# Lincheck: Testing Concurrent Data Structures in Java

Nikita Koval, Hydra 2019

JET BRAINS

Kotlin

I S T AUSTRIA

This is joint work with
Dmitry Tsitelov, Maria Sokolova,
Roman Elizarov, and Anton Evdokimov

# Speaker: Nikita Koval

@nkoval_

- Graduated @ ITMO University
- Previously worked as developer and research engineer @ Devexperts
- Teaching concurrent programming course @ ITMO University
- Researcher @ JetBrains
- PhD student @ IST Austria

# Writing concurrent code is pain

Writing concurrent code is pain

… testing it is not much easier!

```
var i = 0
```

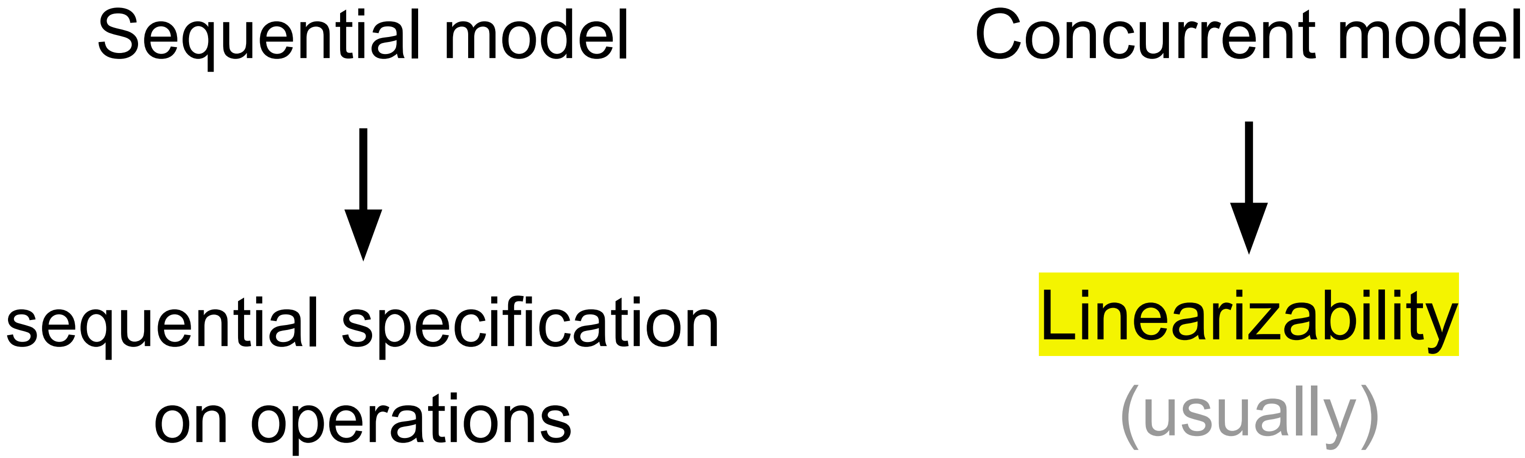
| | |
|---|---|
| `i.inc()` | `i.inc()` |

```
          var i = 0
─────────────────────────────────
i.inc() // 0 │ i.inc() // 1
        // 1 │         // 0
```

```
               var i = 0
-----------------------------------------
i.inc() // 0        |  i.inc() // 0
```

```
          var i = 0
i.inc() // 0    i.inc() // 0
```

We do not expect this!



Sequential model

sequential specification on operations

Concurrent model

Linearizability

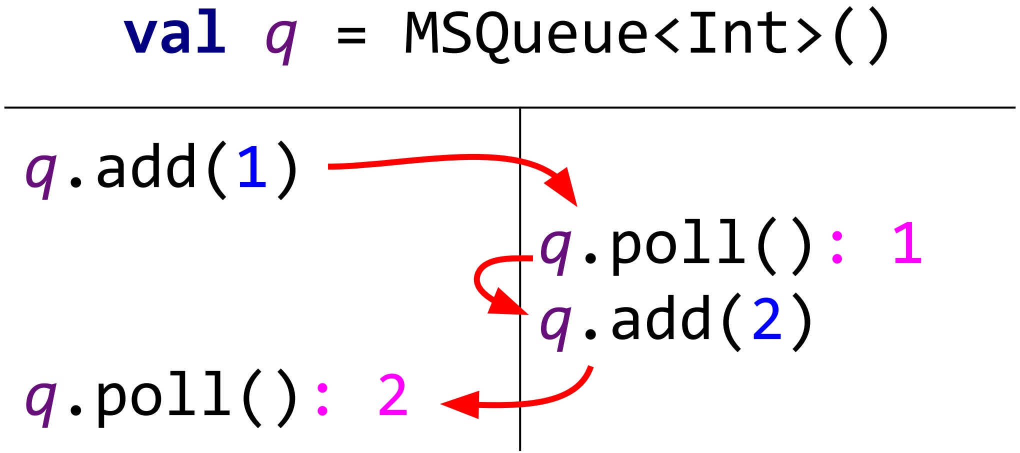
(usually)

Execution ***is linearizable*** ⇔ ∃ equivalent *sequential* execution wrt *happens-before* order (a bit harder)

Execution ***is linearizable*** ⇔ ∃ equivalent *sequential* execution wrt *happens-before* order (a bit harder)

```
         val q = MSQueue<Int>()
-----------------------------------------
q.add(1)          | q.poll(): 1
q.poll(): 2       | q.add(2)
```

Execution ***is linearizable*** ⇔ ∃ equivalent *sequential* execution wrt *happens-before* order (a bit harder)



```
          val q = MSQueue<Int>()
----------------------------------------
q.add(1)           |
                   | q.poll(): 1
                   | q.add(2)
q.poll(): 2        |
          var i = 0
-------------------------------
i.inc() // 0  |  i.inc() // 0
```

This counter is not linearizable

# How to check whether my data structure is linearizable?

# How to check whether my data structure is linearizable?

Formal proofs

# How to check whether my data structure is linearizable?

Formal proofs

Model checking

# How to check whether my data structure is linearizable?

Formal proofs

Testing

Model checking

# How to check whether my data structure is linearizable?

Formal proofs

**Testing**

Model checking

# How does the ideal test look?

# How does the ideal test look?

```
class MSQueueTest {
    val q = MSQueue<Int>()















}
```

Initial state

# How does the ideal test look?

```
class MSQueueTest {
    val q = MSQueue<Int>()

    @Operation fun add(element: Int) =
        q.add(element)

    @Operation fun poll() = q.poll()


}
```

Operations on the data structure

# How does the ideal test look?

```
class MSQueueTest {
    val q = MSQueue<Int>()

    @Operation fun add(element: Int) =
        q.add(element)

    @Operation fun poll() = q.poll()


}
```

Operation parameters can be non-fixed!

# How does the ideal test look?

```
class MSQueueTest {
   val q = MSQueue<Int>()

   @Operation fun add(element: Int) =
         q.add(element)

   @Operation fun poll() = q.poll()

   @Test fun runTest() =
         LinChecker.check(QueueTest::class)
}
```

**The Magic Button**

# How does the ideal test look?

```
class MSQueueTest {
   val q = MSQueue<Int>()

   @Operation fun add(element: Int) =
         q.add(element)

   @Operation fun poll() = q.poll()

   @Test fun runTest() =
         LinChecker.check(QueueTest::class)
}
```

Do we have such instrument?

# How does the ideal test look?

```
class MSQueueTest {
   val q = MSQueue<Int>()

   @Operation fun add(element: Int) =
         q.add(element)

   @Operation fun poll() = q.poll()

   @Test fun runTest() =
         LinChecker.check(QueueTest::class)
}
```

Do we have such instrument?

YEEES!

# Lin-Check Overview

*Lincheck* = **Lin**earizability **Check**er (supports not only linearizability)

https://github.com/Kotlin/kotlinx-lincheck

# Lin-Check Overview

*Lincheck* = **Lin**earizability **Check**er (supports not only linearizability)

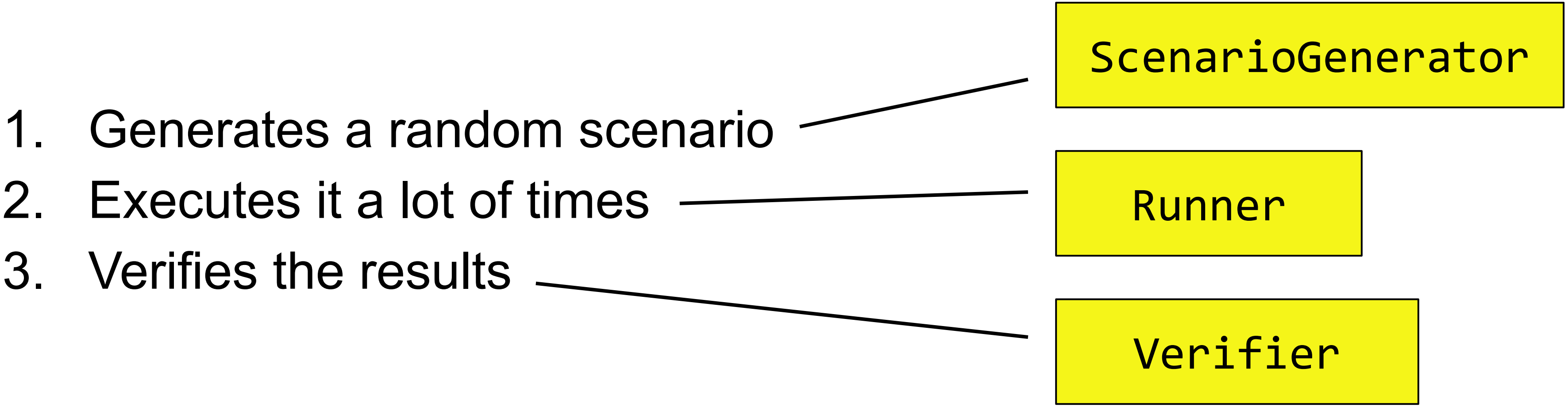
https://github.com/Kotlin/kotlinx-lincheck

1. Generates a random scenario
2. Executes it a lot of times
3. Verifies the results

# Lin-Check Overview

*Lincheck* = **Lin**earizability **Check**er (supports not only linearizability)

https://github.com/Kotlin/kotlinx-lincheck



1. Generates a random scenario
2. Executes it a lot of times
3. Verifies the results

ScenarioGenerator

Runner

Verifier

# Invalid Execution Example

```
Init part:
[poll(): null, add(9)]
Parallel part:
| poll(): null | add(4)    |
| add(3)       | add(6)    |
| poll(): 4    | poll(): 3 |
Post part:
[add(1), poll(): 6]
```

# How to check results for correctness?

Simplest solution:

1. Generate all possible sequential histories
2. Check whether one of them produces the same results

# How to check results for correctness?

Simplest solution:

1. Generate all possible sequential histories
2. Check whether one of them produces the same results

2 threads x 15 operations ⇒ OutOfMemoryError

# How to check results for correctness?

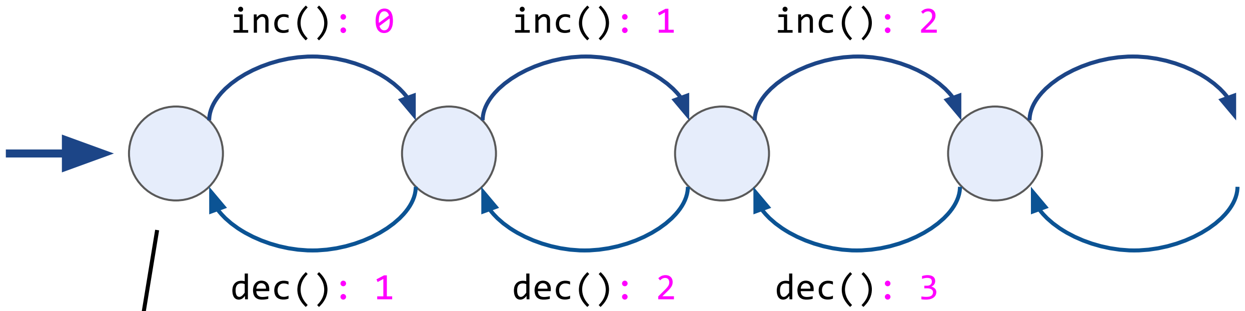
Simplest solution:

1. Generate all possible sequential histories
2. Check whether one of them produces the same results

Smarter solution: Labeled Transition System (LTS)

# LTS (Labeled Transition System)

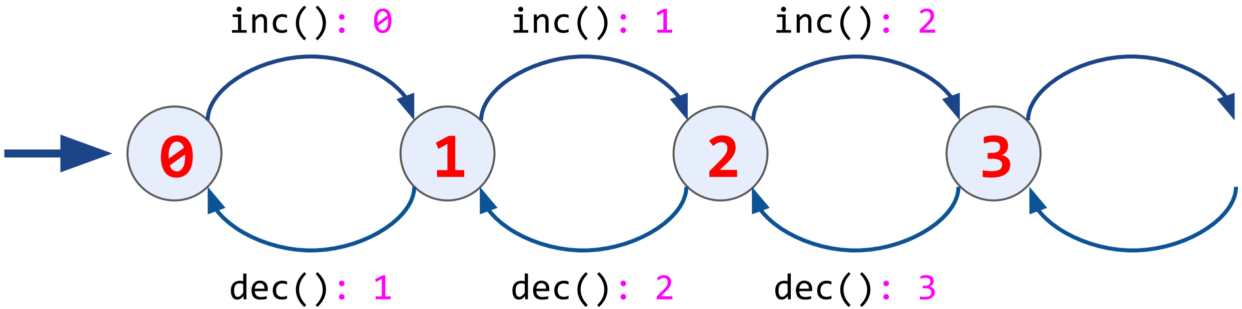
LTS is infinite



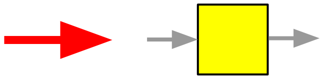
Initial state

Operation with result

# LTS (Labeled Transition System)

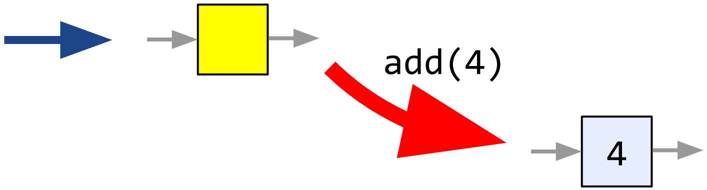


# LTS-based verification



```
        val q = MSQueue<Int>()
-------------------------------------
q.add(4)          | q.poll(): 4
q.poll(): 9       | q.add(9)
```

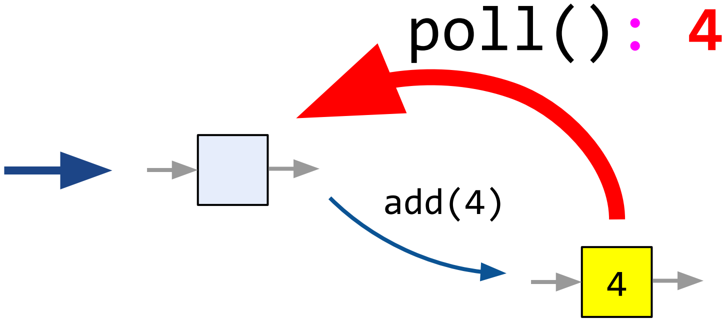
# LTS-based verification



```
val q = MSQueue<Int>()
```

| | |
|---|---|
| q.add(4)<br>q.poll(): 9 | q.poll(): 4<br>q.add(9) |

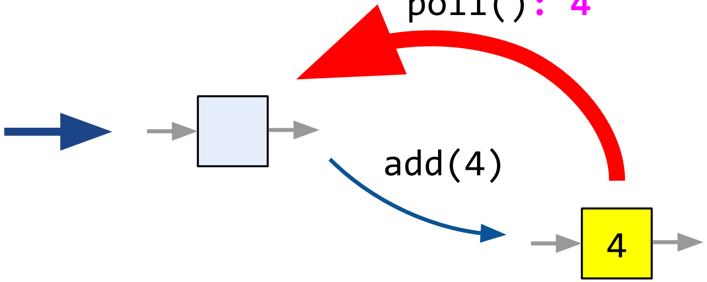
# LTS-based verification



```
       val q = MSQueue<Int>()
--------------------------------------
q.add(4)          | q.poll(): 4
q.poll(): 9       | q.add(9)
```

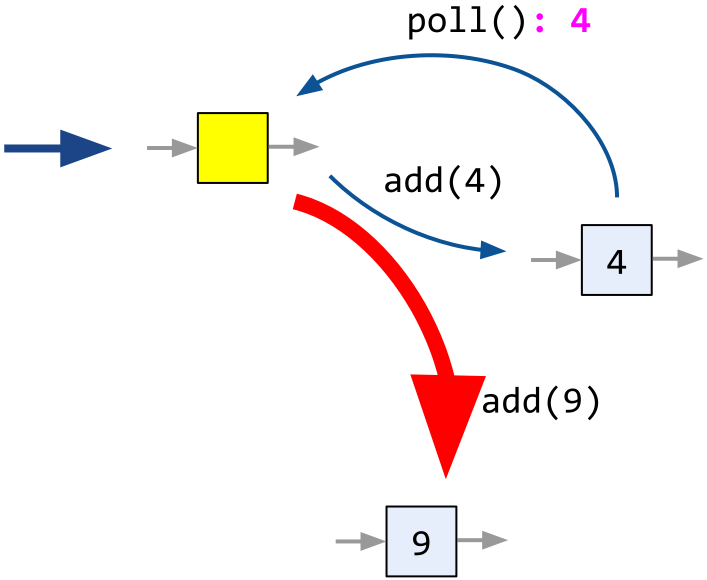
Result is different

# LTS-based verification



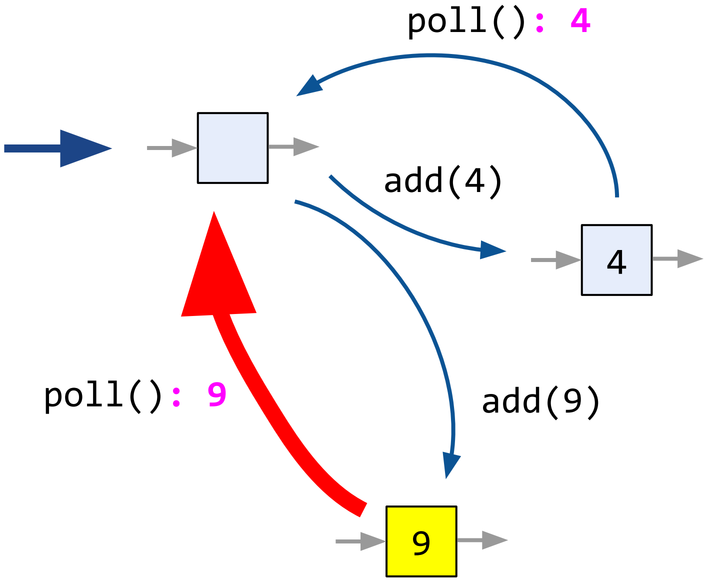
```
      val q = MSQueue<Int>()
-----------------------------------
q.add(4)         | q.poll(): 4
q.poll(): 9      | q.add(9)
```

# LTS-based verification



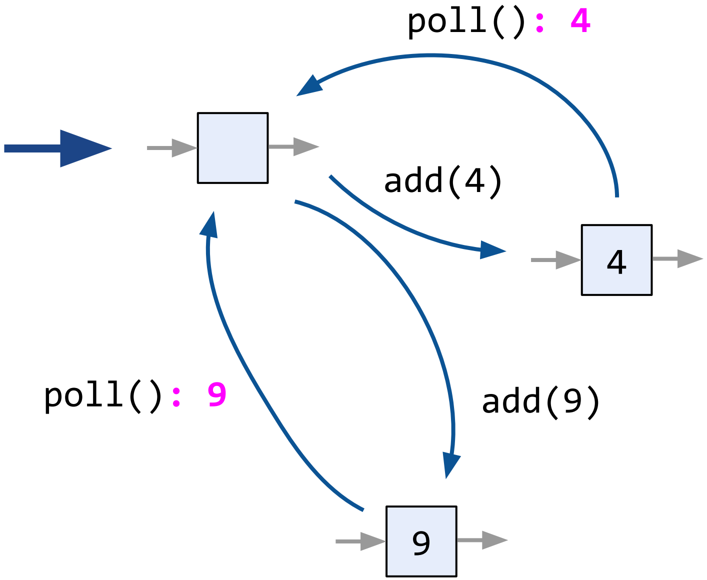
```
      val q = MSQueue<Int>()
---------------------------------------
q.add(4)        | q.poll(): 4
q.poll(): 9     | q.add(9)
```

# LTS-based verification



```
       val q = MSQueue<Int>()
-------------------------------------
q.add(4)          | q.poll(): 4
q.poll(): 9       | q.add(9)
```

# LTS-based verification



```
        val q = MSQueue<Int>()
q.add(4)          | q.poll(): 4
q.poll(): 9       | q.add(9)
```

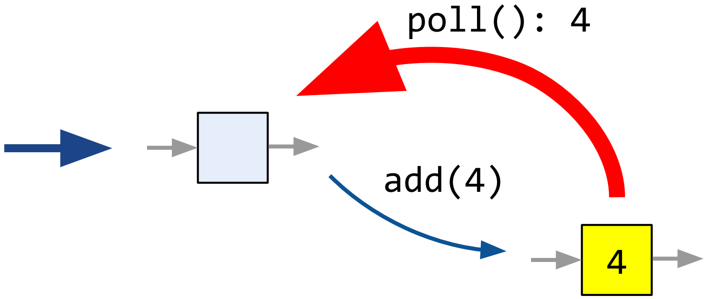
A path is found ⇒ correct

# Lazy LTS creation

- We build LTS lazilly, like on the previous slides
- We use sequential implementation

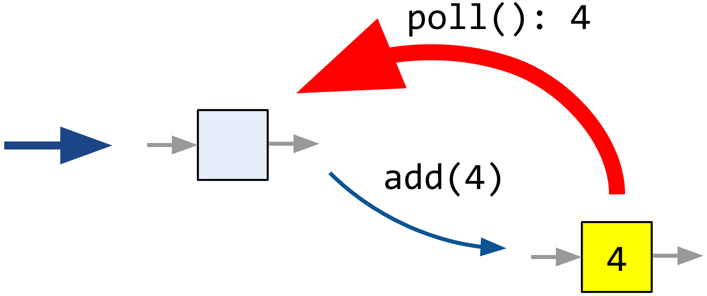
# Lazy LTS creation

- We build LTS lazilly, like on the previous slides
- We use sequential implementation



# Lazy LTS creation

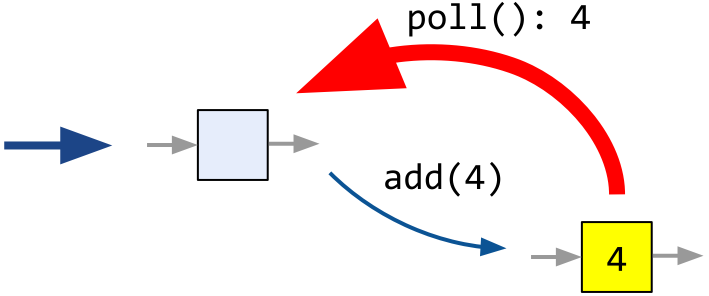
- We build LTS lazilly, like on the previous slides
- We use sequential implementation
- Equivalence via `equals/hashcode` implementations



```
class MSQueueTest {
    val q = MSQueue<Int>()

    // Operations here

    override fun equals(other: Any?) = ...
    override fun hashCode() = ...
}
```

# Lazy LTS creation

- We build LTS lazilly, like on the previous slides
- We use sequential implementation
- Equivalence via `equals`/`hashcode` implementations



```
class MSQueueTest: VerifierState() {
    val q = MSQueue<Int>()

    // Operations here

    override fun generateState() = q
}
```

# What if our data structure is blocking by design?

```
        val c = Channel<Int>()
c.send(4)           c.receive() // 4
```

`send` waits for `receive` and vice versa

Producer 1

```
val elem = ...
 c.send(elem)
```

Consumer

```
while(true) {
   val elem = c.receive()
   process(elem)
}
```

Producer 2

```
val elem = ...
 c.send(elem)
```

```
val c = Channel()
```

Producer 1

```
val elem = ...
 c.send(elem)
```

Producer 2

```
val elem = ...
 c.send(elem)
```

Has to wait for send

Consumer

```
while(true) {
1   val elem = c.receive()
    process(elem)
}
```

```
val c = Channel()
```

Producer 1

```
val elem = ...
c.send(elem)
```

Producer 2

```
val elem = ...
c.send(elem)
```

Consumer

```
  while(true) {
1   val elem = c.receive()
    process(elem)
  }
```

```
val c = Channel()
```

Producer 1

```
val elem = ...
c.send(elem)
```

Consumer

```
while(true) {
1   val elem = c.receive()
    process(elem)
}
```

Producer 2

```
val elem = ...
c.send(elem)
```

```
val c = Channel()
```

Rendezvous!

Producer 1

```
    val elem = ...
 2  c.send(elem)
```

Consumer

```
    while(true) {
 1      val elem = c.receive()
        process(elem)
    }
```

Producer 2

```
    val elem = ...
    c.send(elem)
```

```
val c = Channel()
```

Producer 1

```
    val elem = ...
2   c.send(elem)
```

Producer 2

```
    val elem = ...
    c.send(elem)
```

Consumer

```
  while(true) {
1     val elem = c.receive()
3     process(elem)
  }
```

```
val c = Channel()
```

Producer 1

```
    val elem = ...
2   c.send(elem)
```

Producer 2

```
    val elem = ...
    c.send(elem)
```

Consumer

```
  while(true) {
1   val elem = c.receive()
3   process(elem)
  }
```

```
val c = Channel()
```

Producer 1

```
   val elem = ...
2 c.send(elem)
```

Consumer

```
  while(true) {
1     val elem = c.receive()
3     process(elem)
  }
```

Producer 2

```
   val elem = ...
4  c.send(elem)
```

Has to wait for receive

```
val c = Channel()
```

## Producer 1

```
    val elem = ...
2   c.send(elem)
```

## Consumer

```
  while(true) {
1     val elem = c.receive()
3     process(elem)
  }
```

## Producer 2

```
    val elem = ...
4   c.send(elem)
```

```
val c = Channel()
```

## Producer 1

```
    val elem = ...
2  c.send(elem)
```

## Consumer

```
    while(true) {
5 1   val elem = c.receive()
  3   process(elem)
    }
```

## Producer 2

```
    val elem = ...
4  c.send(elem)
```

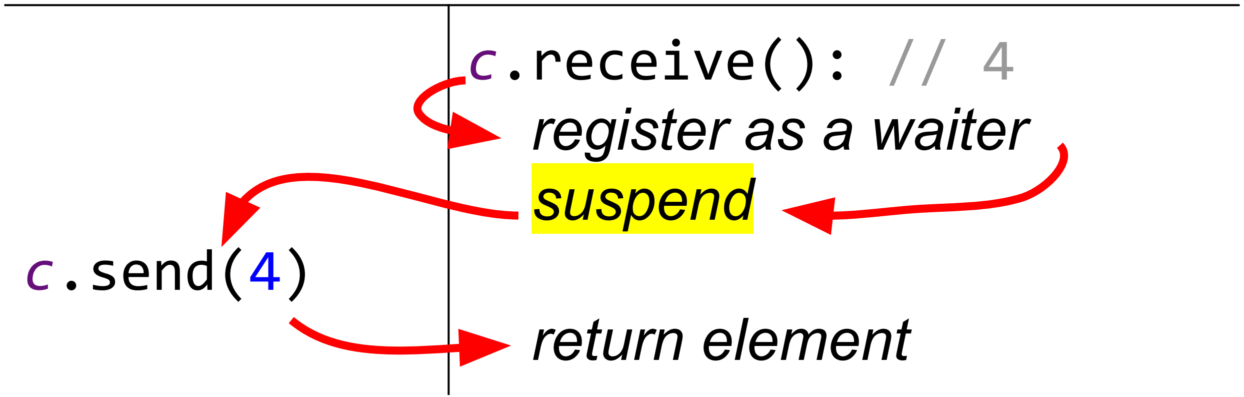
Has to wait for receive

```
val c = Channel()
```

```
         val c = Channel<Int>()
-------------------------------------------
c.send(4)         | c.receive() // 4
```

Non-linearizable
because of suspension

```
val c = Channel<Int>()
```



```
c.receive(): // 4
```

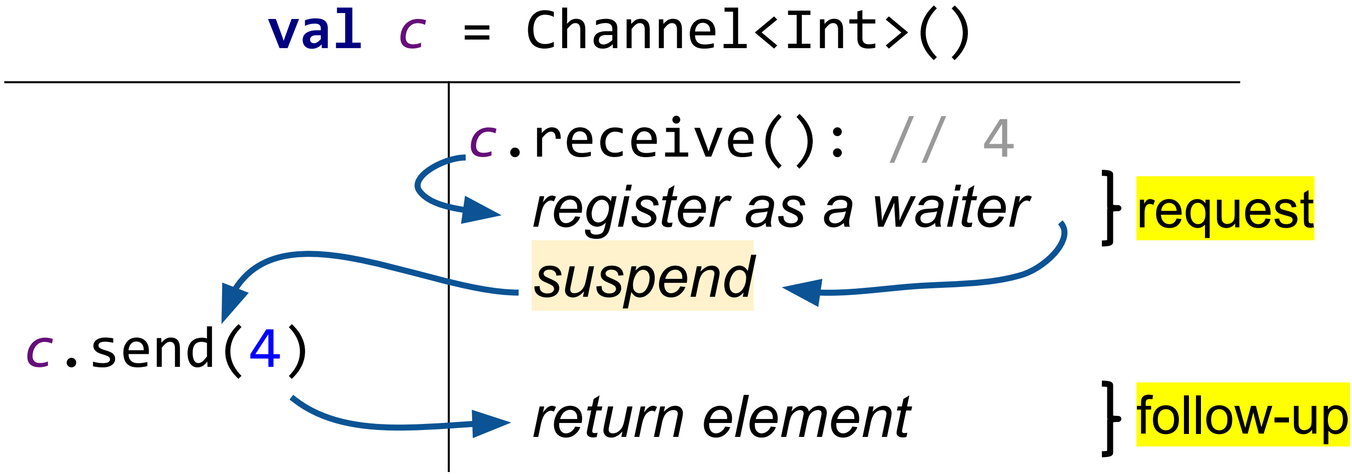
*register as a waiter*

*suspend*

```
c.send(4)
```

*return element*

# Dual Data Structures [1]



```
val c = Channel<Int>()
```

```
c.receive(): // 4
```

*register as a waiter*

*suspend*

```
c.send(4)
```

*return element*

request

follow-up

[1] “Nonblocking Concurrent Data Structures with Condition Synchronization” by Scherer, W.N. and Scott, M.L.

# Dual Data Structures

```
val c = Channel<Int>()
c.receive^REQ(): tik
c.send(4)
c.receive^FUP(tik): 4
```

Unique ticket, $\in \mathbb{N}$

# Dual Data Structures

`val c = Channel<Int>()`<br>
`c.receive`$^{\text{REQ}}$`(): tik`<br>
`c.send(4)`<br>
`c.receive`$^{\text{FUP}}$`(tik): 4`

Follow-ups should be invoked ***after*** the corresponding requests

# Dual Data Structures

```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

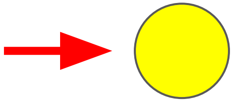
Let’s always pass tickets,
for simplicity

# Dual Data Structures

```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

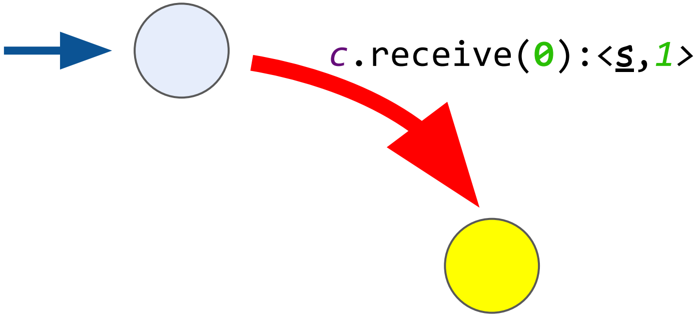
suspended

# LTS for Dual Data Structures



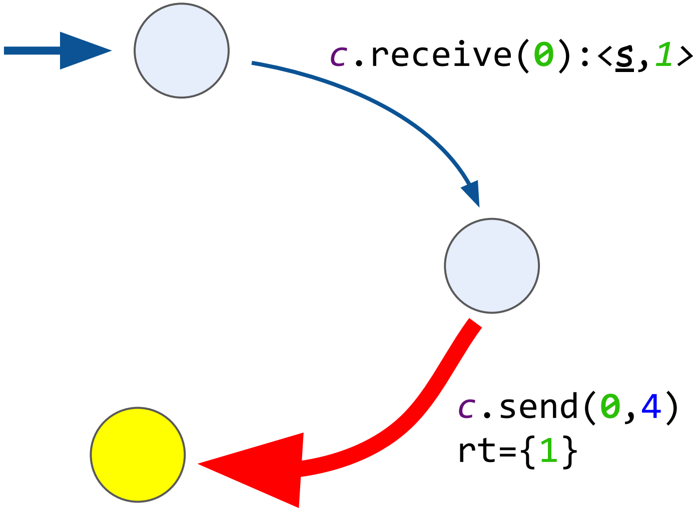
```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

# LTS for Dual Data Structures



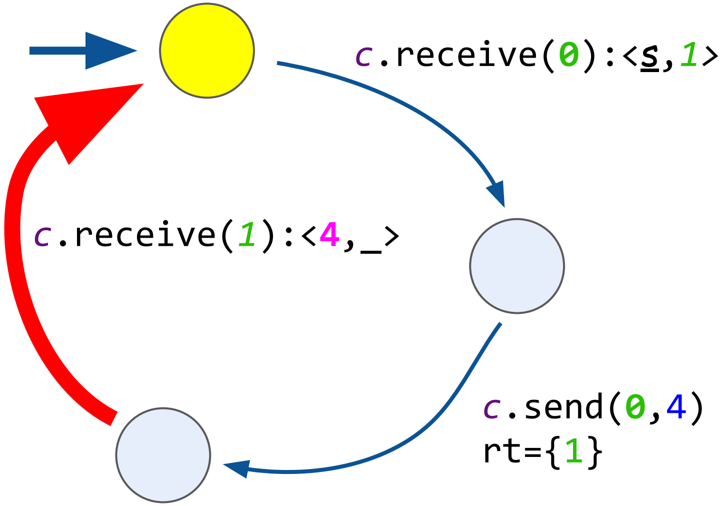
```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

# LTS for Dual Data Structures



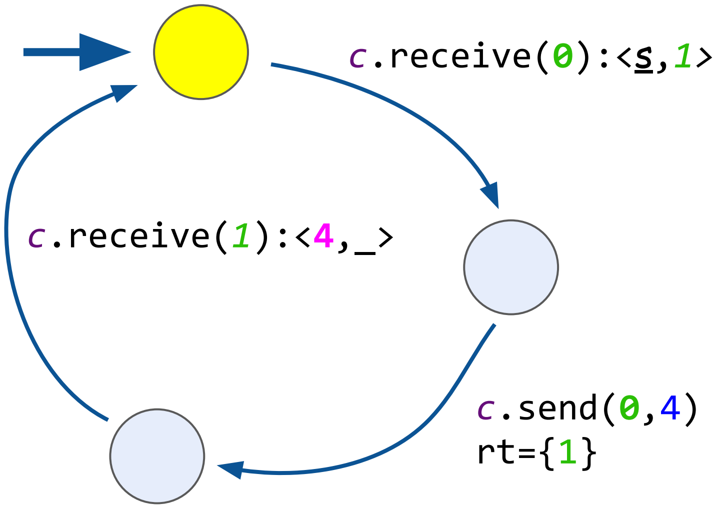
```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

# LTS for Dual Data Structures



```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

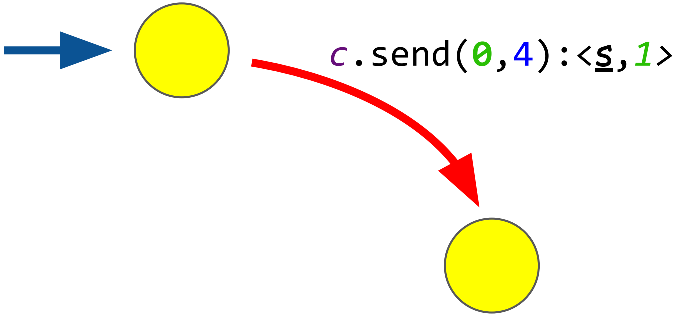
# LTS for Dual Data Structures



```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

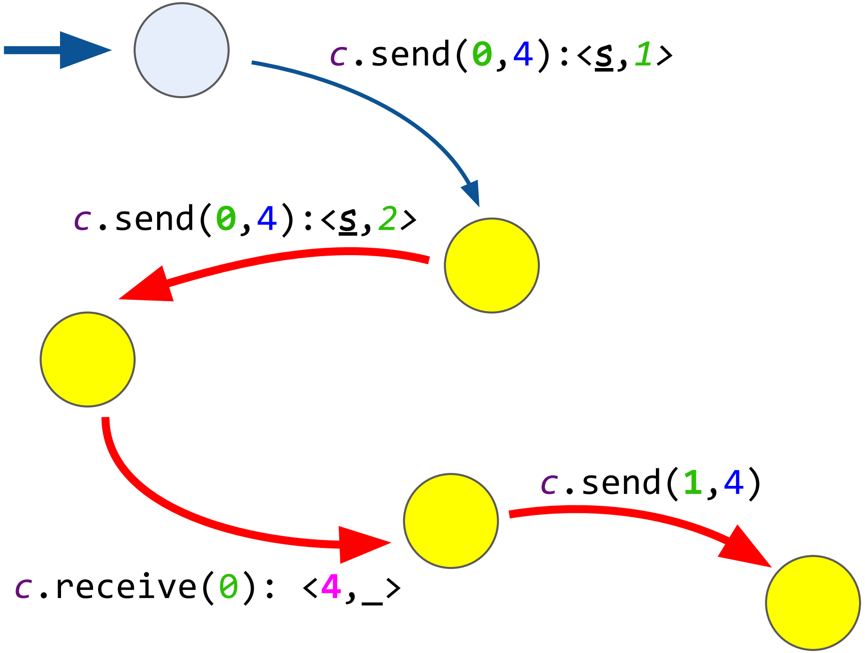
Looks similar

# LTS for Dual Data Structures



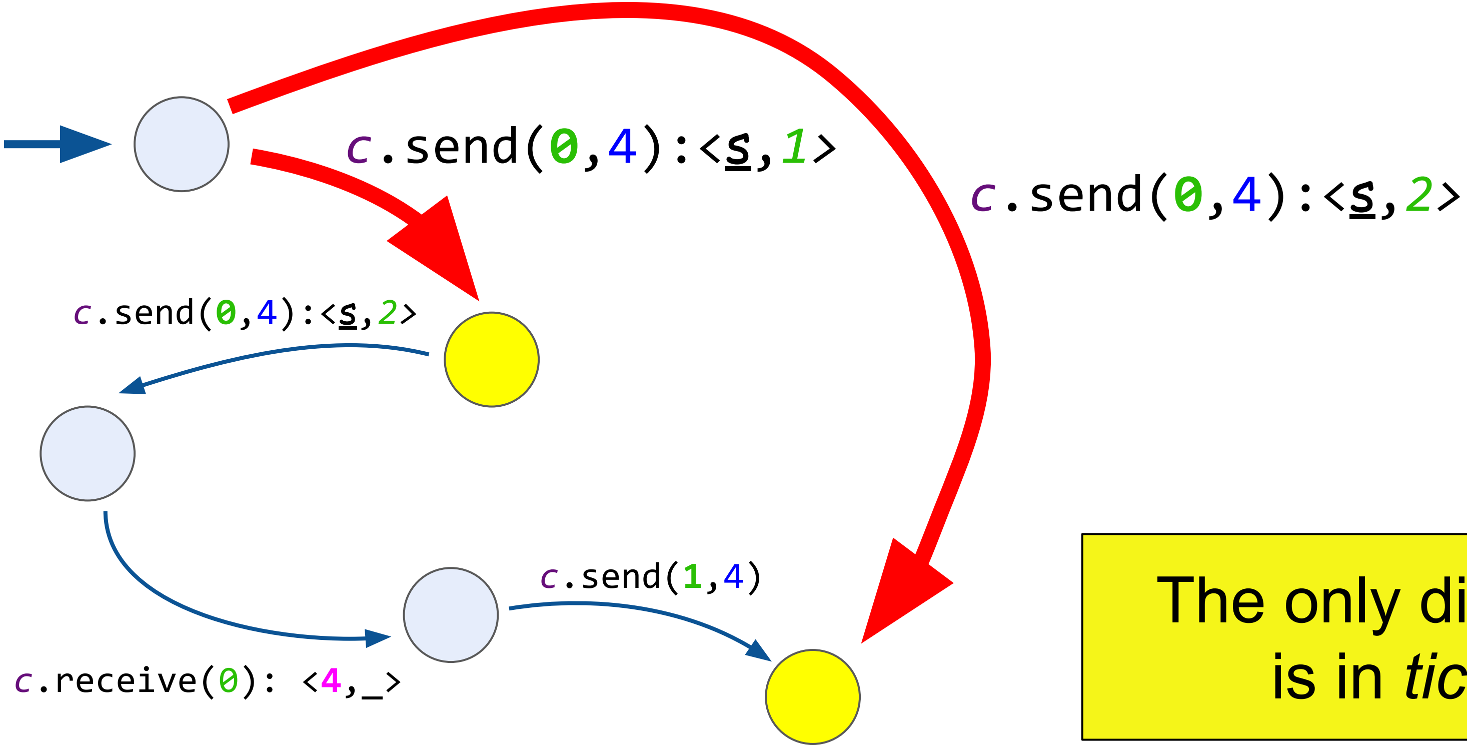
```
val c = Channel<Int>()
c.send(0, 4): <s,1>
```

# LTS for Dual Data Structures



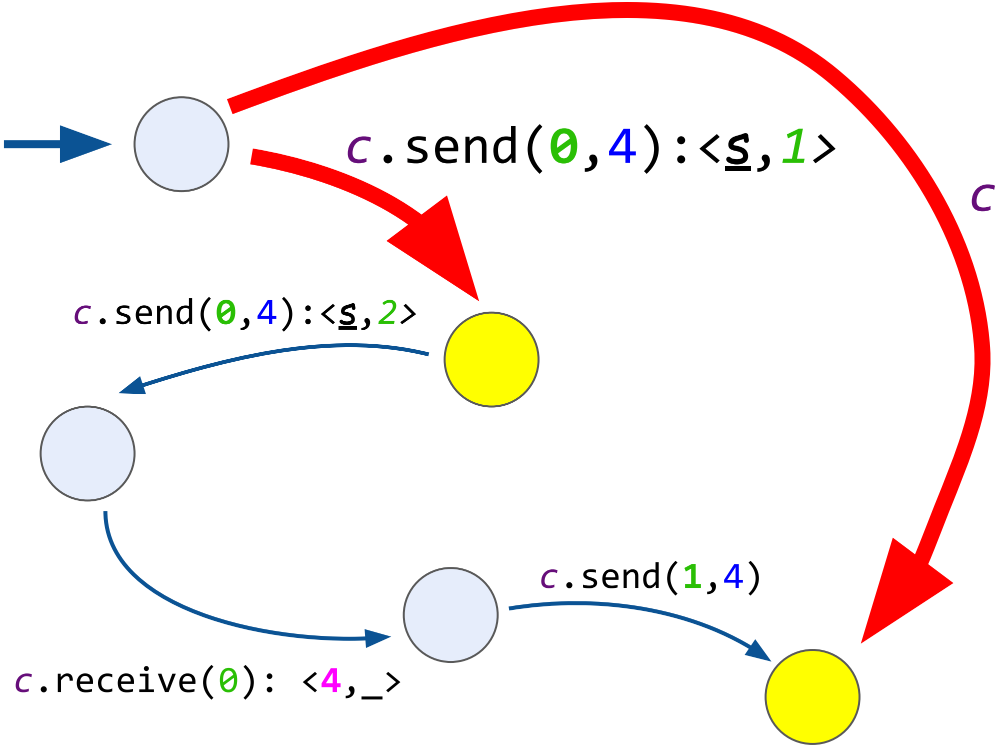
```
val c = Channel<Int>()
c.send(0, 4): <s,1>
c.send(0, 4): <s,2>
c.receive(0): <4,_>
c.send(1, 4)
```

# LTS for Dual Data Structures



The only difference
is in *tickets*

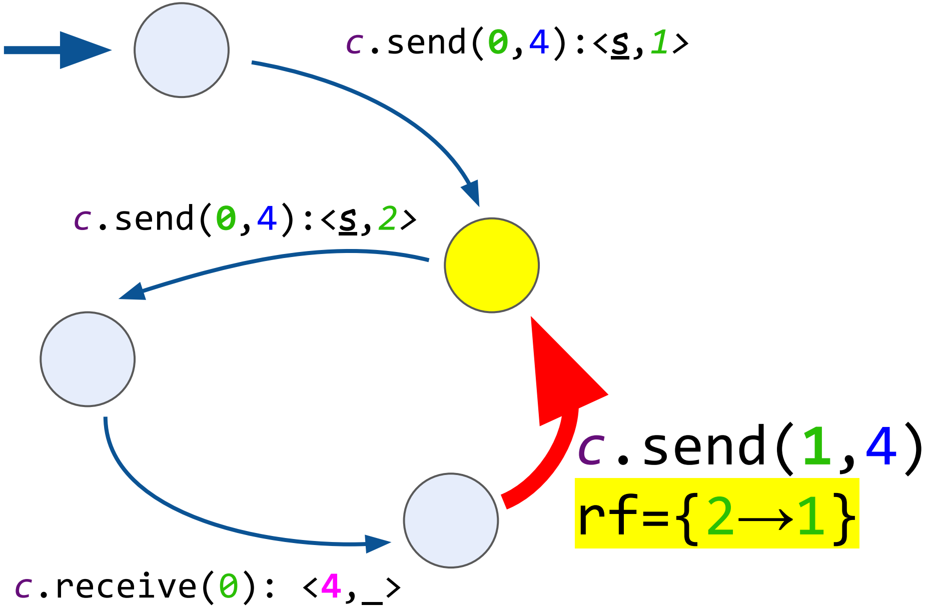
# LTS for Dual Data Structures



Let’s forbid such duplicate transitions

The only difference is in *tickets*

# LTS for Dual Data Structures

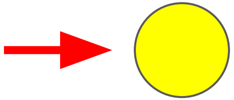


```
val c = Channel<Int>()
c.send(0, 4): <s,1>
c.send(0, 4): <s,2>
c.receive(0): <4,_>
c.send(1, 4)
```

# Verifier for Dual Data Structures

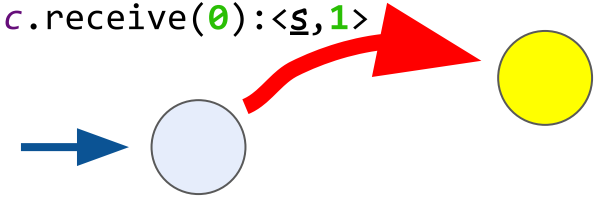
```
          val c = Channel<Int>()

c.receive(): 4     | c.send(4): s+Unit
c.receive(): s     |
```

# Verifier for Dual Data Structures



```
                        val c = Channel<Int>()
-------------------------------------------------------------
c.receive(): 4          | c.send(4): s+Unit
c.receive(): s          |
```

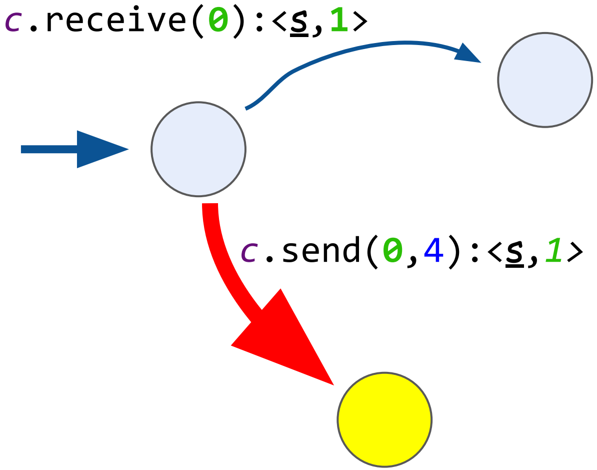
# Verifier for Dual Data Structures



```
                       val c = Channel<Int>()
-----------------------------------------------------------
c.receive(): 4          | c.send(4): s+Unit
c.receive(): s          |
```

Results are different

# Verifier for Dual Data Structures

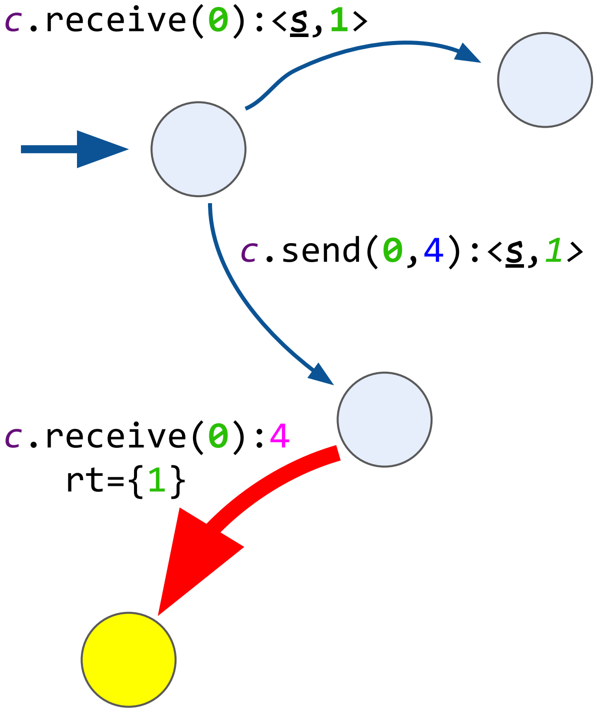


```
val c = Channel<Int>()
```

| c.receive(): 4 | c.send(4): s+Unit |
|---|---|
| c.receive(): s | |

suspended, ticket 1

# Verifier for Dual Data Structures

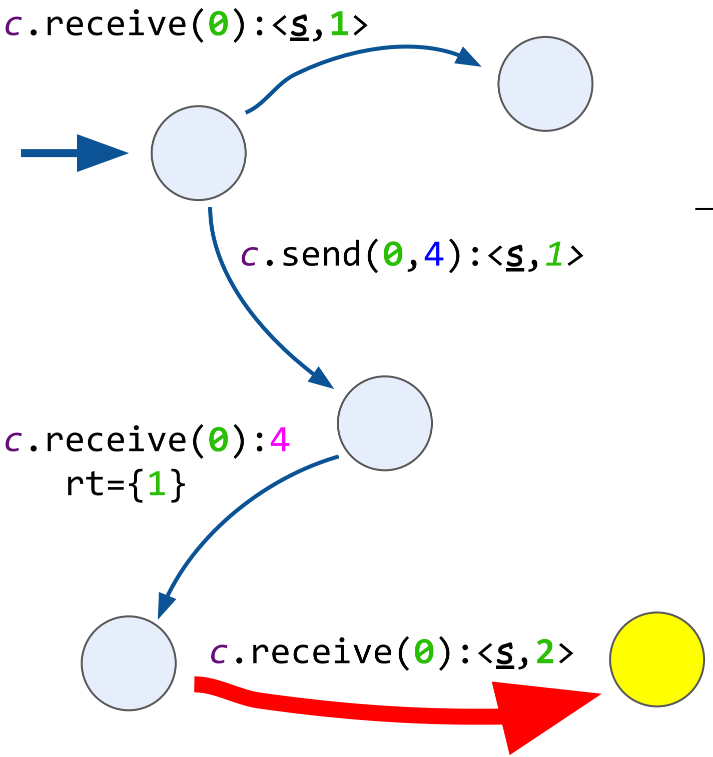


`val c = Channel<Int>()`

| | |
|---|---|
| `c.receive(): 4` | `c.send(4): s+Unit` |
| `c.receive(): s` | |

~~suspended~~, ticket 1
resumed

# Verifier for Dual Data Structures

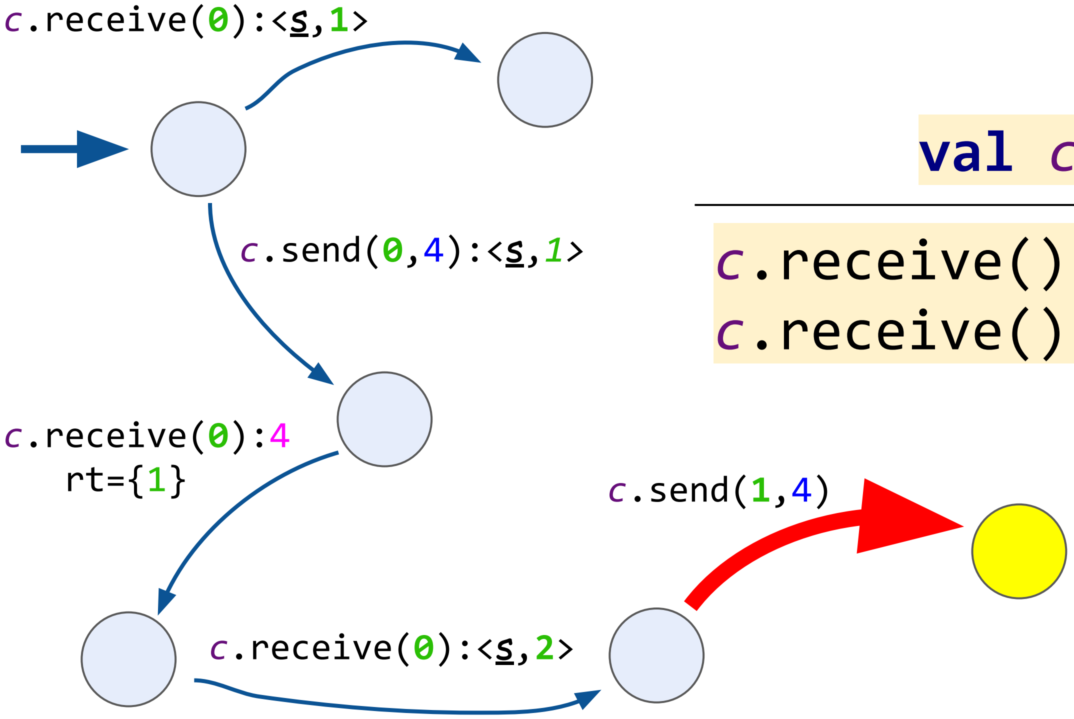


```
val c = Channel<Int>()
```

| | |
|---|---|
| c.receive(): 4<br>c.receive(): s | c.send(4): s+Unit |

~~suspended~~, ticket *1*
resumed

# Verifier for Dual Data Structures

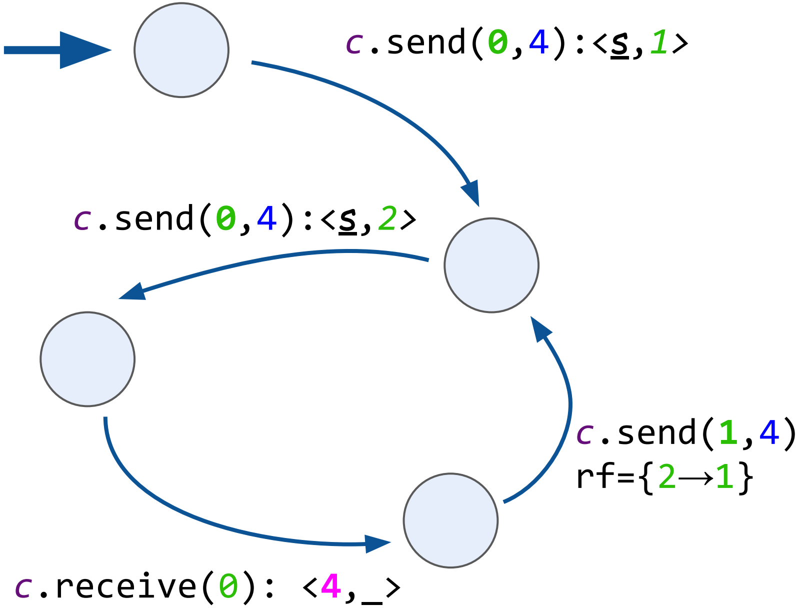


```
val c = Channel<Int>()
```

```
c.receive(): 4        c.send(4): s+Unit
c.receive(): s
```

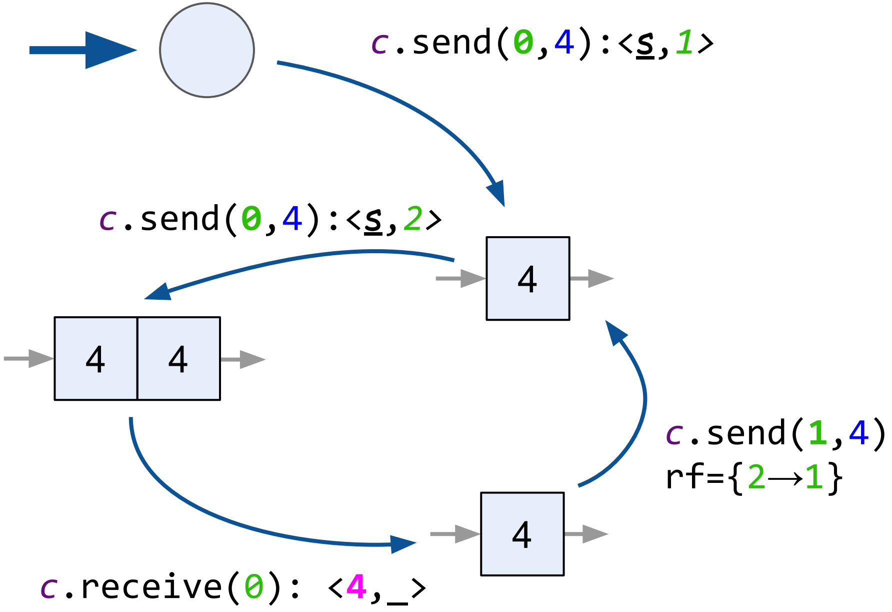
~~suspended~~, ticket 1
~~resumed~~

# Lazy Dual Data Structures LTS creation



```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

# Lazy Dual Data Structures LTS creation



c.send(0,4):<s,1>

c.send(0,4):<s,2>

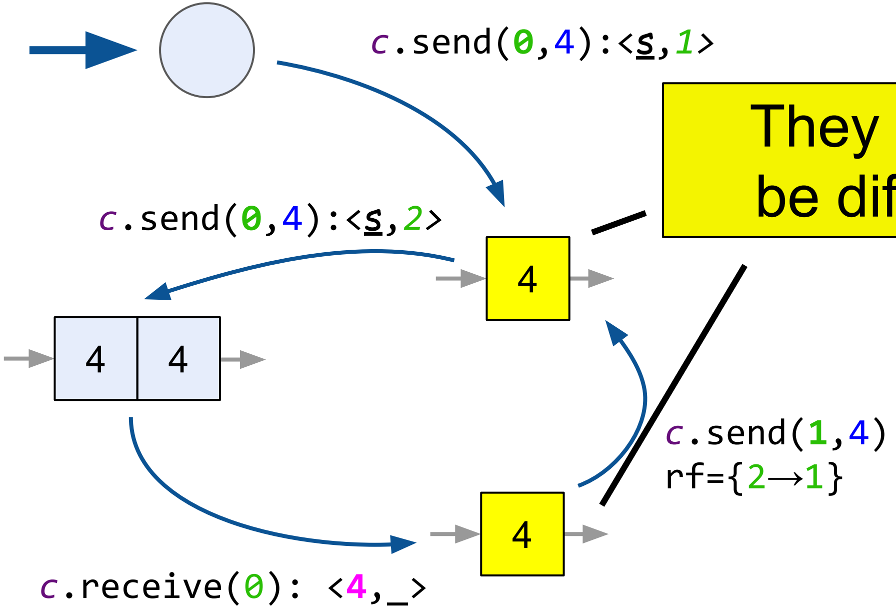
c.send(1,4)
rf={2→1}

c.receive(0): <4,_>

```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

Let’s define as an externally observable state as before

# Lazy Dual Data Structures LTS creation

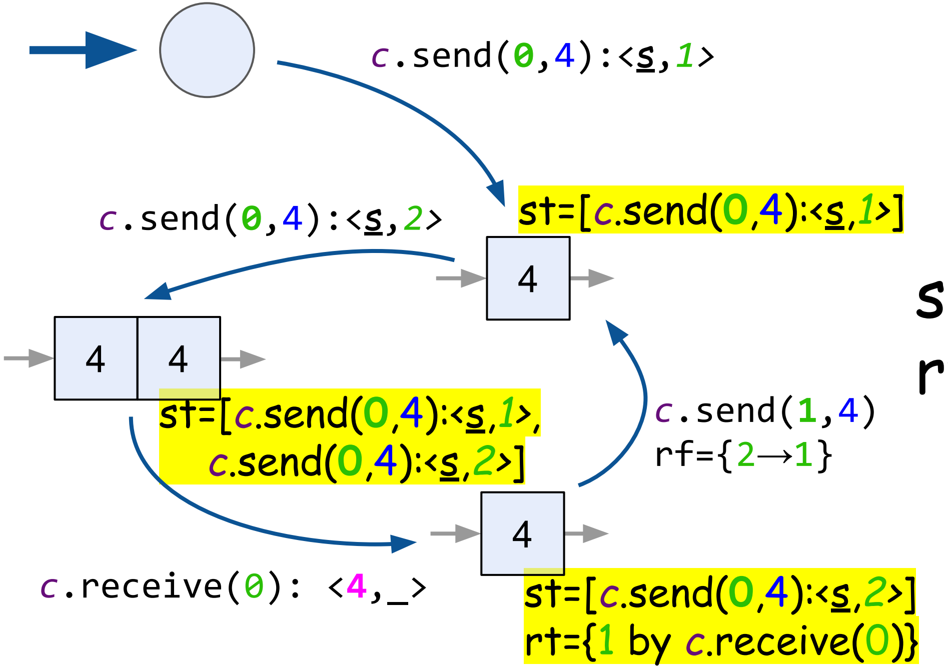


They should be different!

```
val c = Channel<Int>()
c.receive(0): <s,1>
c.send(0, 4)
c.receive(1): <4,_>
```

Let’s define as an externally observable state as before

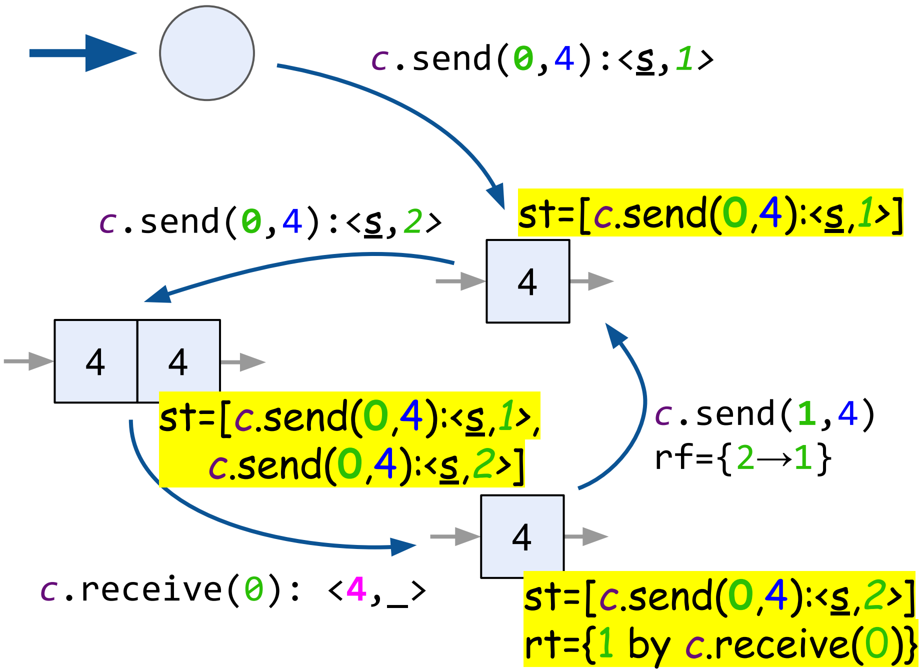
# Lazy Dual Data Structures LTS creation



st = list of suspended operations

rt = set of resumed operations

# Lazy Dual Data Structures LTS creation

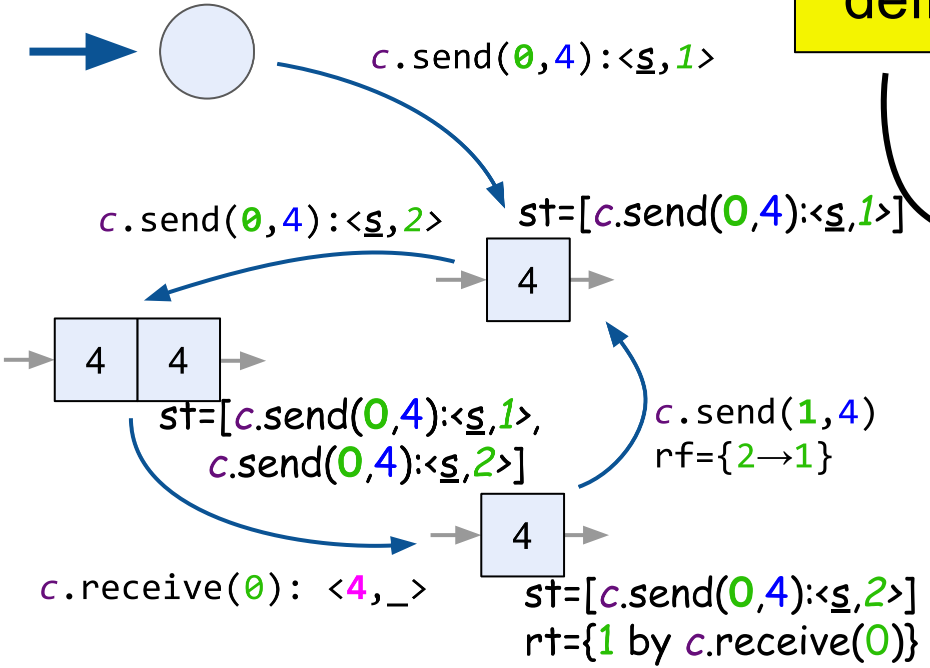


States are equal iff ∃`f:ℕ→ℕ` that

1. externally observable states
2. `st`-s wrt `rf` on tickets (as lists)
3. `rt`-s wrt `rf` on tickets (as sets)

are equal

# Lazy Dual Data Structures LTS creation



c.send(0,4):<s,1>

st=[c.send(0,4):<s,1>]

c.send(0,4):<s,2>

st=[c.send(0,4):<s,1>,
c.send(0,4):<s,2>]

c.send(1,4)
rf={2→1}

c.receive(0): <4,_>

st=[c.send(0,4):<s,2>]
rt={1 by c.receive(0)}

defined via `equals/hashcode`

States are equal iff ∃ `f:ℕ→ℕ` that

1. externally observable states
2. `st`-s wrt `rf` on tickets (as lists)
3. `rt`-s wrt `rf` on tickets (as sets)

are equal

maintained by Lin-Check

# Channel Test Example

```
class RendezvousChannelTest: LinCheckState() {
   val c = Channel()

   @Operation suspend fun send(x: Int) = c.send(x)
   @Operation suspend fun receive(): Int = c.receive()

   override fun generateState() = Unit
}
```

# Channel Test Example

```
class BufferedChannelTest: LinCheckState() {
    val c = Channel()

    @Operation suspend fun send(x: Int) = c.send(x)
    @Operation suspend fun receive(): Int = c.receive()

    override fun generateState(): Any {
        val state = ArrayList<Int>()
        var x: Int?
        while(true) {
            x = c.poll()
            if (x == null) break
            state += x
        }
        return state
    }
}
```

# Uncovered topics

- Verifiers for several relaxed contracts
- How to run scenarios in the most “dangerous” way
- API

# Future plans

- Smart running strategies
- Supporting randomized relaxed contracts

# Questions?

https://github.com/Kotlin/kotlinx-lincheck