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

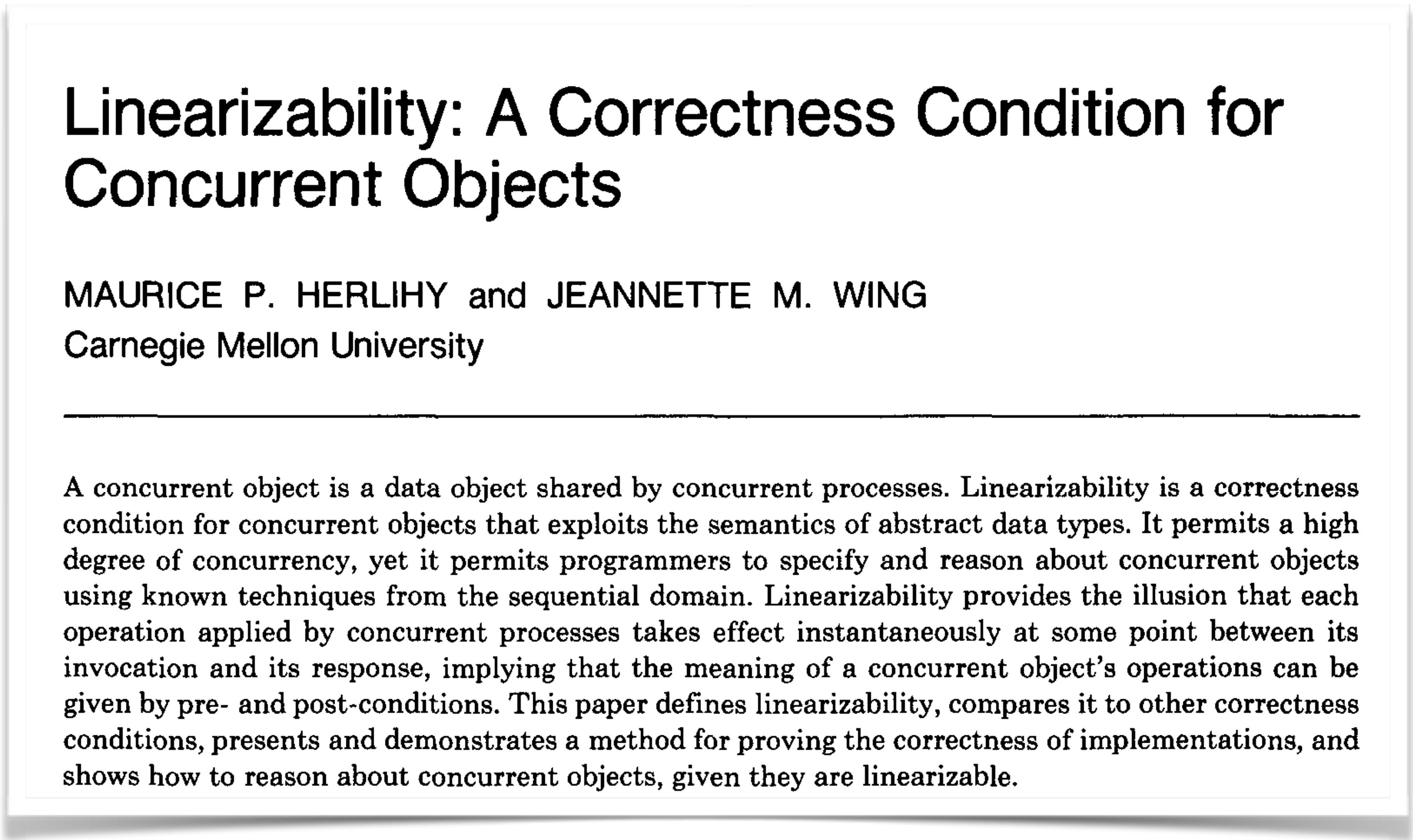
Ilya Sergey

UCL PPLV

institute IMDEA software

joint work with
Aleks Nanevski, Anindya Banerjee, and Germán Andrés Delbianco

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

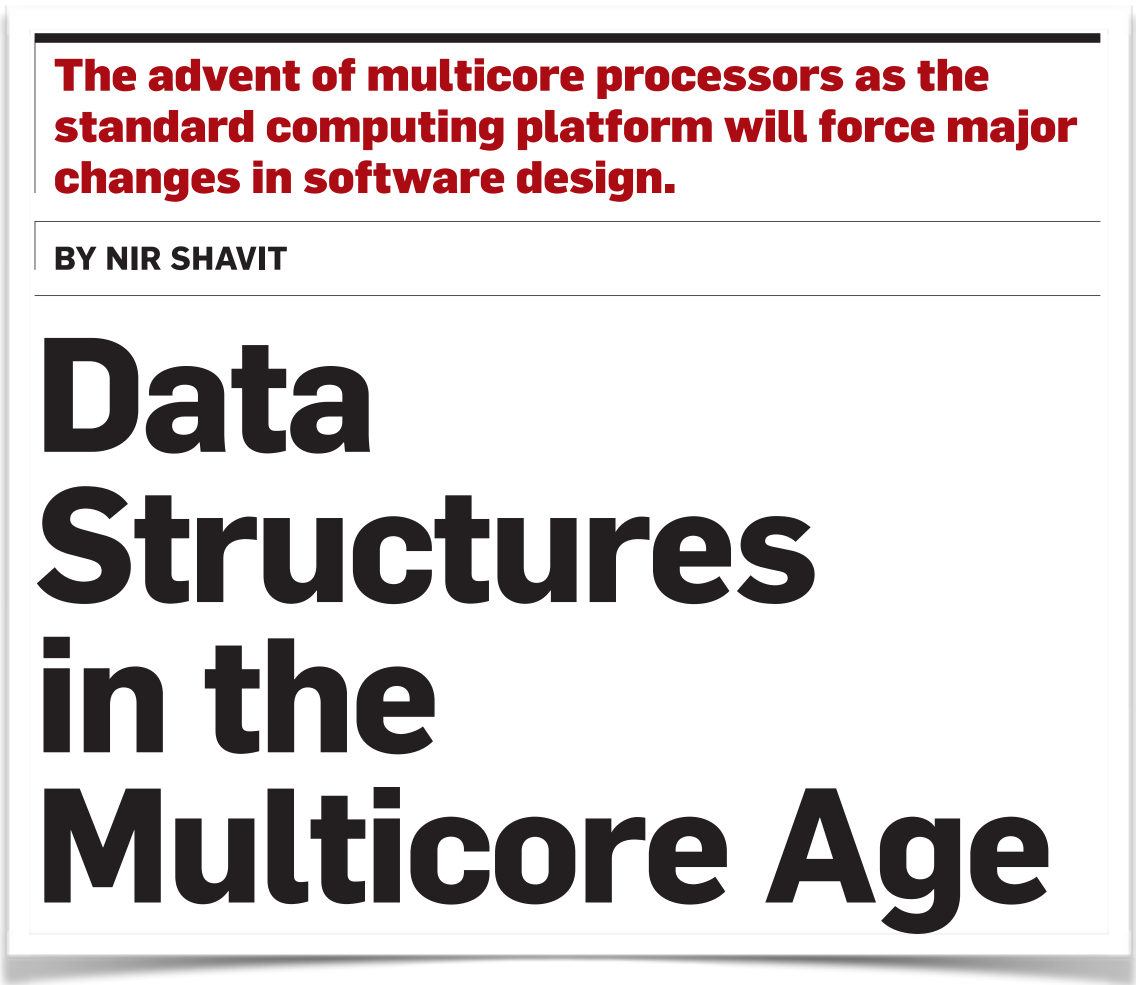
Danny Hendler
Ben-Gurion University
hendlerd@cs.bgu.ac.il

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

An alternative to linearizability?



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 (QC) [Aspnes-al:JACM94]
- Quasi-Linearizability (QL) [Afek-al:OPODIS10]
- Quantitative Relaxation (QR) [Henzinger-al:POPL13]
- Concurrency-Aware Linearizability (CAL) [Hemed-Rinetzky:PODC14]
- Quantitative Quiescent Consistency (QQC) [Jagadeesan-Riely:ICALP14]
- Local Linearizability (LL) [Haas-al:arXiv15]
- …

Challenges of diversity

- *Composing* different conditions (CAL, QC, QQC, *etc.*) 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

UCL PPLV

institute IMdea software

joint work with
Aleks Nanevski, Anindya Banerjee, and Germán Andrés Delbianco

Hoare-style Specifications

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

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

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

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

Hoare-style Specifications 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 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
```

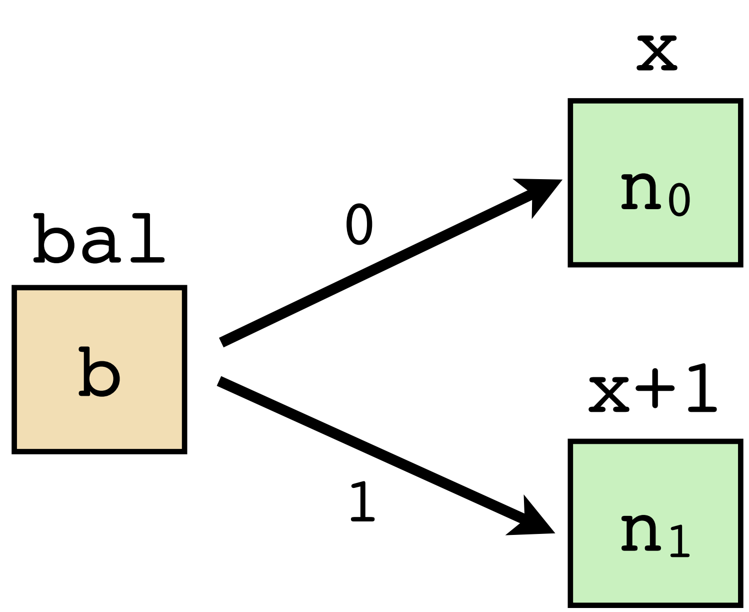
Simple Counting Network

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

high contention location

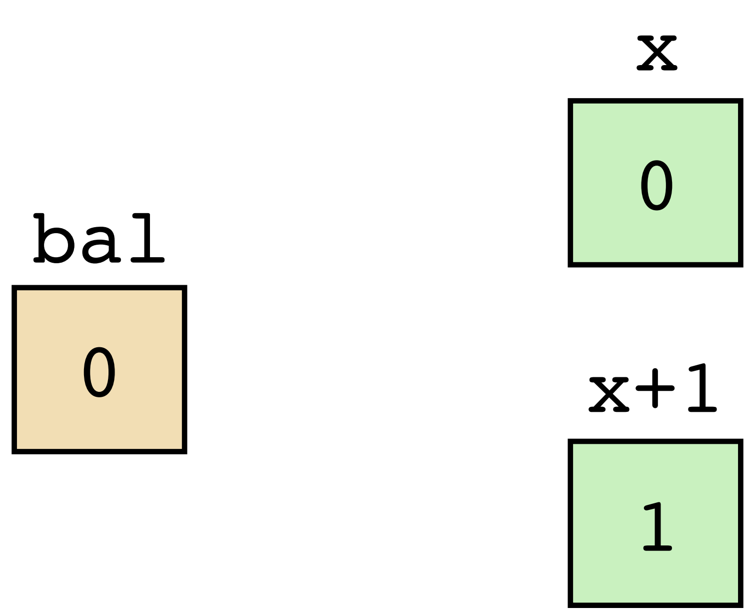
Simple Counting Network

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



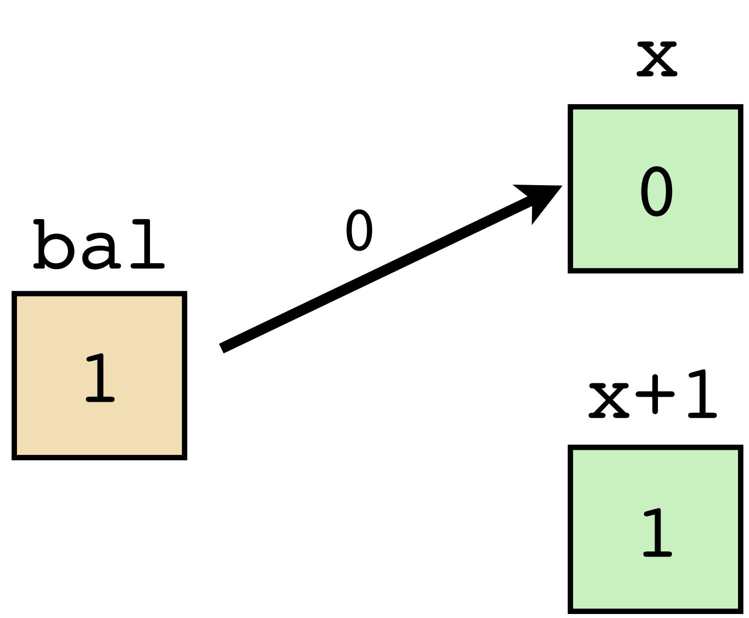
Sequential Execution (T_1)

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



Sequential Execution (T_1)

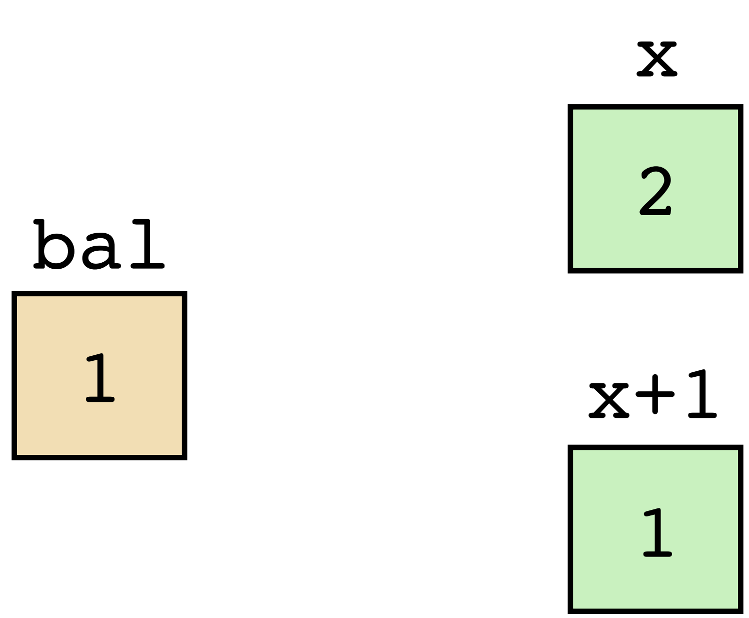
```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



$T_1.b_1 = 0$

Sequential Execution (T_1)

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

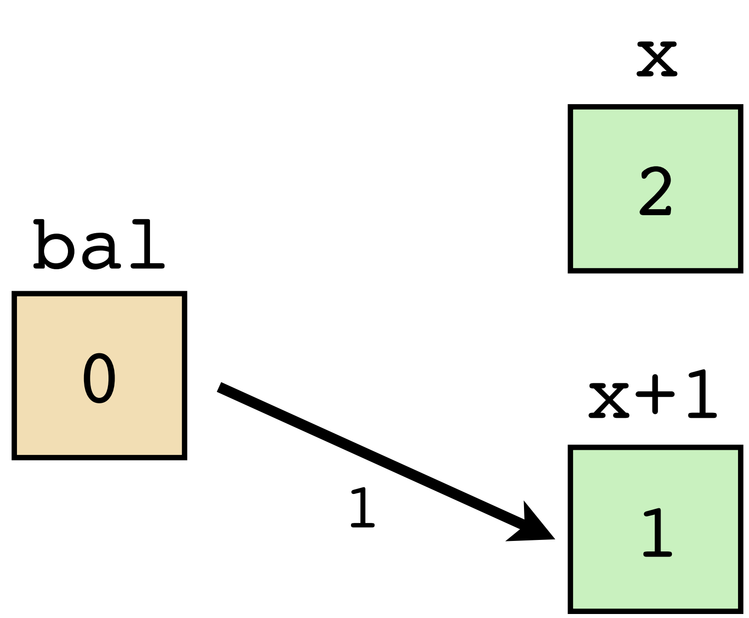


$T_1.b_1 = 0$

$T_1.res_1 = 0$

Sequential Execution (T_1)

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



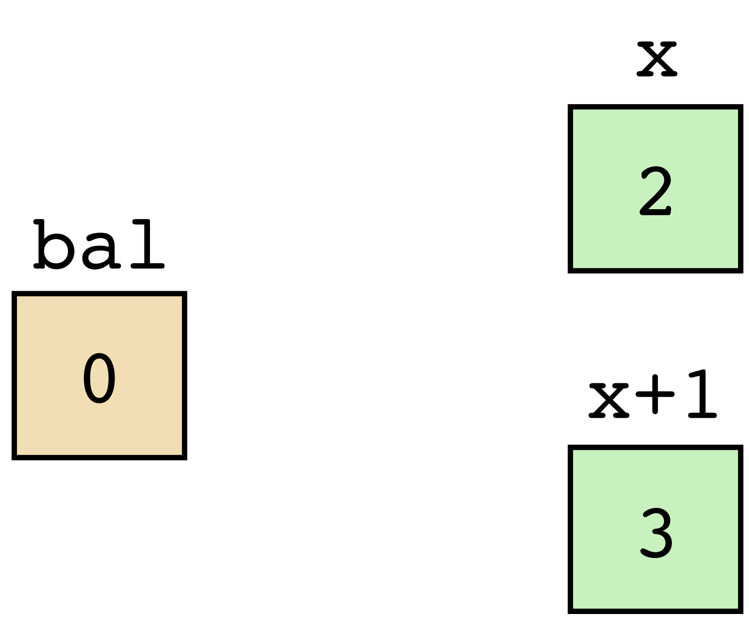
$T_1.b_1 = 0$

$T_1.res_1 = 0$

$T_1.b_2 = 1$

Sequential Execution (T_1)

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



$T_1.b_1 = 0$

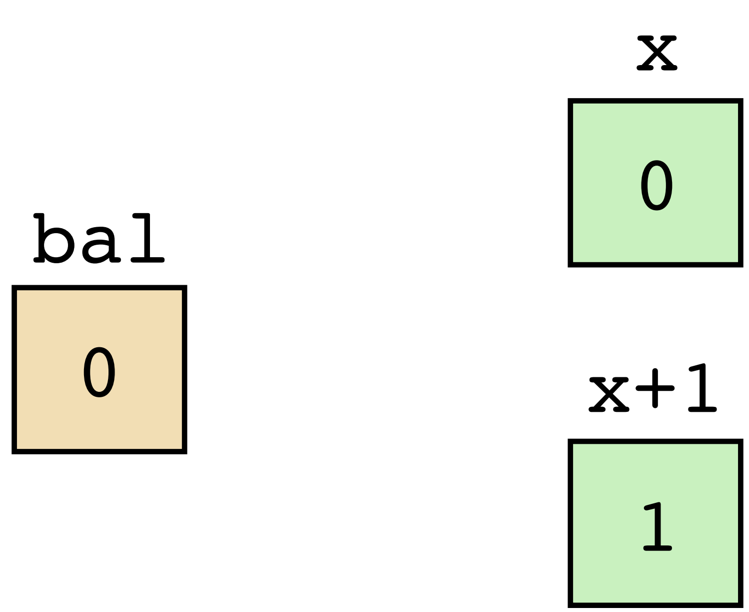
$T_1.res_1 = 0$

$T_1.b_2 = 1$

$T_1.res_2 = 1$

Concurrent Execution (T_1, T_2)

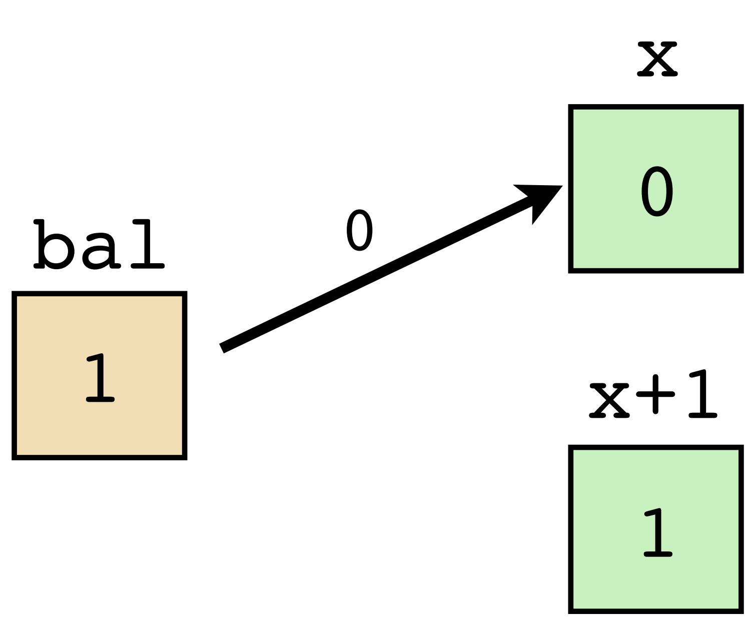
```
def getAndInc() : nat = {
    b   ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```



Concurrent Execution (T_1, T_2)

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

$T_1.b_1 = 0$

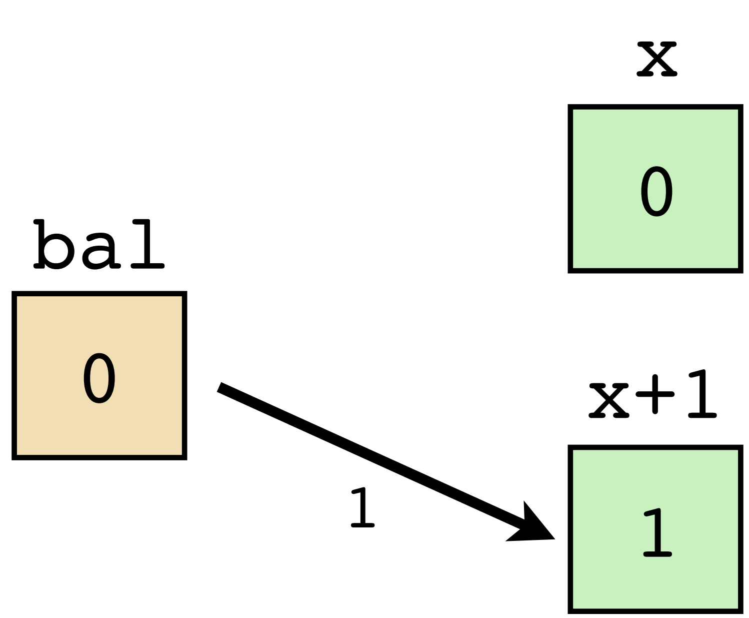


Concurrent Execution (T_1, T_2)

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

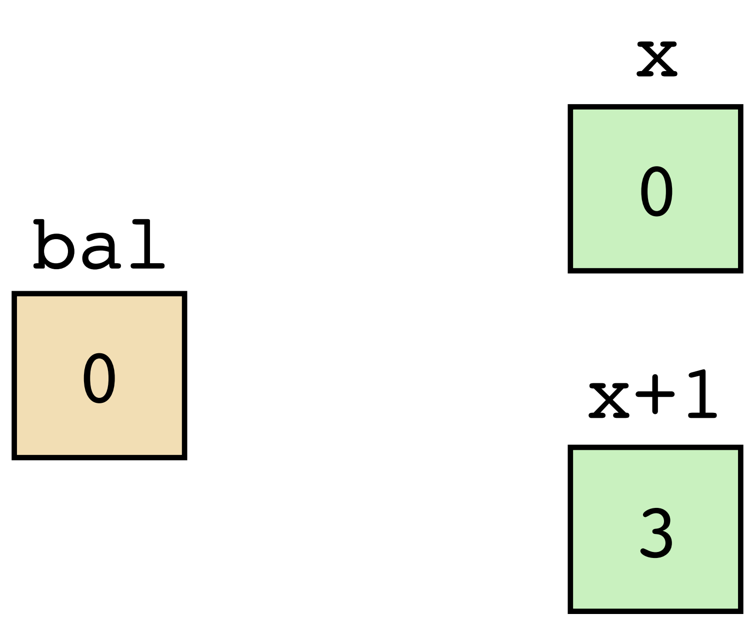
$T_1.b_1 = 0$

$T_2.b_1 = 1$



Concurrent Execution (T_1, T_2)

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



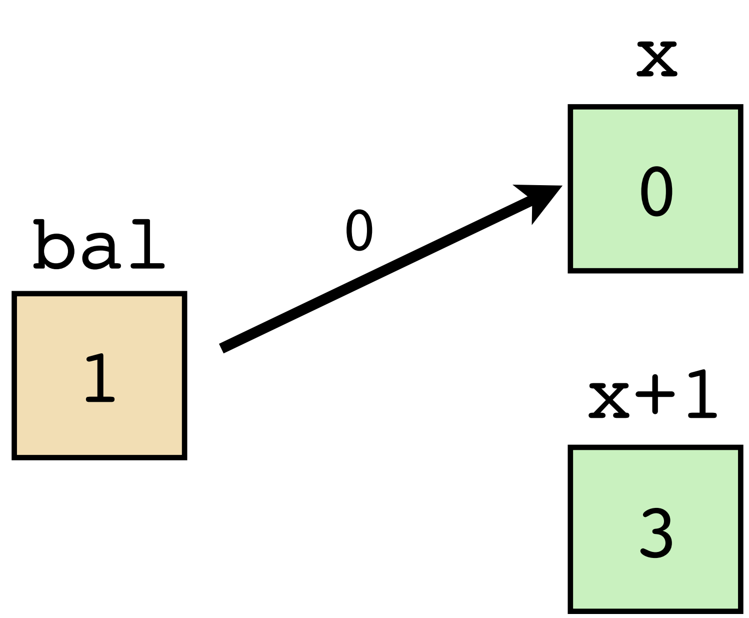
$T_1.b_1 = 0$

$T_2.b_1 = 1$

$T_2.res_1 = 1$

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**)

Invariants of 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;

- Threads, which gained capabilities but haven't yet incremented, cause one counter to *"run ahead"* of another one, leading to *out-of-order* anomalies.

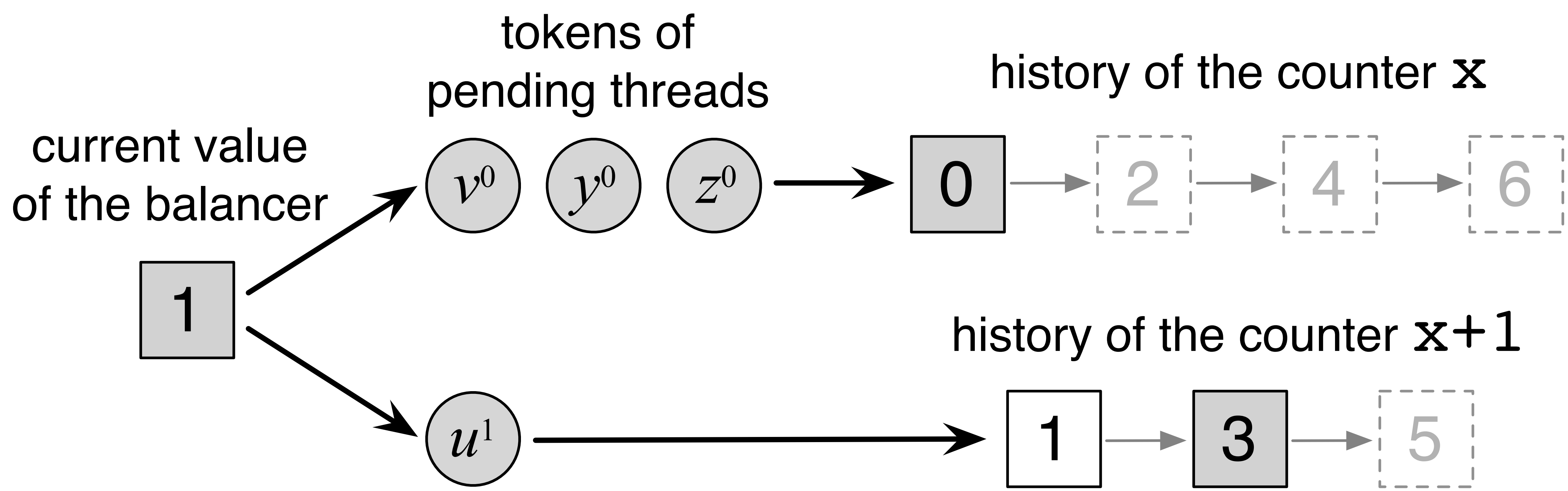
"Histories"

Sergey-al:ESOP15

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, represented as a PCM (S, ⊕, **0**), e.g.,
  - ✦ *Tokens* — disjoint sets;
  - ✦ *Histories* — partial maps with `nat` as domain.

Tokens and Histories of the Network



- New *unique* tokens are emitted upon calling **`flip()`**;
- Calling **`fetchAndAdd2()`** *consumes* a token and *adds* an entry to the history.

Interference-capturing histories

$$\eta = \{\ \ldots, \boxed{t} \mapsto (\iota, z), \ \ldots\ \}$$

“‘timestamp’”, a value written to a counter **`x`** or **`x+1`** (0, 1, 2, etc.)

Interference-capturing histories

$$\eta = \{ \ldots, t \mapsto (\iota, z), \ldots \}$$

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

Interference-capturing histories

$$\eta = \{\ \ldots,\ t \mapsto (\iota, \boxed{Z}),\ \ldots\ \}$$

a token, spent to increment **`x`** or **`x+1`** from t-2 to t

Notation for *Subjective* Histories and Tokens

- χ_s, χ_o — histories, contributed by *self* and *other* threads;
- τ_s, τ_o — tokens, held by *self* and *other* threads;
- η, ι — logical variables for histories and tokens.

Specification of `getAndInc()`

$$\{ \quad \tau_s = \emptyset, \ \ \chi_s = \eta_s,$$
$$\eta_o \subseteq \chi_o,$$
$$\iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o) \quad \}$$

`getAndInc()`

$$\{\exists\, \iota, z, \quad \tau_s = \emptyset, \ \ \chi_s = \eta_s \cup \mathbf{res}{+}2 \mapsto (\iota, z),$$
$$\eta_o \subseteq \chi_o, \ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o),$$
$$\mathbf{last}(\eta_s \cup \eta_o) < \mathbf{res} + 2 + 2\,|\,\iota \cap \iota_o\,|\} \ @\ C$$

Specification of `getAndInc()`

$$\left\{ \boxed{\tau_s = \varnothing,\ \ \chi_s = \eta_s,} \quad \begin{array}{c} \eta_o \subseteq \chi_o, \\ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o) \end{array} \right\}$$

initial tokens
and self-history

```
getAndInc()
```

$$\{\exists\, \iota, z, \quad \tau_s = \varnothing,\ \ \chi_s = \eta_s \cup \mathbf{res}{+}2 \mapsto (\iota, z),$$
$$\eta_o \subseteq \chi_o,\ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o),$$
$$\mathbf{last}(\eta_s \cup \eta_o) < \mathbf{res} + 2 + 2\,|\,\iota \cap \iota_o\,|\} \ @\ \mathcal{C}$$

Specification of `getAndInc()`

boundary on initial
other-history and tokens

$$\{\quad \tau_s = \emptyset,\ \ \chi_s = \eta_s,$$
$$\eta_o \subseteq \chi_o,$$
$$\iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o) \quad \}$$

```
getAndInc()
```

$$\{\exists\, \iota,\, z, \quad \tau_s = \emptyset,\ \ \chi_s = \eta_s \cup \mathbf{res}{+}2 \mapsto (\iota, z),$$
$$\eta_o \subseteq \chi_o,\ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o),$$
$$\mathbf{last}(\eta_s \cup \eta_o) < \mathbf{res} + 2 + 2\,|\,\iota \cap \iota_o\,|\} \ @\ C$$

Specification of `getAndInc()`

$$\{\quad \tau_s = \emptyset,\ \ \chi_s = \eta_s,$$
$$\eta_o \subseteq \chi_o,$$
$$\iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o) \quad \}$$

final tokens
and self-history

`getAndInc()`

$$\{\exists\ \iota,\ z,\quad \boxed{\tau_s = \emptyset,\ \ \chi_s = \eta_s \cup \mathbf{res}{+}2 \mapsto (\iota,\ z)},$$
$$\eta_o \subseteq \chi_o,\ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o),$$
$$\mathbf{last}(\eta_s \cup \eta_o) < \mathbf{res} + 2 + 2\,|\,\iota \cap \iota_o\,|\} \ @\ C$$

Specification of `getAndInc()`

$$\{ \quad \tau_s = \emptyset,\ \ \chi_s = \eta_s,$$
$$\eta_o \subseteq \chi_o,$$
$$\iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o) \quad \}$$

constraining final
other-history and tokens

`getAndInc()`

$$\{\exists\, \iota, z, \quad \tau_s = \emptyset,\ \ \chi_s = \eta_s \cup \mathbf{res}{+}2 \mapsto (\iota, z),$$
$$\boxed{\eta_o \subseteq \chi_o,\ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o)},$$
$$\mathbf{last}(\eta_s \cup \eta_o) < \mathbf{res} + 2 + 2\,|\,\iota \cap \iota_o\,|\} \ @\ C$$

Specification of `getAndInc()`

$$\{\quad \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ \eta_o \subseteq \chi_o,\ \ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o) \quad\}$$

`getAndInc()`

$$\{\exists\, \iota, z,\quad \tau_s = \varnothing,\ \ \chi_s = \eta_s \cup \mathbf{res}{+}2 \mapsto (\iota, z),\ \ \eta_o \subseteq \chi_o,\ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o),\ \ \boxed{\mathbf{last}(\eta_s \cup \eta_o) < \mathbf{res} + 2 + 2\,|\,\iota \cap \iota_o\,|}\} \ @\ C$$

result + 2 is
greater than any previous value
of the counters, recorded in history
(modulo past ∩ present interference)

Implications of the derived spec

Trivial from invariants: each result corresponds to a new 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 derived spec

- **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);
```

{ ¿ $res_1 < res_2$? }

Generic spec for Interference

$$\{\ \tau_S = \varnothing,\ \ \chi_S = \varnothing,\ \ \iota_O \subseteq \tau_O \cup \mathbf{spent}(\chi_O)\ \}$$

$$\mathtt{e_i}$$

$$\{\boxed{\exists\ \eta_i},\ \tau_S = \varnothing,\ \ \chi_S = \boxed{\eta_i}\ \ \iota_O \subseteq \tau_O \cup \mathbf{spent}(\chi_O)\ \}\ @\ C$$

arbitrary contribution to the history

Spec for parallel composition

$\{\ \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ \eta_o \subseteq \chi_o,\ \iota_o \subseteq \tau_o \cup \textbf{spent}(\chi_o \setminus \eta_o),\ \ldots\ \}$

```
getAndInc() || eᵢ
```

$\{\exists\ \iota, z, \eta_i,\ \ \tau_s = \varnothing,\ \ \chi_s = \eta_s \cup \eta_i \cup \textbf{res}{+}2 \mapsto (\iota, z), \eta_o \subseteq \chi_o,$
$\iota_o \subseteq \tau_o \cup \textbf{spent}(\chi_o \setminus \eta_0),$
$\textbf{last}(\eta_s \cup \eta_o) < \textbf{res}.1 + 2 + 2\,|\,\iota \cap \iota_o\,|\} \ @\ C$

Spec for parallel composition

$\{\ \tau_s = \emptyset,\ \ \chi_s = \eta_s,\ \ \eta_o \subseteq \chi_o,\ \iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o),\ \ldots\ \}$

```
getAndInc() || eᵢ
```

$\{\exists\ \iota, z, \eta_i,\ \ \tau_s = \emptyset,\ \ \chi_s = \eta_s \cup \eta_i \cup \mathbf{res}{+}2 \mapsto (\iota, z),\ \eta_o \subseteq \chi_o,$
$\iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_0),$
$\mathbf{last}(\eta_s \cup \eta_o) < \mathbf{res}.1 + 2 + 2\,|\,\iota \cap \iota_o\,|\}\ @\ C$

$$\{\ \tau_s = \emptyset,\ \ \chi_s = \eta_s,\ \ldots\ \}$$

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

$$\{\exists\ \eta_1,\ \tau_s = \emptyset,\ \boxed{\chi_s = \eta'_s},\ \ \eta_o \subseteq \chi_o,$$
$$\text{where}\ \boxed{\eta'_s = \eta_s \cup \eta_1 \cup \mathbf{res}_1{+}2 \mapsto -}\ ,\ \eta_o = \chi_o\ \text{and}\ \iota_o = \tau_o\ \}$$

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

$$\{\exists\ \eta_1,\eta_2,\ \iota,\ \tau_s = \emptyset,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o, \boxed{\chi_s = \eta'_s \cup \eta_2 \cup \mathbf{res}_2{+}2 \mapsto -}$$
$$\iota_o \subseteq \tau_o \cup \mathbf{spent}(\chi_o \setminus \eta_o),$$
$$\boxed{\mathbf{last}(\eta'_s \cup \eta_o) < \mathbf{res}_2 + 2 + 2\,|\,\iota \cap \iota_o\,|\}}$$

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

$\{\ \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ldots\ \}$

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

$\{\exists\ \eta_1,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,$
where $\eta'_s = \eta_s \cup \eta_1 \cup \mathtt{res_1}{+}2 \mapsto -$, $\eta_o = \chi_o$ and $\iota_o = \tau_o\ \}$

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

$\{\exists\ \eta_1, \eta_2,\ \iota,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,\ \chi_s = \eta'_s \cup \eta_2 \cup \mathtt{res_2}{+}2 \mapsto -$
$\iota_o \subseteq \tau_o \cup \mathsf{spent}(\chi_o \setminus \eta_o),$
$\mathsf{last}(\eta'_s \cup \eta_o) < \mathtt{res_2} + 2 + 2\,|\,\iota \cap \iota_o\,|\ \}$

```
return (res₁, res₂);
```

$\{ \tau_s = \emptyset, \;\; \chi_s = \eta_s, \; \ldots \}$

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

$\{\exists\, \eta_1, \tau_s = \emptyset, \;\; \chi_s = \eta'_s, \;\; \eta_o \subseteq \chi_o,$

where $\eta'_s = \eta_s \cup \eta_1 \cup \texttt{res}_1{+}2 \mapsto -$, $\eta_o = \chi_o$ and $\iota_o = \tau_o \}$

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

$\{\exists\, \eta_1, \eta_2, \iota, \tau_s = \emptyset, \;\; \chi_s = \eta'_s, \;\; \eta_o \subseteq \chi_o, \chi_s = \eta'_s \cup \eta_2 \cup \texttt{res}_2{+}2 \mapsto -$

$\iota_o \subseteq \emptyset,$

$\mathrm{last}(\eta'_s \cup \eta_o) < \texttt{res}_2 + 2 + 2\,|\,\iota \cap \iota_o\,| \}$

```
return (res₁, res₂);
```

$\{ \tau_S = \varnothing, \ \ \chi_S = \eta_S, \ \ldots \}$

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

$\{\exists\ \eta_1, \tau_S = \varnothing, \ \ \chi_S = \eta'_S, \ \ \eta_O \subseteq \chi_O,$

where $\eta'_S = \eta_S \cup \eta_1 \cup \texttt{res}_1{+}2 \mapsto -$, $\eta_O = \chi_O$ and $\iota_O = \tau_O$ $\}$

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

$\{\exists\ \eta_1,\eta_2, \iota, \tau_S = \varnothing, \ \ \chi_S = \eta'_S, \ \ \eta_O \subseteq \chi_O, \chi_S = \eta'_S \cup \eta_2 \cup \texttt{res}_2{+}2 \mapsto -$

$\iota_O = \varnothing,$

$\mathrm{last}(\eta'_S \cup \eta_O) < \texttt{res}_2 + 2 + 2\,|\,\iota \cap \iota_O\,|\,\}$

```
return (res₁, res₂);
```

$\{\ \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ldots\ \}$

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

$\{\exists\ \eta_1,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,$
where $\eta'_s = \eta_s \cup \eta_1 \cup \mathbf{res}_1{+}2 \mapsto -$, $\eta_o = \chi_o$ and $\iota_o = \tau_o\ \}$

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

$\{\exists\ \eta_1,\eta_2,\ \iota,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,\ \chi_s = \eta'_s \cup \eta_2 \cup \mathbf{res}_2{+}2 \mapsto -$
$\iota_o = \varnothing,$
$\mathrm{last}(\eta'_s \cup \eta_o) < \mathbf{res}_2 + 2 + 2\,|\,\iota \cap \iota_o\,|\ \}$

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

$\{\ \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ldots\ \}$

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

$\{\exists\ \eta_1,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,$

where $\eta'_s = \eta_s \cup \eta_1 \cup \mathtt{res}_1{+}2 \mapsto -$, $\eta_o = \chi_o$ and $\iota_o = \tau_o\ \}$

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

$\{\exists\ \eta_1,\eta_2,\ \iota,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,\ \chi_s = \eta'_s \cup \eta_2 \cup \mathtt{res}_2{+}2 \mapsto -$

$\iota_o = \varnothing,$

$\mathsf{last}(\eta'_s \cup \eta_o) < \mathtt{res}_2 + 2 + 2\,|\,\varnothing\,|\ \}$

```
return (res₁, res₂);
```

$\{\ \tau_s = \emptyset,\ \ \chi_s = \eta_s,\ \ldots\ \}$

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

$\{\exists\ \eta_1,\ \tau_s = \emptyset,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,$

where $\eta'_s = \eta_s \cup \eta_1 \cup \mathtt{res}_1{+}2 \mapsto -$, $\eta_o = \chi_o$ and $\iota_o = \tau_o\ \}$

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

$\{\exists\ \eta_1{,}\eta_2,\ \iota,\ \tau_s = \emptyset,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,\ \chi_s = \eta'_s \cup \eta_2 \cup \mathtt{res}_2{+}2 \mapsto -$

$\iota_o = \emptyset,$

$\mathsf{last}(\eta'_s \cup \eta_o) < \mathtt{res}_2 + 2\ \}$

```
return (res₁, res₂);
```

$\{\ \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ldots\ \}$

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

$\{\exists\ \eta_1,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,$

where $\eta'_s = \eta_s \cup \eta_1 \cup \mathtt{res_1}{+}2 \mapsto -$, $\eta_o = \chi_o$ and $\iota_o = \tau_o\ \}$

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

$\{\exists\ \eta_1, \eta_2,\ \iota,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,\ \chi_s = \eta'_s \cup \eta_2 \cup \mathtt{res_2}{+}2 \mapsto -$

$\iota_o = \varnothing,$

$\mathrm{last}(\eta'_s \cup \eta_o) < \mathtt{res_2} + 2\ \}$

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

$\{\ \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ldots\ \}$

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

$\{\exists\ \eta_1,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,$

where $\eta'_s = \eta_s \cup \eta_1 \cup \mathbf{res_1}{+}2 \mapsto$ - , $\eta_o = \chi_o$ and $\iota_o = \tau_o\ \}$

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

$\{\exists\ \eta_1,\eta_2,\ \iota,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,\ \chi_s = \eta'_s \cup \eta_2 \cup \mathbf{res_2}{+}2 \mapsto$ -

$\iota_o = \varnothing,$

$\mathbf{res_1} + 2 < \mathbf{res_2} + 2\ \}$

```
return (res₁, res₂);
```

$\{\ \tau_s = \varnothing,\ \ \chi_s = \eta_s,\ \ldots\ \}$

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

$\{\exists\ \eta_1,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,$

where $\eta'_s = \eta_s \cup \eta_1 \cup \mathtt{res}_1{+}2 \mapsto -\ ,\ \eta_o = \chi_o$ and $\iota_o = \tau_o\ \}$

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

$\{\exists\ \eta_1, \eta_2,\ \iota,\ \tau_s = \varnothing,\ \ \chi_s = \eta'_s,\ \ \eta_o \subseteq \chi_o,\ \chi_s = \eta'_s \cup \eta_2 \cup \mathtt{res}_2{+}2 \mapsto -$

$\iota_o = \varnothing,$

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

```
return (res₁, res₂);
```

Implications of the derived spec

- **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.

Not discussed today

- 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).

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.

Specification is in the eye of the beholder.

Thanks!