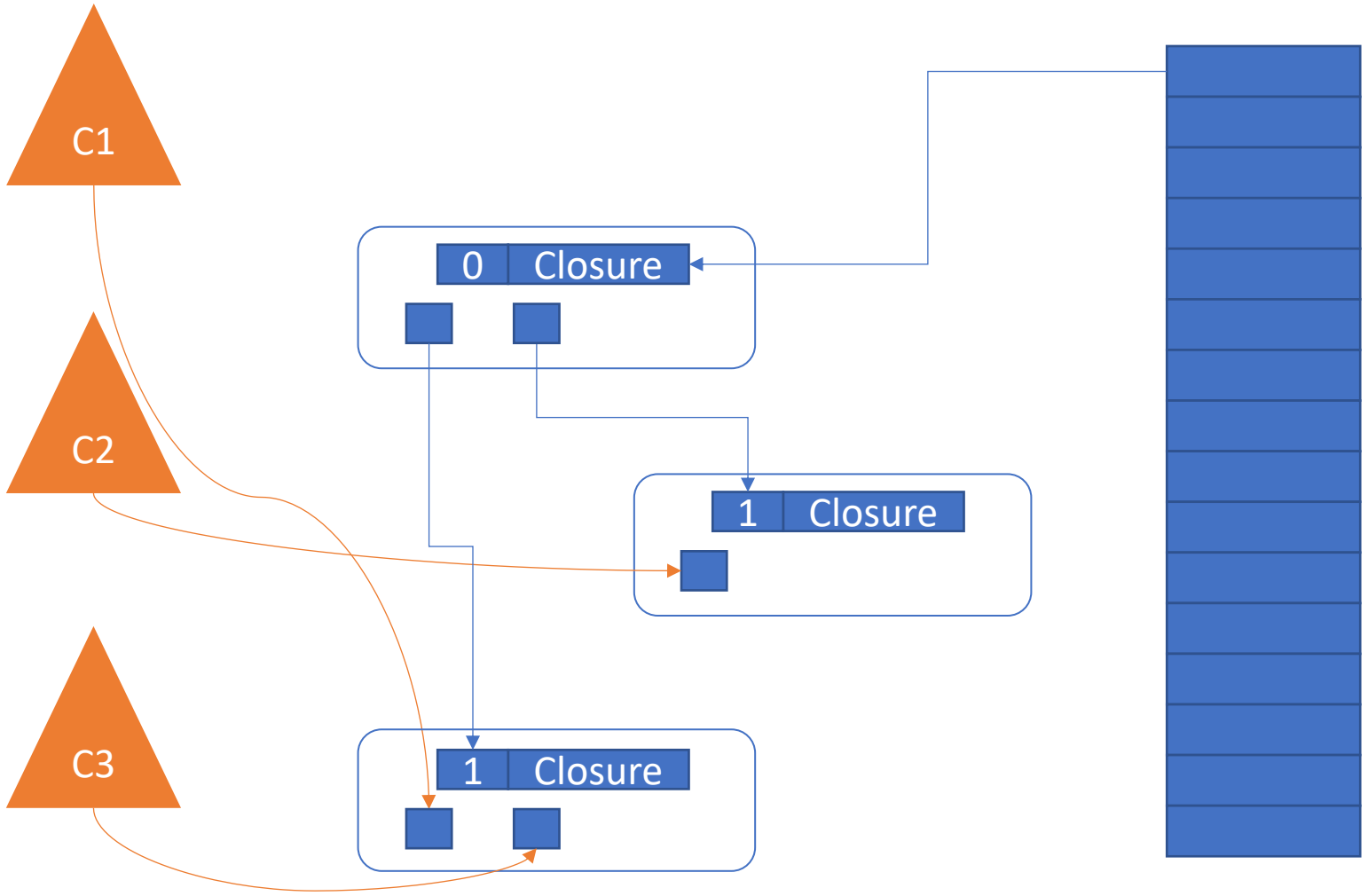
~~when (c1) { .. }~~
when (c1,c2) { .. }
when (c2) { .. }

Cowns

Behaviours

Work Queue

Program



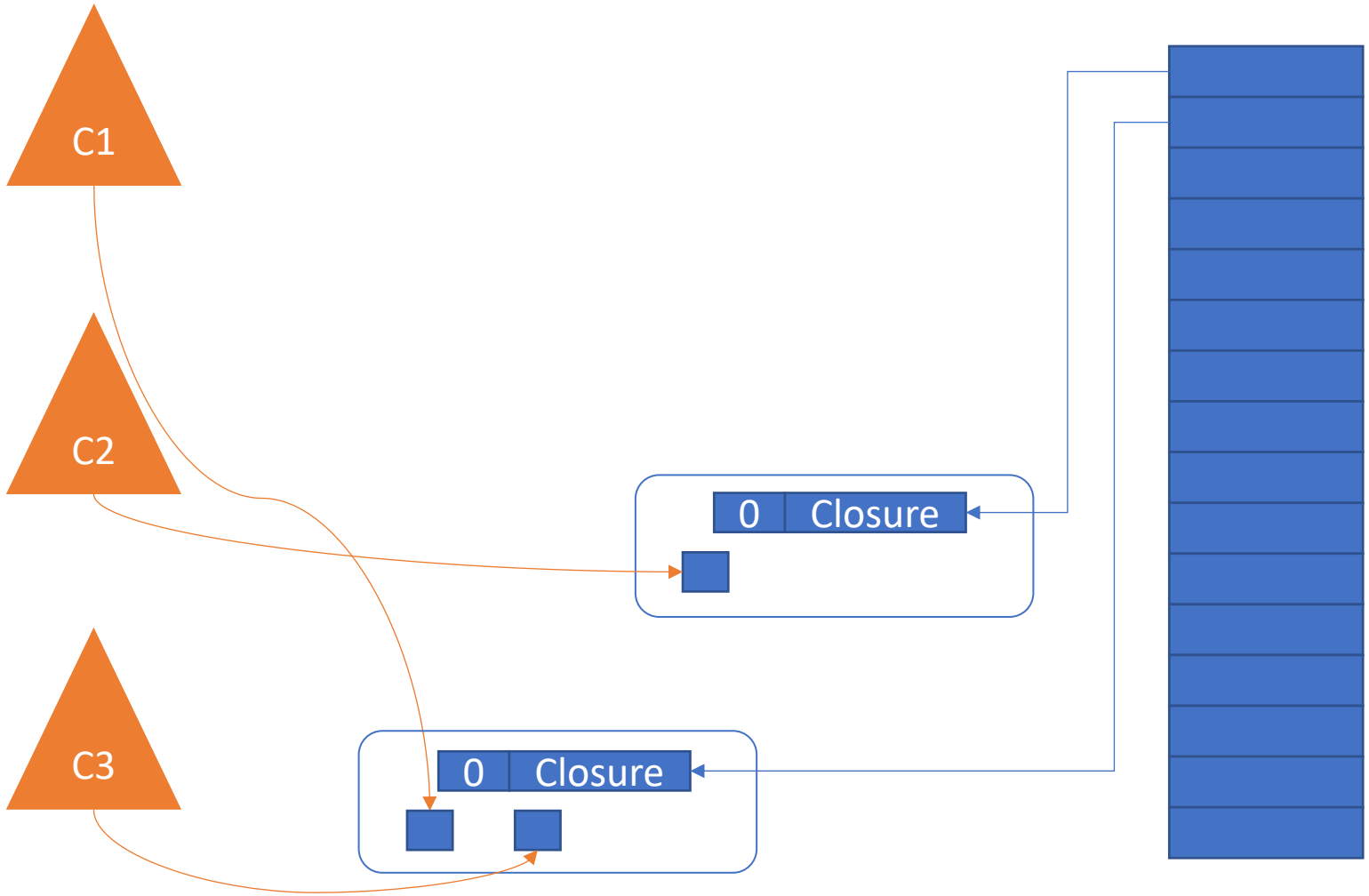
~~when (c1) { .. }~~
when (c1,c2) { .. }
when (c2) { .. }
when (c1,c3) { .. }

Cowns

Behaviours

Work Queue

Program



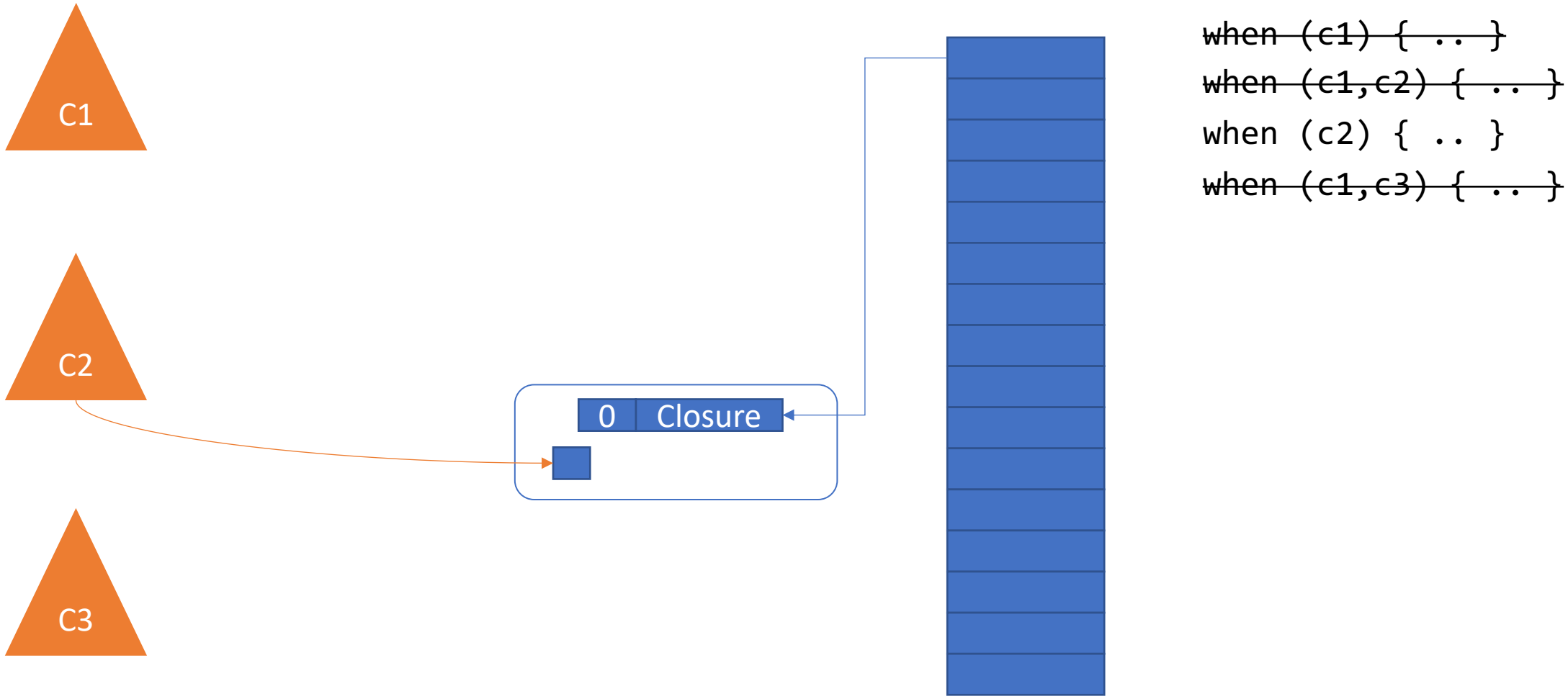
~~when (c1) { .. }~~
~~when (c1,c2) { .. }~~
when (c2) { .. }
when (c1,c3) { .. }

Cowns

Behaviours

Work Queue

Program



~~when (c1) { .. }~~
~~when (c1,c2) { .. }~~
when (c2) { .. }
~~when (c1,c3) { .. }~~

Cowns

Behaviours

Work Queue

Program

Main implementation challenges: represent *P*

$$\textsc{step}\ \frac{E, h \hookrightarrow E', h'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P, h'} \qquad \textsc{spawn}\ \frac{E \hookrightarrow_{\text{when } (\overline{\kappa'})\ \{E''\}} E'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P : (\overline{\kappa'}, E''), h}$$

$$\textsc{run}\ \frac{\left(\bigcup_{(\overline{\kappa'}, \_) \in (P' \cup R)} \overline{\kappa'}\right) \cap \overline{\kappa} = \varnothing}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R \uplus (\overline{\kappa}, E), P' : P'', h} \qquad \textsc{end}\ \frac{\mathit{finished}(E)}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R, P, h}$$

~~Efficient behaviour dispatch~~

~~Efficient behaviour spawning~~

Concurrent updates of *P*

Atomicity of updates of *P*

Key ideas

- MCS-Queue Lock
- Two phase “enqueue”

```
class CownBase : StableOrder {
  volatile Request? last = null;
}

class Request {
  volatile Behaviour? next = null;
  volatile bool scheduled = false;
  CownBase target;

  Request(CownBase t) { target = t; }

  void StartEnqueue(Behaviour behaviour) {
    var prev = Exchange(ref target.last, this);

    if (prev == null) {
      behaviour.ResolveOne();
      return;
    }
    while (!prev.scheduled) { /*spin*/ }
    prev.next = behaviour;
  }

  void FinishEnqueue() { scheduled = true; }

  void Release() {
    if (next == null) {
      if (CompareExchange(ref target.last,
                          null, this) == this)
        return;
      while (next == null) { /*spin*/ }
    }
    next.ResolveOne();
  }
}
```

```
class Behaviour {
  Action thunk;
  int count;
  Request[] requests;

  Behaviour(Action t, CownBase[] cowns) {
    thunk = t;
    requests = new Request[cowns.Length];
    for (int i = 0; i < cowns.Length; i++)
      requests[i] = new Request(cowns[i]);
  }

  static void Schedule(Action t,
                       params CownBase[] cowns) {
    Array.Sort(cowns);
    var behaviour = new Behaviour(t, cowns);
    behaviour.count = cowns.Length + 1;
    foreach (var r in behaviour.requests)
      r.StartEnqueue(behaviour);
    foreach (var r in behaviour.requests)
      r.FinishEnqueue();
    behaviour.ResolveOne();
  }

  void ResolveOne() {
    if (Decrement(ref count) != 0)
      return;
    Task.Run(() => {
      thunk();
      foreach (var r in requests)
        r.Release();
    });
} }
```

https://github.com/microsoft/verona-rt/blob/main/docs/internal/concurrency/modelimpl/When.cs

Evaluation of C++ implementation

https://github.com/Microsoft/verona-rt

Savina: BoC versus Actors

Table 2. Average runtime (ms) on selected Savina benchmarks

	BoC (Actor)					BoC (Full)					
										behaviours	
Benchmark	LoC	1 core (ms)	8 cores (ms)	cowns	behaviours	LoC	1 core (ms)	8 cores (ms)	cowns	1 cown	2 cowns
Banking	117	19.8 ± 0.6	22.6 ± 1.6	1001	252407	78	**7.9** ± 1.2	9.3 ± 4.5	1001	50001	50000
Chameneos	105	**46.1** ± 0.2	94.5 ± 1.4	101	800300	85	49.2 ± 0.3	122.1 ± 1.7	101	400100	200100
Count	40	46.1 ± 0.4	47.7 ± 0.4	2	1000002	30	**45.4** ± 0.4	46.9 ± 0.6	2	1000000	1
Dining Philosophers	94	73.3 ± 0.2	164.9 ± 0.9	21	1350118	61	22.2 ± 0.7	**16.6** ± 0.8	41	200020	199980
Fib	51	29.3 ± 0.3	19.4 ± 0.7	150049	300097	28	**10.9** ± 0.4	13.5 ± 0.8	75025	0	75024
Fork-Join Create	42	10.5 ± 0.4	12.4 ± 0.4	40001	80000	42	**9.6** ± 0.3	11.8 ± 0.3	40001	40000	40000
Fork-Join Throughput	53	53.4 ± 0.9	100.8 ± 0.6	61	1200000	52	**35.8** ± 1.2	45.8 ± 2.2	61	600000	60
Logistic Map Series	133	39.0 ± 0.4	41.2 ± 4.0	21	750022	52	**17.4** ± 1.0	19.8 ± 0.9	21	0	250010
Quicksort	121	124.7 ± 0.3	**58.8** ± 0.4	1935	3869	85	105.9 ± 0.3	78.6 ± 0.4	968	968	967
Sleeping Barber	127	1660 ± 0.1	3544 ± 0.9	5003	24027388	106	**12.5** ± 7.7	16.5 ± 10.2	5003	68077	10000
Trapezoid	72	970.0 ± 0.1	**172.0** ± 0.1	101	201	68	958.3 ± 0.1	172.6 ± 0.1	100	100	99

Savina-inspired: Banking

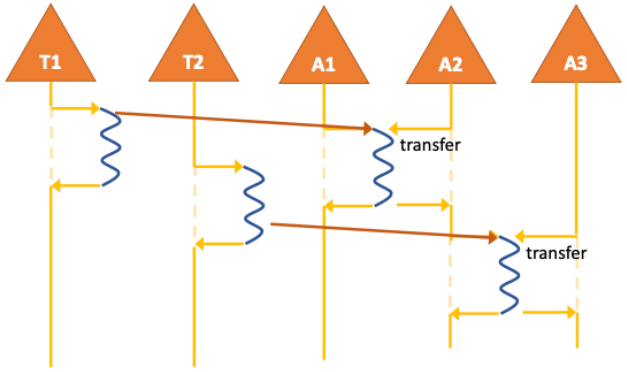
”Reasonable” Banking

- Several Tellers may issue transfers between shared accounts
- Transfers must be atomic
- Order of transfers per teller executed in the order issued
- No deadlock; no livelock

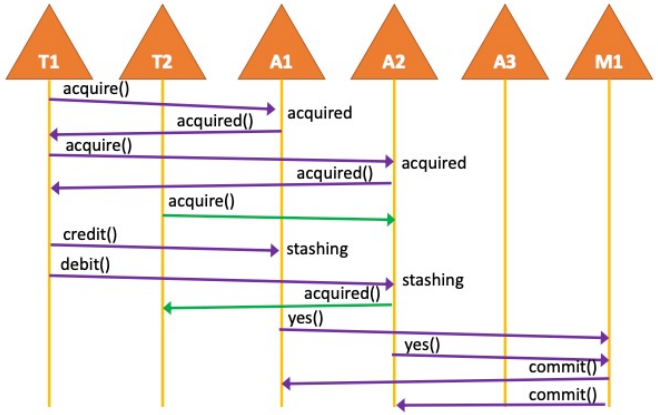
Savina-inspired: Banking

”Reasonable” Banking

- Several Tellers may issue transfers between shared accounts
- Transfers must be atomic
- Order of transfers per teller executed in the order issued
- No deadlock; no livelock



(a) with BoC (Full)



(b) with Actors

Savina-inspired: Banking

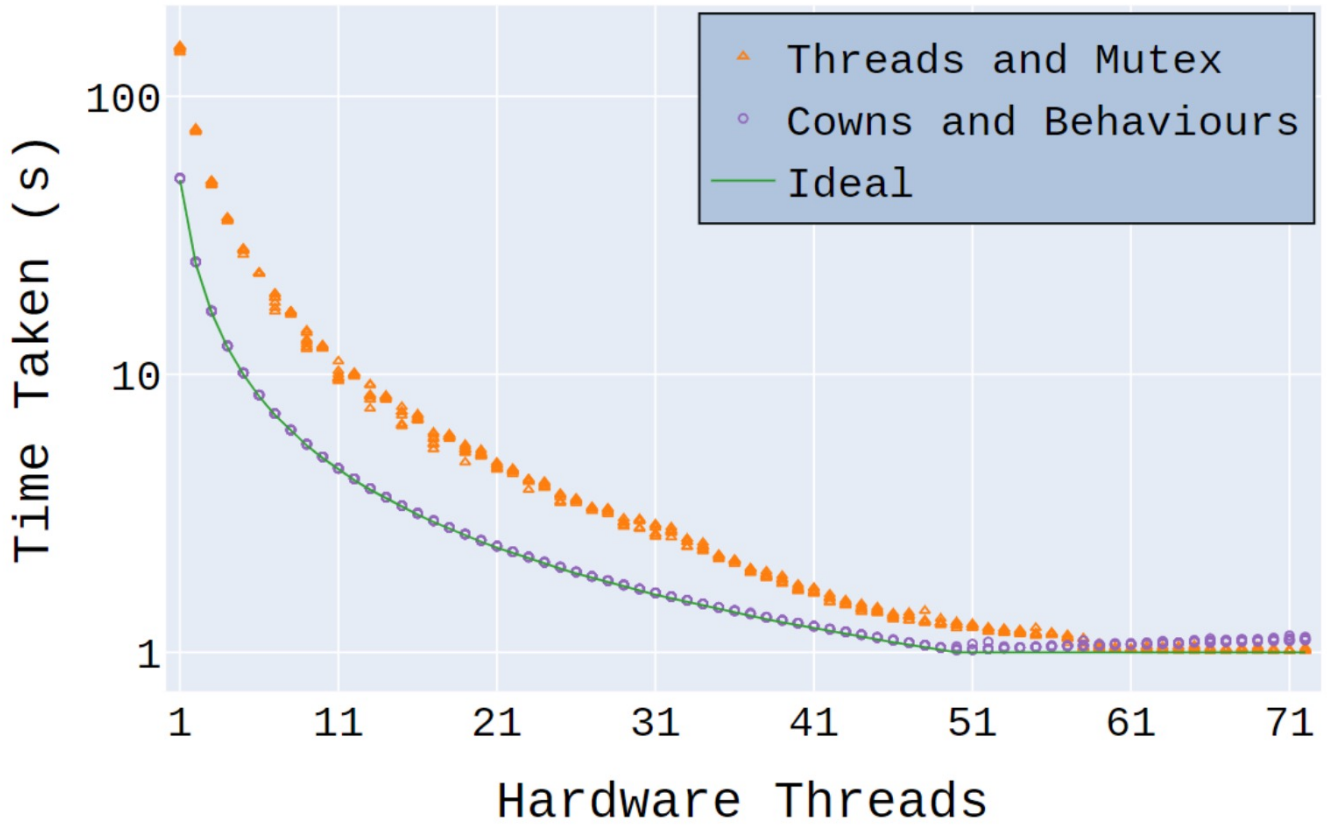
”Reasonable” Banking

- Several Tellers may issue transfers between shared accounts
- Transfers must be atomic
- Order of transfers per teller executed in the order issued
- No deadlock; no livelock

Table 4. Average runtime (ms) and lines of code (LoC) of ReasonableBanking benchmark

BoC (Full)			Pony		
LoC	1 core (ms)	8 cores (ms)	LoC	1 core (ms)	8 cores (ms)
78	7.9 ± 1.2	9.3 ± 4.5	226	987.9 ± 0.33	344.717 ± 0.35

Scaling- Dining Philosophers



What more is happening

Model implementation in C#

Full implementation in C++

Prototype implementation for Python

Performance comparison with Pony on Actor based benchmarks

Integration with ownership-based type systems

Relation to memory models

Promises and Whiteboards

Reasoning with Behaviour Oriented Concurrency

Conclusion: **`when`**, can we have it?

`when`

- is simple, intuitive and expressive,
- provides a single mechanism for both parallelism and coordination,
- it addresses the issue of the Actor model
- combines with ownership types

Why is concurrency important?

Cannot be a library without data-races

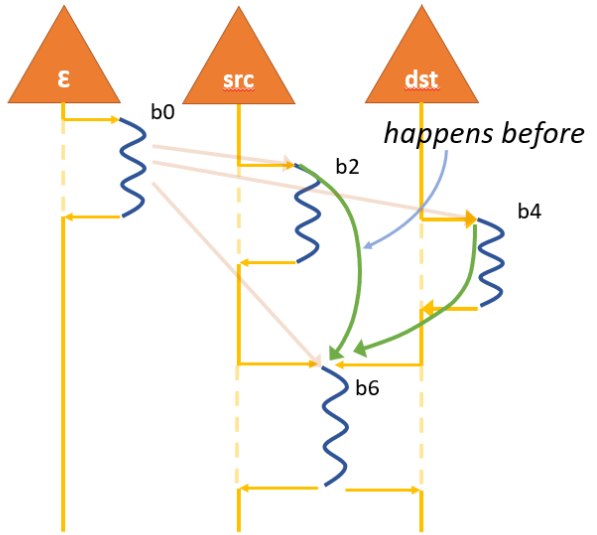
Parallelism
- Threads
- Tasks
- Coroutines
- …

Coordination
- Locks
- Queues
- Transactions
- …

Actors

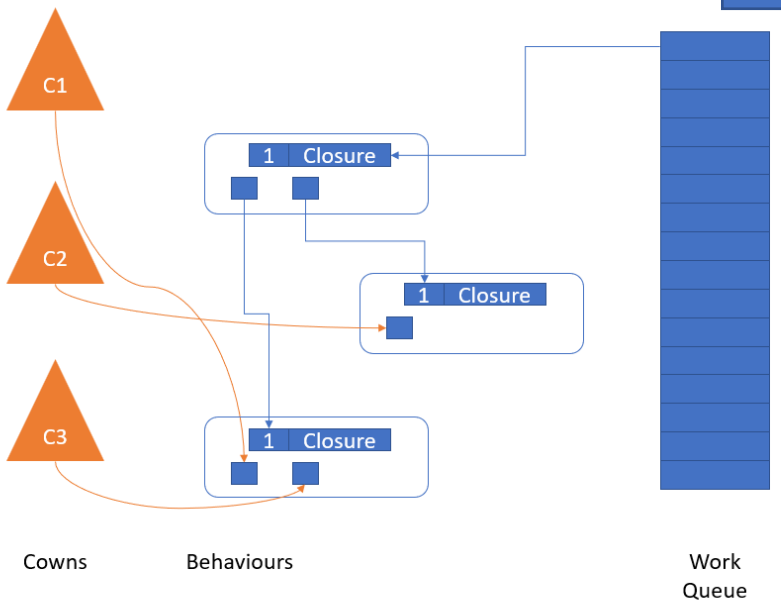
Restricted Determinism

```
1. main(src: cown[Account],
2.      dst: cown[Account]) { /* b0 */
3.   when (src) { /* b2 */ };
4.   when (dst) { /* b4 */ };
5.   when (src, dst) { /* b6 */ }
6. }
```



∀ b, b' . b happens before b' iff cownsof(b) ∩ cownsof(b') = ∅ and ∃ b_0 s.t. b_0 spawns b and subsequently b_0 spawns* b',

http://github.com/Microsoft/verona-rt




```
when (c1) { .. }
when (c1,c2) { .. }
when (c2) { .. }
when (c1,c3) { .. }
```

Program

Scaling- Dining Philosophers