Hoare-style Specifications as Correctness Conditions for Non-Linearizable Concurrent Objects

Ilya Sergey

Aleks Nanevski
Anindya Banerjee
Germán Andrés Delbianco

OOPSLA 2016
November 2, 2016

Linearizable Concurrent Objects

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Non-overlapping calls to methods of a *concurrent* object should appear to take effect in their *sequential* order.

Linearizability is expensive

Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated

Hagit Attiya
Technion
hagit@cs.technion.il

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Danny Hendler
Ben-Gurion University
hendlerd@cs.bgu.ac.il

Petr Kuznetsov
TU Berlin/Deutsche Telekom Labs
pkuznets@acm.org

Maged M. Michael
IBM T. J. Watson Research Center
magedm@us.ibm.com

Martin Vechev
IBM T. J. Watson Research Center
mtvechev@us.ibm.com

Enabling better parallelism

The advent of multicore processors as the standard computing platform will force major changes in software design.

BY NIR SHAVIT

Data Structures in the Multicore Age

Relaxing the correctness condition would allow one to implement concurrent data structures more efficiently, as they would be free of synchronization bottlenecks.

Alternatives to linearizability

- Quiescent Consistency [Aspnes-al:JACM94]
- Quasi-Linearizability [Afek-al:OPODIS10]
- Quantitative Relaxation [Henzinger-al:POPL13]
- Quantitative Quiescent Consistency [Jagadeesan-Riely:ICALP14]
- Concurrency-Aware Linearizability [Hemed-Rinetzky:DISC15]
- Local Linearizability [Haas-al:CONCUR16]
- …

Challenges of diversity

- *Composing* different conditions (CAL, QC, QQC) in a single program, which uses multiple objects;
- Providing *syntactic* proof methods for establishing all these conditions (akin to *linearization points*);
- Employing these criteria for *client-side reasoning* (*uniformity*).

Hoare-style Specifications as Correctness Conditions for Non-Linearizable Concurrent Objects

Ilya Sergey

Aleks Nanevski
Anindya Banerjee
Germán Andrés Delbianco

OOPSLA 2016
November 2, 2016

Hoare-style Specifications

$\{ P \}$ **e** $\{ Q \}$ @ C

precondition

postcondition

concurrent invariants and protocol

If the initial *state* satisfies P, then, after **e** terminates, the final *state* satisfies Q (no matter the *interference* manifested by **C**).

Hoare-style Specifications

$$\{\, P \,\}\ \mathbf{e}\ \{\, Q \,\} @ C$$

- *Compositional* — substitution principle;
- *Syntactic proof method* — inference rules;
- *Uniform* — reasoning about objects and their clients in the *same* proof system.

Hoare-style Specifications

$$\{\, P \,\}\ \mathbf{e}\ \{\, Q \,\} @\, C$$

Rich	• *Compositional* — substitution principle;
Live	• *Syntactic proof method* — inference rules;
Two-sided	• *Uniform* — reasoning about objects and their clients in the *same* proof system.

This work: Hoare-style specs as CAL, QC, QQC

Concurrency-Aware Linearizability (CAL):

Effects of some concurrent method calls should appear to happen simultaneously.

Quiescent Consistency (QC):

Method calls separated by a period of ***no interference*** *(quiescence) should appear to take effect in their order.*

This talk

Quantitative Quiescent Consistency (QQC):

The number of out-of-order method results is bounded by the number of interfering threads (with a constant factor).

Simple Counting Network

```
def getAndInc() : nat
```

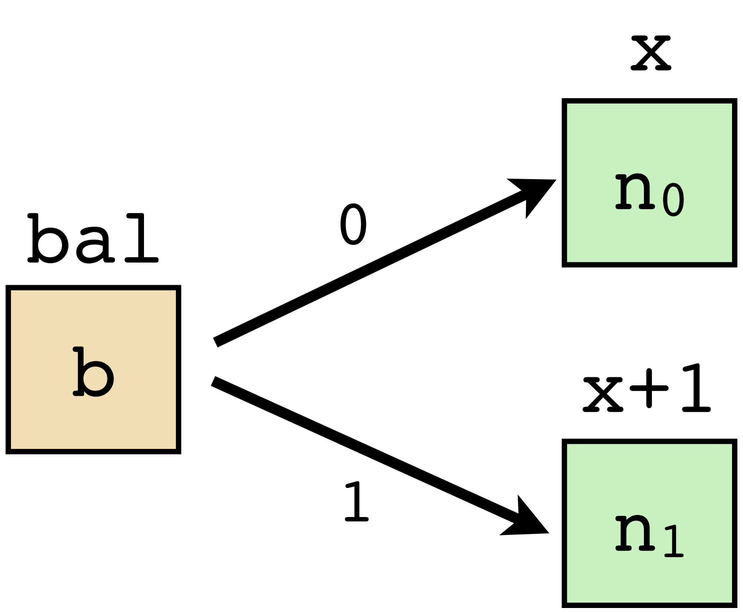
Simple Counting Network

```
def getAndInc() : nat = {
    n ← &x;
    b ← CAS(x, n, n + 1);
    if b then
      return n;
    else getAndInc();
}
```

high contention location

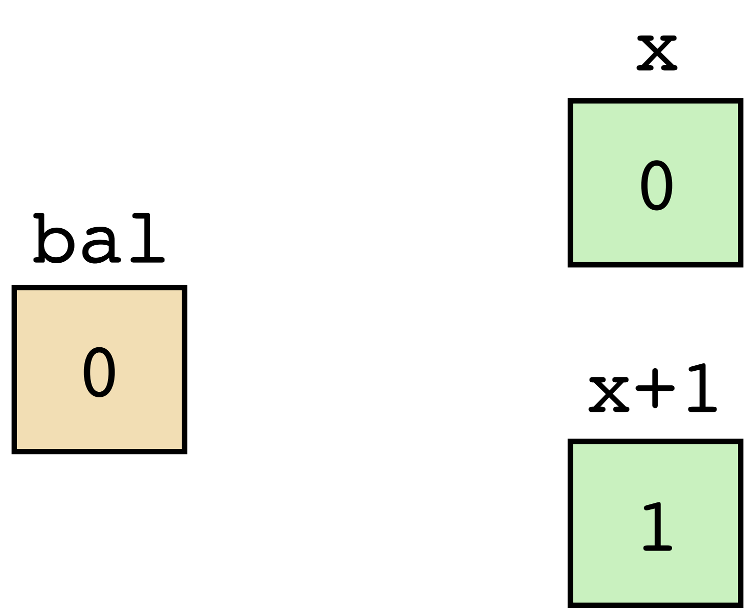
Simple Counting Network

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



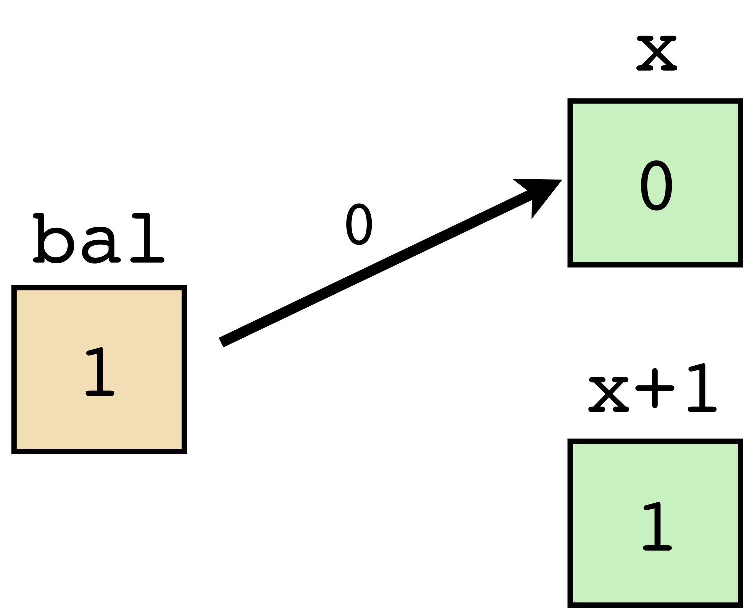
Sequential Execution (T_1)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



Sequential Execution (T_1)

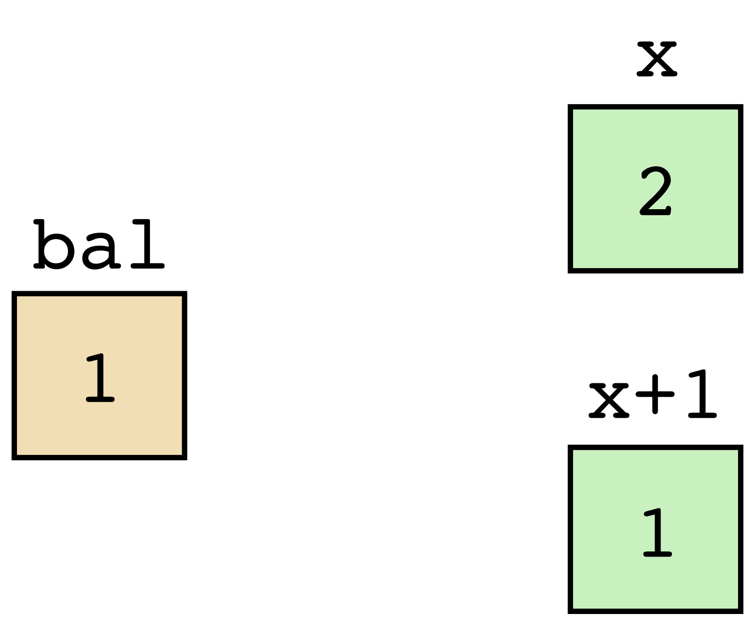
```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



$T_1.b_1 = 0$

Sequential Execution (T_1)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```

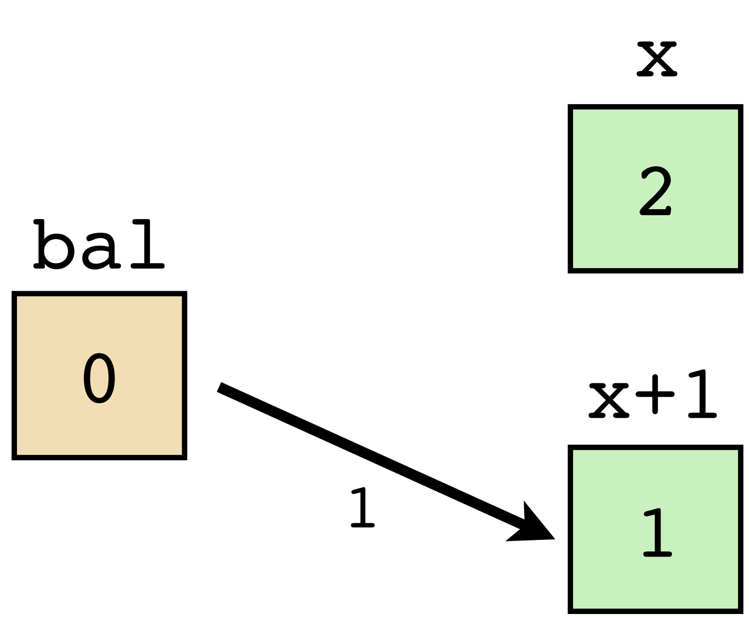


$T_1.b_1 = 0$

$T_1.res_1 = 0$

Sequential Execution (T_1)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



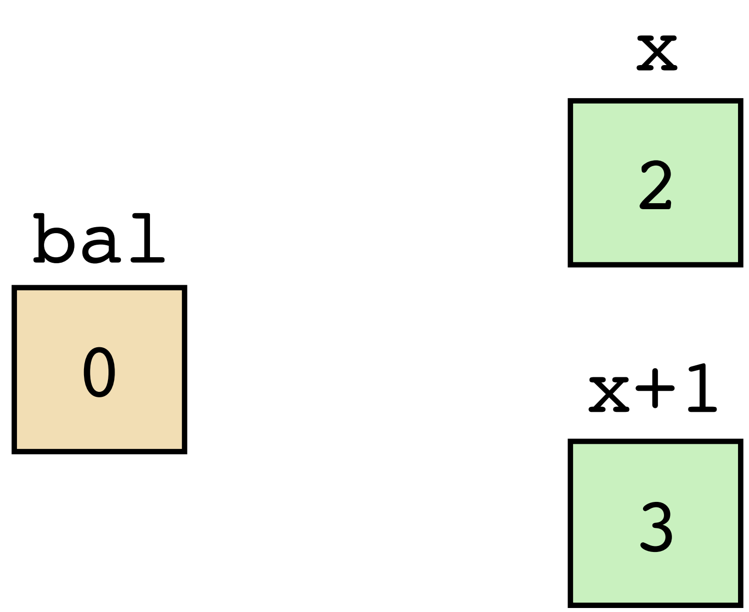
$T_1.b_1 = 0$

$T_1.res_1 = 0$

$T_1.b_2 = 1$

Sequential Execution (T_1)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



$T_1.b_1 = 0$

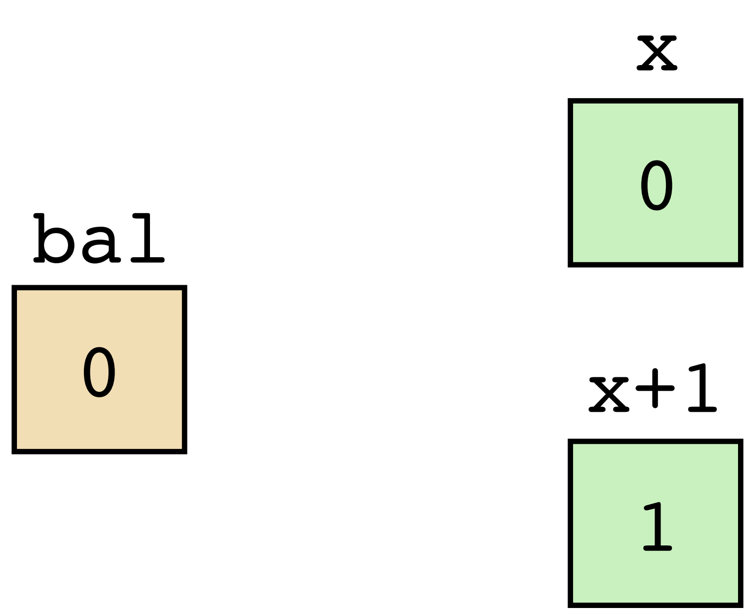
$T_1.res_1 = 0$

$T_1.b_2 = 1$

$T_1.res_2 = 1$

Concurrent Execution (T_1, T_2)

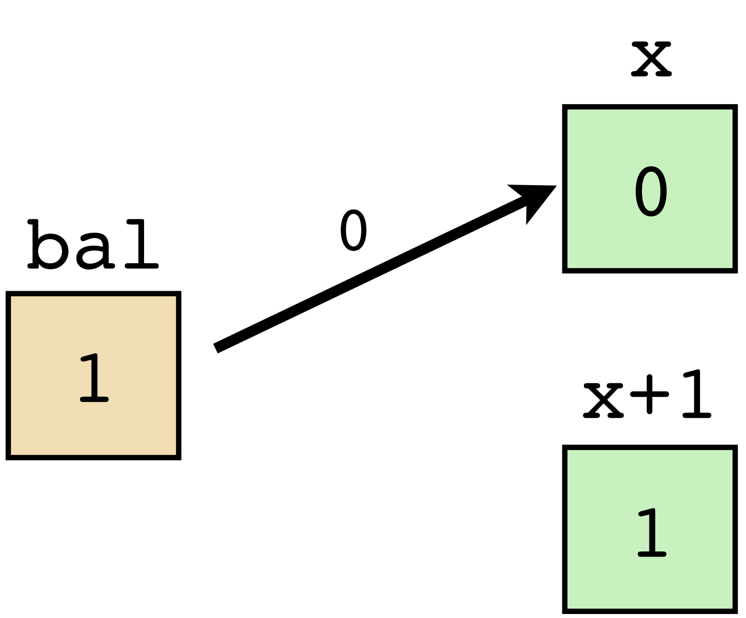
```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



Concurrent Execution (T_1, T_2)

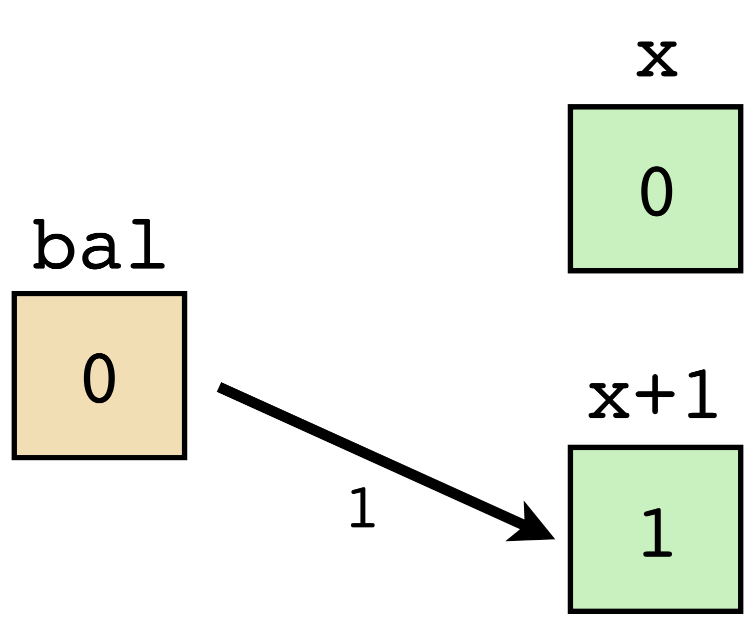
```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```

$T_1.b_1 = 0$



Concurrent Execution (T_1, T_2)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



$T_1.b_1 = 0$

$T_2.b_1 = 1$

Concurrent Execution (T_1, T_2)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```

x

0

bal

0

x+1

3

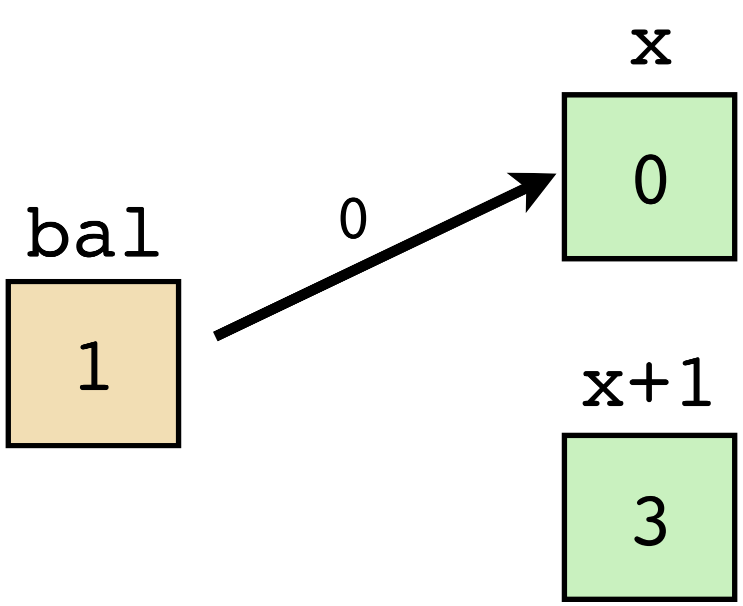
$T_1.b_1 = 0$

$T_2.b_1 = 1$

$T_2.res_1 = 1$

Concurrent Execution (T_1, T_2)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



$T_1.b_1 = 0$

$T_2.b_1 = 1$

$T_2.res_1 = 1$

$T_2.b_2 = 0$

Concurrent Execution (T_1, T_2)

```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```

bal: 1

x: 2

x+1: 3

$T_1.b_1 = 0$

$T_2.b_1 = 1$

$\boxed{T_2.res_1 = 1}$

$T_2.b_2 = 0$

$\boxed{T_2.res_2 = 0}$

Correctness Conditions for Counting Network

- ~~**R_0**: calls to `getAndInc()` take effect in their sequential order~~
- **R_1**: different calls return *distinct* results (**strong concurrent counter**)
- **R_2**: two calls, separated by *period of quiescence*, take effect in their sequential order (**QC**)
- **R_3**: results of *two calls* in the same thread are out of order by no more than *2 ** (number of calls *interfering with both*) (**QQC**)

Observations about the Counting Network

“Tokens”

- Every flip of the balancer grants thread a *capability* to add 2 to a counter (`x` or `x+1`);

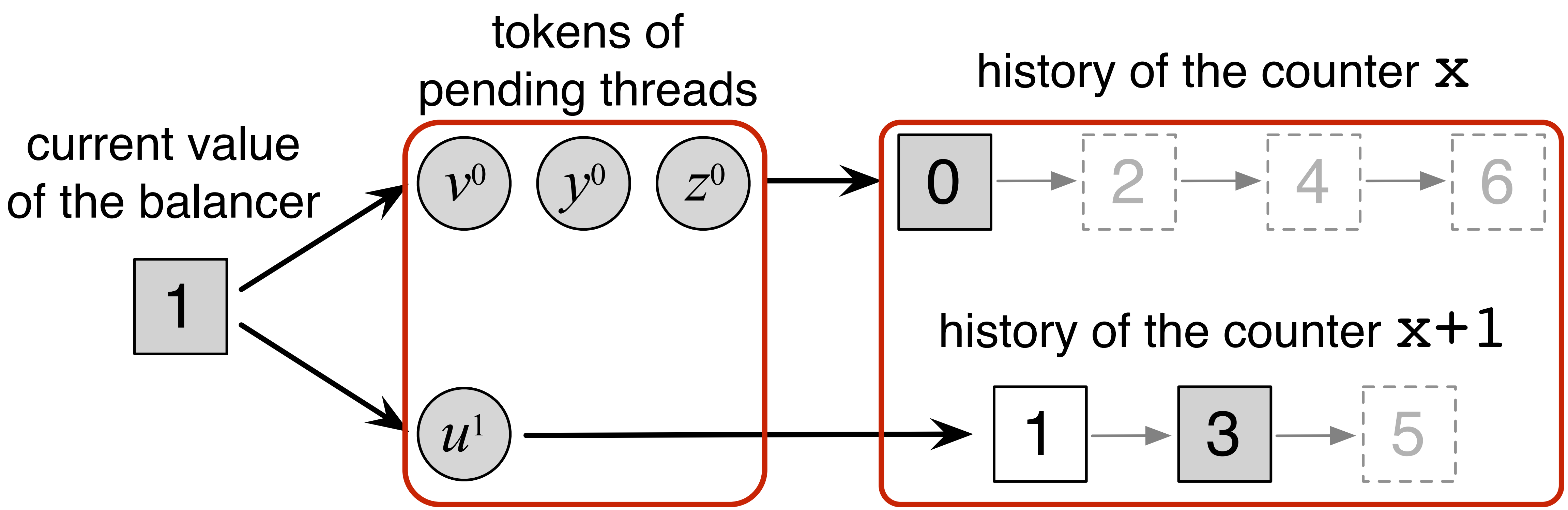
- Each of the counters (`x` and `x+1`) changes *continuously* wrt. *even/odd* values

“Histories”

Real and Auxiliary State

- Hoare-style specs constrain **state**, *auxiliary* or *real*
- **Real state** — *heap* (pointers **`bal`**, **`x`**, **`x+1`**);
- **Auxiliary state** — any *fictional splittable* resource:
  - ✦ *Token sets* (τ) — disjoint sets;
  - ✦ *Histories* (χ) — partial maps with **`nat`** as domain.

Auxiliary State of the Network



Tokens =
pending updates

Histories =
observed updates

Interference-capturing histories

$$X = \{\ \ldots,\ \boxed{n} \mapsto \iota,\ \ldots\ \}$$

"timestamp", a value written to a counter **`x`** or **`x+1`** (0, 1, 2, etc.)

Interference-capturing histories

$$\chi = \{\ \ldots,\ n \mapsto \boxed{\iota},\ \ldots\ \}$$

sets of tokens, held by interfering threads
at the moment the entry has been written

Notation for *Subjective* Histories and Tokens

- $\boldsymbol{\chi}$, χ — histories, contributed by **this** and *other* threads;

- $\boldsymbol{\tau}$, τ — tokens, held by **this** and *other* threads

Specification of `getAndInc()`

$$\{\ \boldsymbol{\tau} = \varnothing\ \}$$

```
res ← getAndInc()
```

$$\{\exists\ \iota,\ \ \boldsymbol{\tau}' = \varnothing,$$
$$\boldsymbol{\chi}' = \boldsymbol{\chi} \cup (\mathrm{res} + 2) \mapsto \iota,$$
$$\tau \subseteq \tau' \cup \mathrm{spent}(\chi' \setminus \chi),$$
$$\mathrm{last}(\boldsymbol{\chi} \cup \chi) < \mathrm{res} + 2 + 2\,|\,\iota \cap \tau\,|\}$$

Specification of `getAndInc()`

$$\boxed{\{\ \boldsymbol{\tau} = \varnothing\ \}}$$

no tokens held initially by this thread

`res ← getAndInc()`

$$\{\exists\ \iota,\ \ \boldsymbol{\tau}' = \varnothing,$$
$$\boldsymbol{\chi}' = \boldsymbol{\chi} \cup (\mathbf{res} + 2) \mapsto \iota,$$
$$\tau \subseteq \tau' \cup \mathbf{spent}(\chi' \setminus \chi),$$
$$\mathbf{last}(\boldsymbol{\chi} \cup \chi) < \mathbf{res} + 2 + 2\,|\,\iota \cap \tau\,|\}$$

Specification of `getAndInc()`

$$\{\ \boldsymbol{\tau} = \varnothing\ \}$$

`res ← getAndInc()`

Final tokens and self-history

$$\{\exists\ \iota,\ \boxed{\begin{array}{l}\boldsymbol{\tau}' = \varnothing, \\ \boldsymbol{\chi}' = \boldsymbol{\chi} \cup (\mathbf{res} + 2) \mapsto \iota,\end{array}}$$

$$\tau \subseteq \tau' \cup \mathbf{spent}(\chi' \setminus \chi),$$

$$\mathrm{last}(\boldsymbol{\chi} \cup \chi) < \mathbf{res} + 2 + 2\,|\,\iota \cap \tau\,|\}$$

Specification of `getAndInc()`

$$\{\, \boldsymbol{\tau} = \emptyset \,\}$$

`res ← getAndInc()`

$$\begin{aligned}
&\{\exists\, \iota,\ \boldsymbol{\tau}' = \emptyset,\\
&\quad \boldsymbol{\chi}' = \boldsymbol{\chi} \cup (\mathrm{res} + 2) \mapsto \iota,\\
&\quad \boxed{\tau \subseteq \tau' \cup \mathrm{spent}(\chi' \setminus \chi),}\\
&\quad \mathrm{last}(\boldsymbol{\chi} \cup \chi) < \mathrm{res} + 2 + 2\,|\,\iota \cap \tau\,|\}
\end{aligned}$$

Tokens don't go missing

Specification of `getAndInc()`

$$\{\ \boldsymbol{\tau} = \emptyset\ \}$$

$$\texttt{res} \leftarrow \texttt{getAndInc()}$$

$$\{\exists\ \iota,\ \ \boldsymbol{\tau}' = \emptyset,$$
$$\boldsymbol{\chi}' = \boldsymbol{\chi} \cup (\mathsf{res} + 2) \mapsto \iota,$$
$$\tau \subseteq \tau' \cup \mathsf{spent}(\chi' \setminus \chi),$$
$$\boxed{\mathsf{last}(\boldsymbol{\chi} \cup \chi) < \mathsf{res} + 2 + 2\,|\,\iota \cap \tau\,|\}}$$

result + 2 is *greater than any* previous value in the history (modulo **past** ∩ **present** interference)

What this spec is good for?

Implications of the spec for `getAndInc`

Each result corresponds to a fresh history entry

- **R_1**: different calls return *distinct* results (**strong concurrent counter**)
- **R_2**: two calls, separated by *period of quiescence*, take effect in their sequential order (**QC**)
- **R_3**: results of *two calls* in the same thread are out of order by no more than *2 ** (number of calls *interfering with both*) (**QQC**)

Implications of the spec for `getAndInc`

- **R_1**: different calls return *distinct* results (**strong concurrent counter**)
- **R_2**: two calls, separated by *period of quiescence*, take effect in their sequential order (**QC**)
- **R_3**: results of *two calls* in the same thread are out of order by no more than *2 ** (number of calls *interfering with both*) (**QQC**)

Exercising Quiescent Consistency

“quiescent moment”

```
(res₁, -) ← (getAndInc() || e₁);
(res₂, -) ← (getAndInc() || e₂);
return (res₁, res₂);
```

{ ¿ $res_1 < res_2$? }

Specification of interfering program

$$\{\ \boldsymbol{\tau} = \varnothing,\ \ \boldsymbol{\chi} = \varnothing\ \}$$

$$e_1$$

$$\{\boxed{\exists\ \eta_1},\ \boldsymbol{\tau} = \varnothing,\ \ \boldsymbol{\chi} = \boxed{\eta_1},\ \ \tau \subseteq \tau' \cup \mathbf{spent}(\chi' \backslash \chi)\ \}$$

adds an arbitrary number of history entries

Spec for parallel composition

$$\{\ \boldsymbol{\tau} = \emptyset\ \}$$

```
(res₁, -) ← getAndInc() || e₁
```

$$\{\exists\ \iota,\ \eta_1,\ \boldsymbol{\tau} = \emptyset,$$
$$\boldsymbol{\chi} = \boldsymbol{\chi} \cup \eta_1 \cup (\mathbf{res}_1{+}2) \mapsto \iota,$$
$$\tau \subseteq \tau' \cup \mathbf{spent}(\chi' \backslash \chi),$$
$$\mathbf{last}(\boldsymbol{\chi} \cup \chi) < \mathbf{res}_1 + 2 + 2\,|\,\iota \cap \tau\,|\}$$

Spec for parallel composition

$$\{\ \boldsymbol{\tau} = \varnothing\ \}$$

```
(res₁, -) ← getAndInc() || e₁
```

$$\{\exists\ \iota,\ \eta_1,\ \boldsymbol{\tau} = \varnothing,$$

$$\boldsymbol{\chi} = \boldsymbol{\chi} \cup \eta_1 \cup (\mathrm{res}_1{+}2) \mapsto \iota,$$

$$\tau \subseteq \tau' \cup \mathrm{spent}(\chi' \backslash \chi),$$

$$\mathrm{last}(\boldsymbol{\chi} \cup \chi) < \mathrm{res}_1 + 2 + 2\,|\,\iota \cap \tau\,|\}$$

```
(res₁, -) ← getAndInc() || e₁;

(res₂, -) ← getAndInc() || e₂;

return (res₁, res₂);
```

```
(res₁, -) ← getAndInc() || e₁;

(res₂, -) ← getAndInc() || e₂;

return (res₁, res₂);
```

$$\{\ \boldsymbol{\tau} = \varnothing\ \}$$

```
(res1, -) ← getAndInc() || e1;
```

$$\{\ \exists\ \eta_1,\ \boldsymbol{\tau} = \varnothing,\ \boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),\ \chi \subseteq \chi'\ \}$$

```
(res2, -) ← getAndInc() || e2;
```

$$\{\exists\ \eta_1,\eta_2,\ \iota,\ \ \boldsymbol{\tau} = \varnothing,\ \boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),\ \tau' \subseteq \tau'' \cup \mathsf{spent}(\chi''\backslash\chi'),\ \mathsf{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \tau'\,|\}$$

```
return (res1, res2);
```

$\{\, \mathbf{T} = \varnothing \,\}$

```
(res₁, -) ← getAndInc() || e₁;
```

$\{\, \exists\, \eta_1,\ \mathbf{T} = \varnothing,$

$\quad \boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$

$\quad \chi \subseteq \chi' \,\}$

```
(res₂, -) ← getAndInc() || e₂;
```

$\{\exists\, \eta_1,\eta_2,\ \iota,\quad \mathbf{T} = \varnothing,$

$\quad \boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$

$\quad \tau' \subseteq \tau'' \cup \mathsf{spent}(\chi'' \backslash \chi'),$

$\quad \mathsf{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \tau'\,|\}$

```
return (res₁, res₂);
```

$$\{\ \mathbf{T} = \varnothing\ \}$$

```
(res1, -) ← getAndInc() || e1;
```

$$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,$$
$$\boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$$
$$\chi \subseteq \chi'\ \}$$

```
(res2, -) ← getAndInc() || e2;
```

$$\{\exists\ \eta_1,\eta_2,\ \iota,\quad \mathbf{T} = \varnothing,$$
$$\boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$$
$$\tau' \subseteq \tau'' \cup \mathrm{spent}(\chi''\backslash\chi'),$$
$$\mathrm{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \tau'\,|\}$$

```
return (res1, res2);
```

$\{\ \mathbf{T} = \varnothing\ \}$

```
(res₁, -) ← getAndInc() || e₁;
```

$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,$

$\quad \mathbf{X}' = \mathbf{X} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$

$\quad \chi \subseteq \chi'\ \}$

```
(res₂, -) ← getAndInc() || e₂;
```

$\{\exists\ \eta_1,\eta_2,\ \iota,\quad \mathbf{T} = \varnothing,$

$\mathbf{X}'' = \mathbf{X} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$

$\tau' \subseteq \tau'' \cup \mathsf{spent}(\chi'' \setminus \chi'),$

$\mathsf{last}(\mathbf{X}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \tau'\,|\}$

```
return (res₁, res₂);
```

No more forked threads at this point!

$$\{\ \mathbf{T} = \varnothing\ \}$$

```
(res₁, -) ← getAndInc() || e₁;
```

$$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,\quad \mathbf{X}' = \mathbf{X} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),\quad \chi \subseteq \chi'\ \}$$

```
(res₂, -) ← getAndInc() || e₂;
```

$$\{\exists\ \eta_1,\eta_2,\ \iota,\quad \mathbf{T} = \varnothing,\quad \mathbf{X}'' = \mathbf{X} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),\quad \tau' \subseteq \varnothing \cup \mathsf{spent}(\varnothing),\quad \mathsf{last}(\mathbf{X}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \tau'\,|\}$$

```
return (res₁, res₂);
```

$$\{\ \boldsymbol{\tau} = \emptyset\ \}$$

```
(res₁, -) ← getAndInc() || e₁;
```

$$\{\ \exists\ \eta_1,\ \boldsymbol{\tau} = \emptyset,$$
$$\boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$$
$$\chi \subseteq \chi'\ \}$$

```
(res₂, -) ← getAndInc() || e₂;
```

$$\{\exists\ \eta_1,\eta_2,\ \iota,\quad \boldsymbol{\tau} = \emptyset,$$
$$\boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$$
$$\tau' = \emptyset,$$
$$\mathrm{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \tau'\,|\}$$

```
return (res₁, res₂);
```

$\{\ \mathbf{T} = \emptyset\ \}$

`(res₁, -) ← getAndInc() || e₁;`

$\{\ \exists\ \eta_1,\ \mathbf{T} = \emptyset,$

$\quad \boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$

$\quad \chi \subseteq \chi'\ \}$

`(res₂, -) ← getAndInc() || e₂;`

$\{\exists\ \eta_1, \eta_2,\ \iota,\quad \mathbf{T} = \emptyset,$

$\boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$

$\tau' = \emptyset,$

$\mathsf{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \tau'\,|\}$

```
return (res₁, res₂);
```

$$\{\ \mathbf{T} = \varnothing\ \}$$

```
(res1, -) ← getAndInc() || e1;
```

$$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,\ \boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),\ \chi \subseteq \chi'\ \}$$

```
(res2, -) ← getAndInc() || e2;
```

$$\{\exists\ \eta_1, \eta_2,\ \iota,\ \ \mathbf{T} = \varnothing,\ \boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),\ \tau' = \varnothing,\ \mathrm{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\,|\,\iota \cap \varnothing\,|\}$$

```
return (res1, res2);
```

$$\{\ \mathbf{T} = \varnothing\ \}$$

```
(res₁, -) ← getAndInc() || e₁;
```

$$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,$$
$$\boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$$
$$\chi \subseteq \chi'\ \}$$

```
(res₂, -) ← getAndInc() || e₂;
```

$$\{\exists\ \eta_1,\eta_2,\ \iota,\quad \mathbf{T} = \varnothing,$$
$$\boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$$
$$\mathrm{T}' = \varnothing,$$
$$\mathrm{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2 + 2\ |\ \varnothing\ |\}$$

```
return (res₁, res₂);
```

$\{\ \mathbf{T} = \emptyset\ \}$

```
(res1, -) ← getAndInc() || e1;
```

$\{\ \exists\ \eta_1,\ \mathbf{T} = \emptyset,$

$\quad \boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$

$\quad \chi \subseteq \chi'\ \}$

```
(res2, -) ← getAndInc() || e2;
```

$\{\exists\ \eta_1, \eta_2,\ \iota,\quad \mathbf{T} = \emptyset,$

$\quad \boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$

$\quad \tau' = \emptyset,$

$\quad \mathrm{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2\ \}$

```
return (res1, res2);
```

$\{\ \mathbf{T} = \varnothing\ \}$

```
(res1, -) ← getAndInc() || e1;
```

$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,$

$\quad \boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$

$\quad \chi \subseteq \chi'\ \}$

```
(res2, -) ← getAndInc() || e2;
```

$\{\exists\ \eta_1, \eta_2,\ \iota,\quad \mathbf{T} = \varnothing,$

$\boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$

$\mathrm{T}' = \varnothing,$

$\mathrm{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2\ \}$

```
return (res1, res2);
```

$$\{\ \mathbf{T} = \varnothing\ \}$$

```
(res₁, -) ← getAndInc() || e₁;
```

$$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,\ \ \boldsymbol{\chi}' = \boldsymbol{\chi} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),\ \ \chi \subseteq \chi'\ \}$$

```
(res₂, -) ← getAndInc() || e₂;
```

$$\{\exists\ \eta_1, \eta_2,\ \iota,\ \ \mathbf{T} = \varnothing,\ \ \boldsymbol{\chi}'' = \boldsymbol{\chi} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),\ \ \mathrm{T}' = \varnothing,\ \ \mathrm{last}(\boldsymbol{\chi}'' \cup \chi') < \mathtt{res}_2 + 2\ \}$$

```
return (res₁, res₂);
```

$\{\ \mathbf{T} = \varnothing\ \}$

```
(res₁, -) ← getAndInc() || e₁;
```

$\{\ \exists\ \eta_1,\ \mathbf{T} = \varnothing,$
$\quad \mathbf{X}' = \mathbf{X} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$
$\quad X \subseteq X'\ \}$

```
(res₂, -) ← getAndInc() || e₂;
```

$\{\exists\ \eta_1, \eta_2,\ \iota,\quad \mathbf{T} = \varnothing,$
$\quad \mathbf{X}'' = \mathbf{X} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$
$\quad T' = \varnothing,$
$\quad \mathtt{res}_1 + 2 < \mathtt{res}_2 + 2\ \}$

```
return (res₁, res₂);
```

$\{\, \mathbf{T} = \varnothing \,\}$

```
(res₁, -) ← getAndInc() || e₁;
```

$\{\, \exists\, \eta_1,\ \mathbf{T} = \varnothing,$

$\quad \mathbf{X}' = \mathbf{X} \cup \eta_1 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}),$

$\quad \mathrm{X} \subseteq \mathrm{X}' \,\}$

```
(res₂, -) ← getAndInc() || e₂;
```

$\{\exists\, \eta_1, \eta_2, \iota,\ \ \mathbf{T} = \varnothing,$

$\mathbf{X}'' = \mathbf{X} \cup \eta_1 \cup \eta_2 \cup (\mathtt{res}_1{+}2 \mapsto \text{-}) \cup (\mathtt{res}_2{+}2 \mapsto \text{-}),$

$\mathrm{T}' = \varnothing,$

$\mathtt{res}_1 < \mathtt{res}_2 \,\}$

```
return (res₁, res₂);
```

Implications of the spec for `getAndInc`

- **R_1**: different calls return *distinct* results (**strong concurrent counter**)
- **R_2**: two calls, separated by *period of quiescence*, take effect in their sequential order (**QC**)
- **R_3**: results of *two calls* in the same thread are out of order by no more than *2 ** (number of calls *interfering with both*) (**QQC**)

Summary of the proof pattern

- Express interference that matters via auxiliary state — *tokens*;
- Capture past interference and results in auxiliary *histories*;
- Assume closed world to *bound* interference (quiescence).

What's in the paper

- ***Full formal specification*** of the counting network;
- Formal proofs of ***QC*** and ***QQC*** properties for the network;
- Discussion on applying the technique for ***QC-queues***;
- Spec and verification of `java.util.concurrent.Exchanger`;
- Verification of an exchanger client in the spirit of ***concurrency-aware linearizability*** (CAL).
- Report on implementation in FCSL/Coq.

To take away

Hoare-style Specifications for Non-linearizable Concurrent Objects

- *Compositional* — substitution principle;
- *Syntactic proof method* — inference rules;
- *Uniform* — reasoning about objects and their clients in the *same* proof system.

Good specification is in the eye of the beholder.

Thanks!