Commutativity Race Detection

Concepts, Algorithms and Open Problems

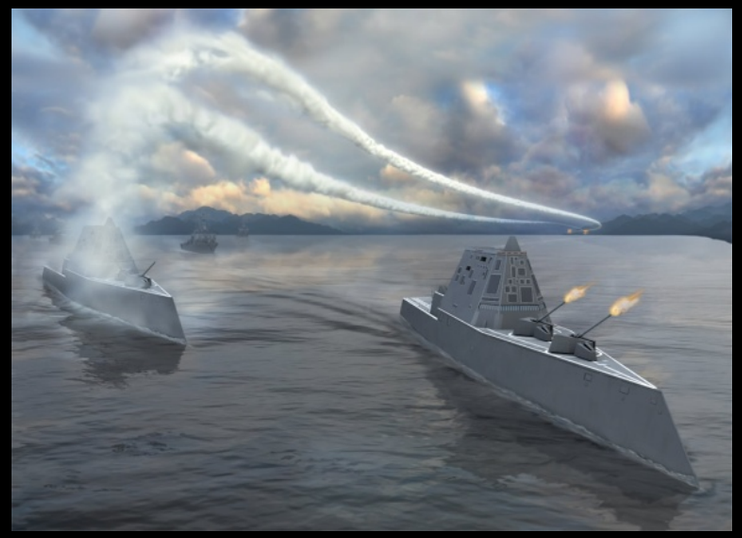
Martin Vechev

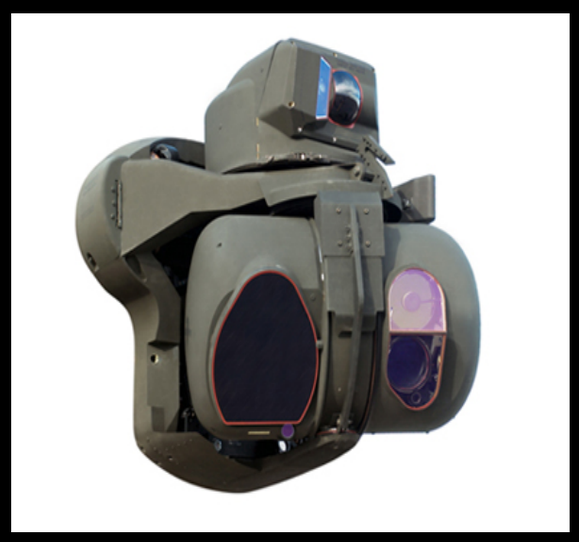
Department of Computer Science

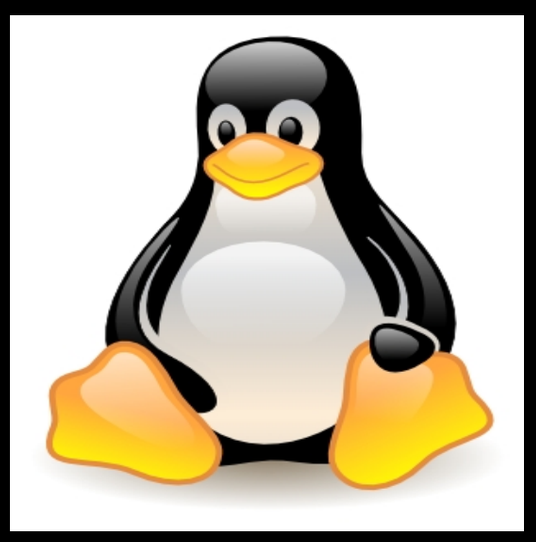
ETH Zurich

ETH zürich

Concurrency is Everywhere







Java™

Concurrency and Interference

Concurrency and Interference go together

Some interference is good (e.g., coordination)

Some interference is harmful (e.g., causes errors)

Data Races

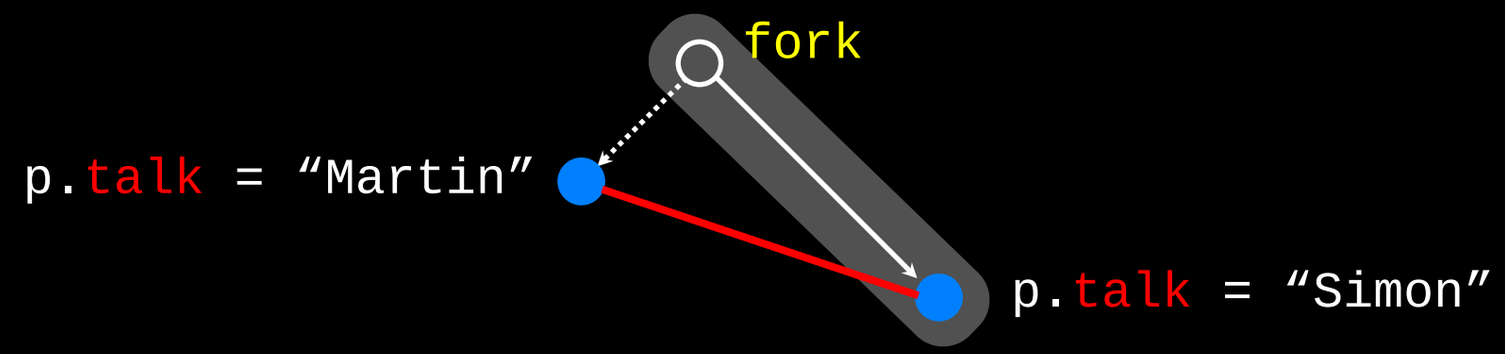
A generic notion of interference: data races

Data race: two unordered operations from different threads access the same memory location, where one of the accesses is a write

Example of a Data Race

```
PLACES p = new PLACES();

fork {
   p.talk = “Martin”
}

p.talk = “Simon”
```

1. *two writes to* p.talk
2. *not ordered by program*

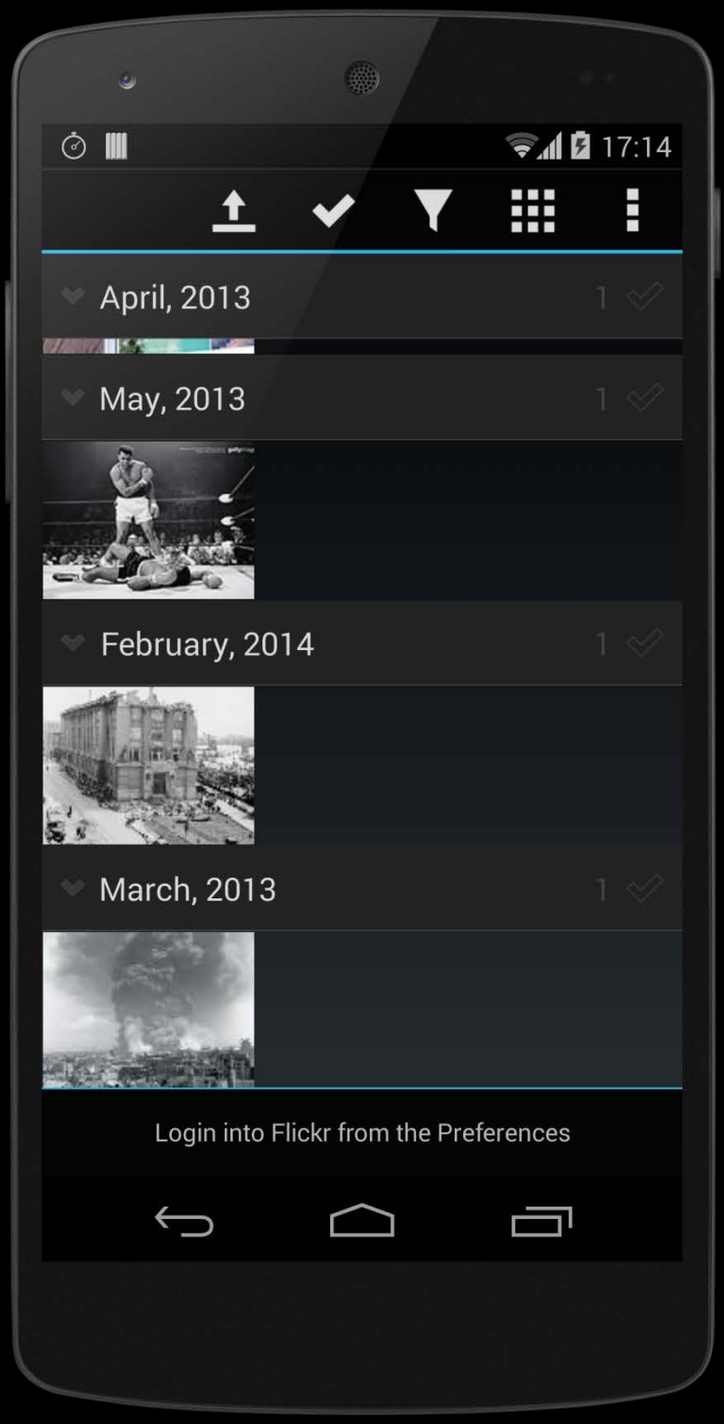


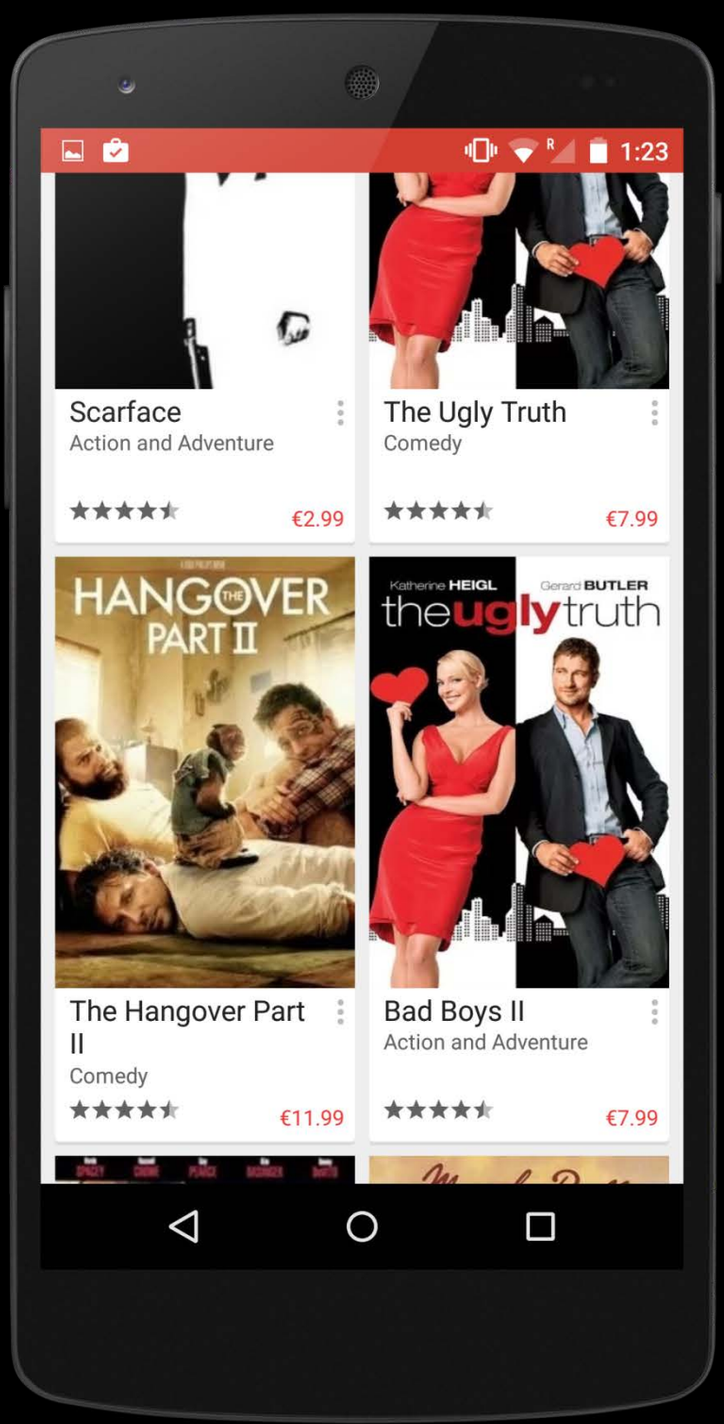
2003 Northeast Blackout

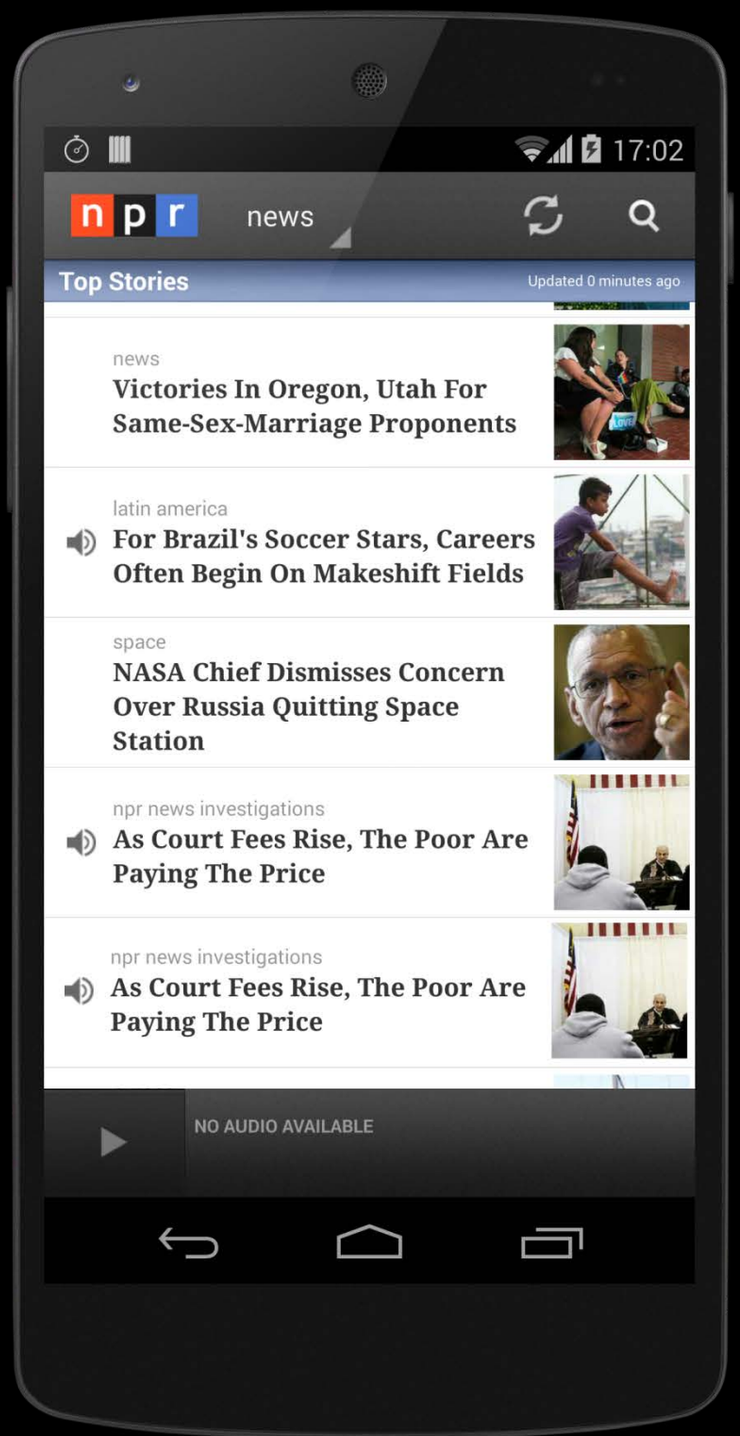
concurrency error triggers massive power outage economic effect : ~ 6 billion $USD

Android Data Races

Undesirable effects







Data races: Web

Data races: Web

Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```

Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```

Gates = great

Data races: Web

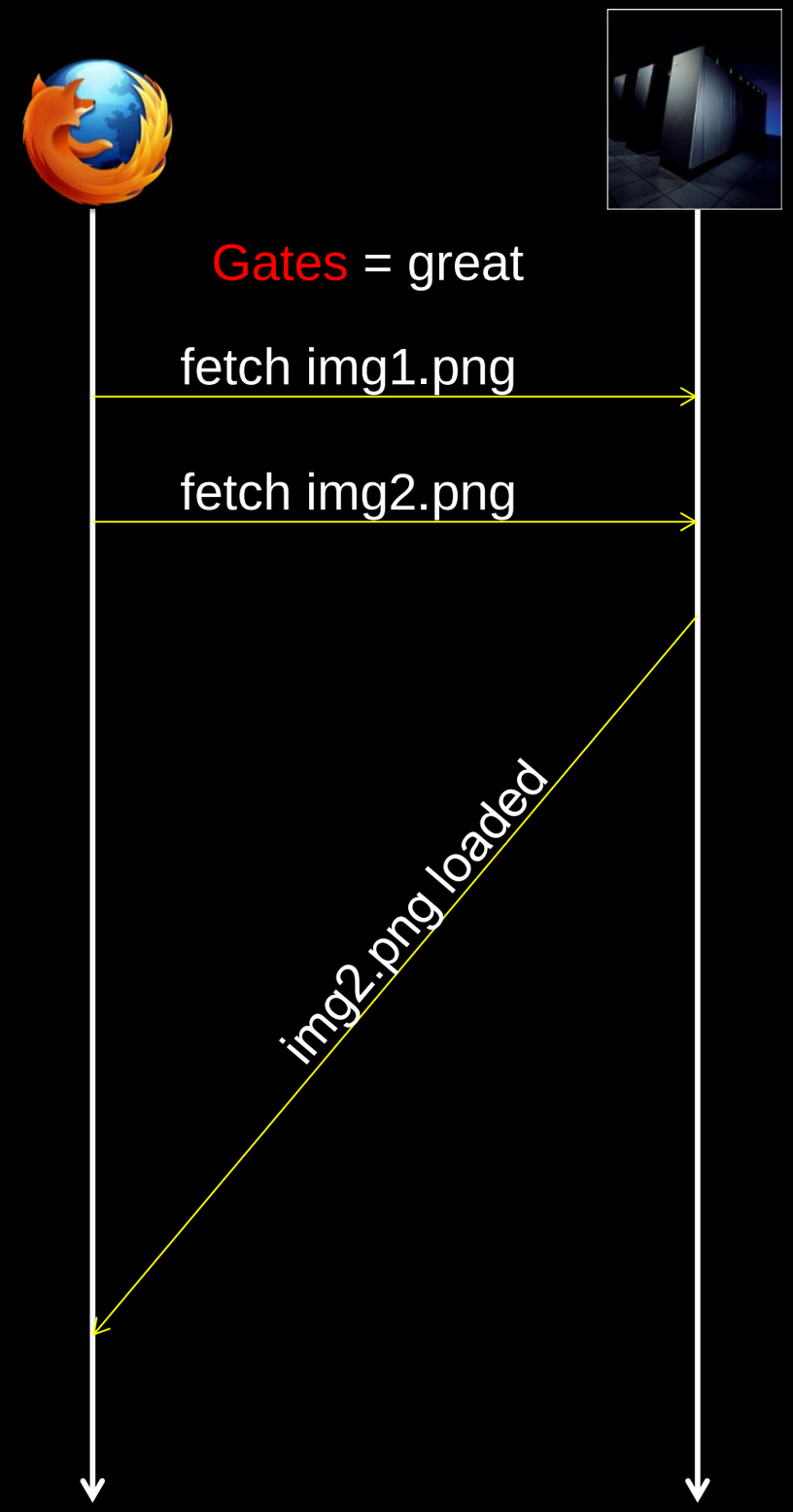
```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```

Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```

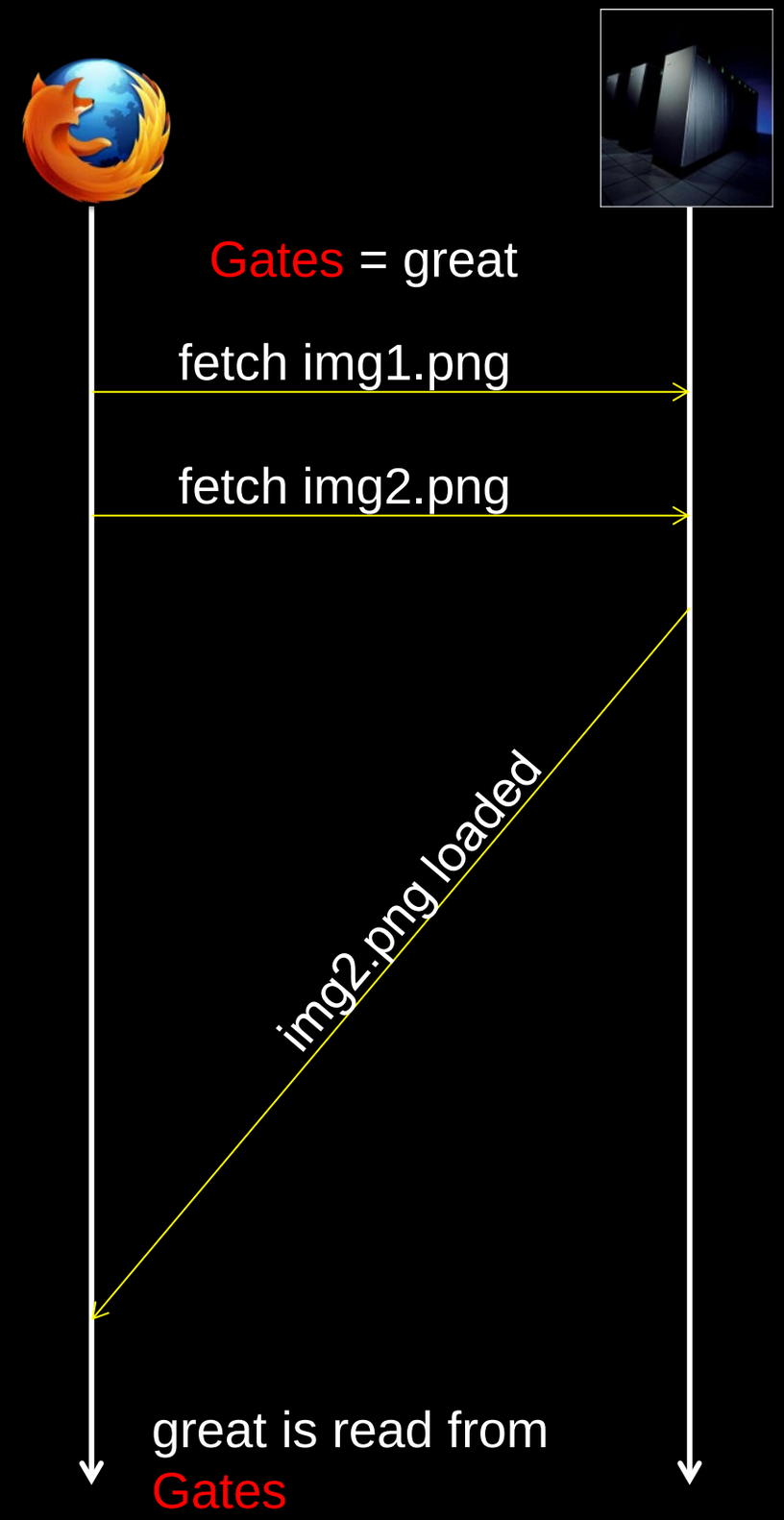
Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```



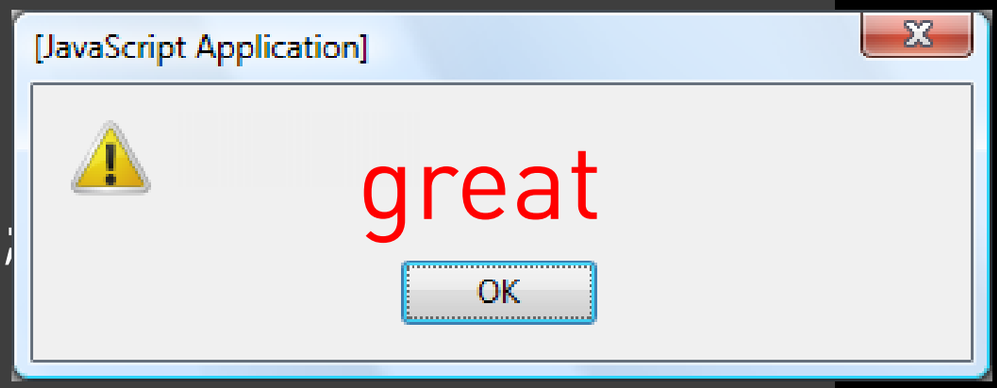
Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```



Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```

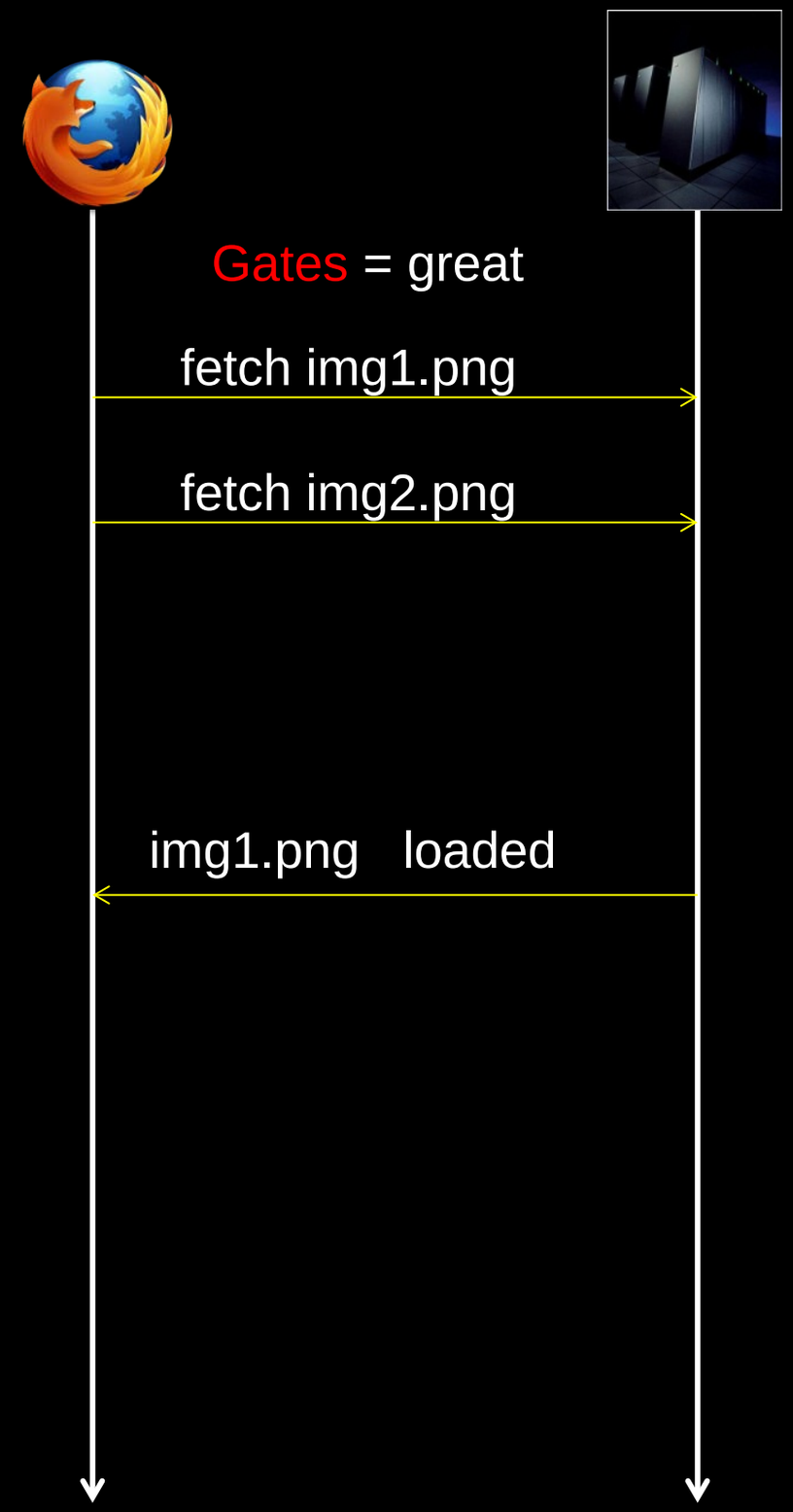


Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```

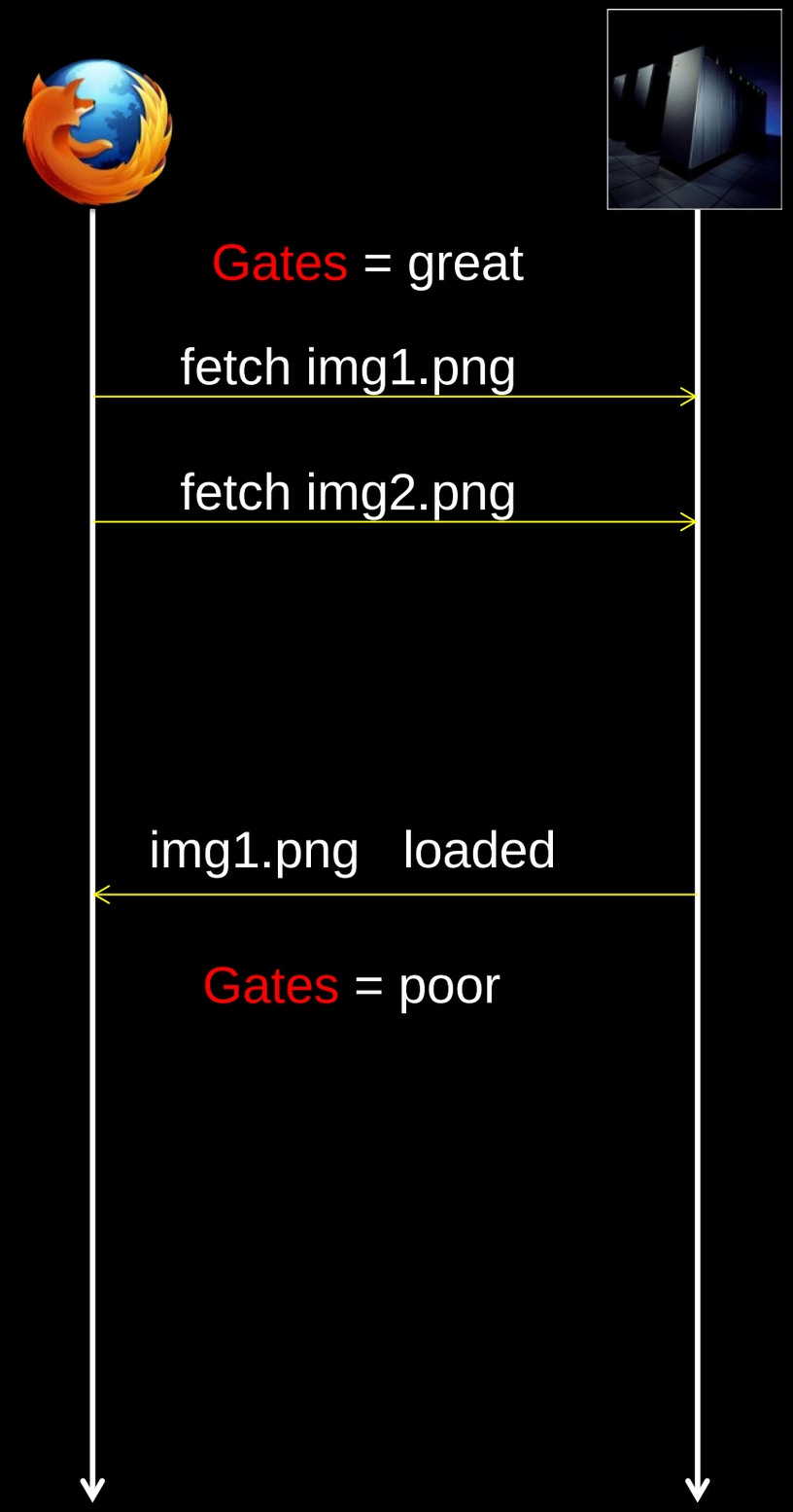
Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```



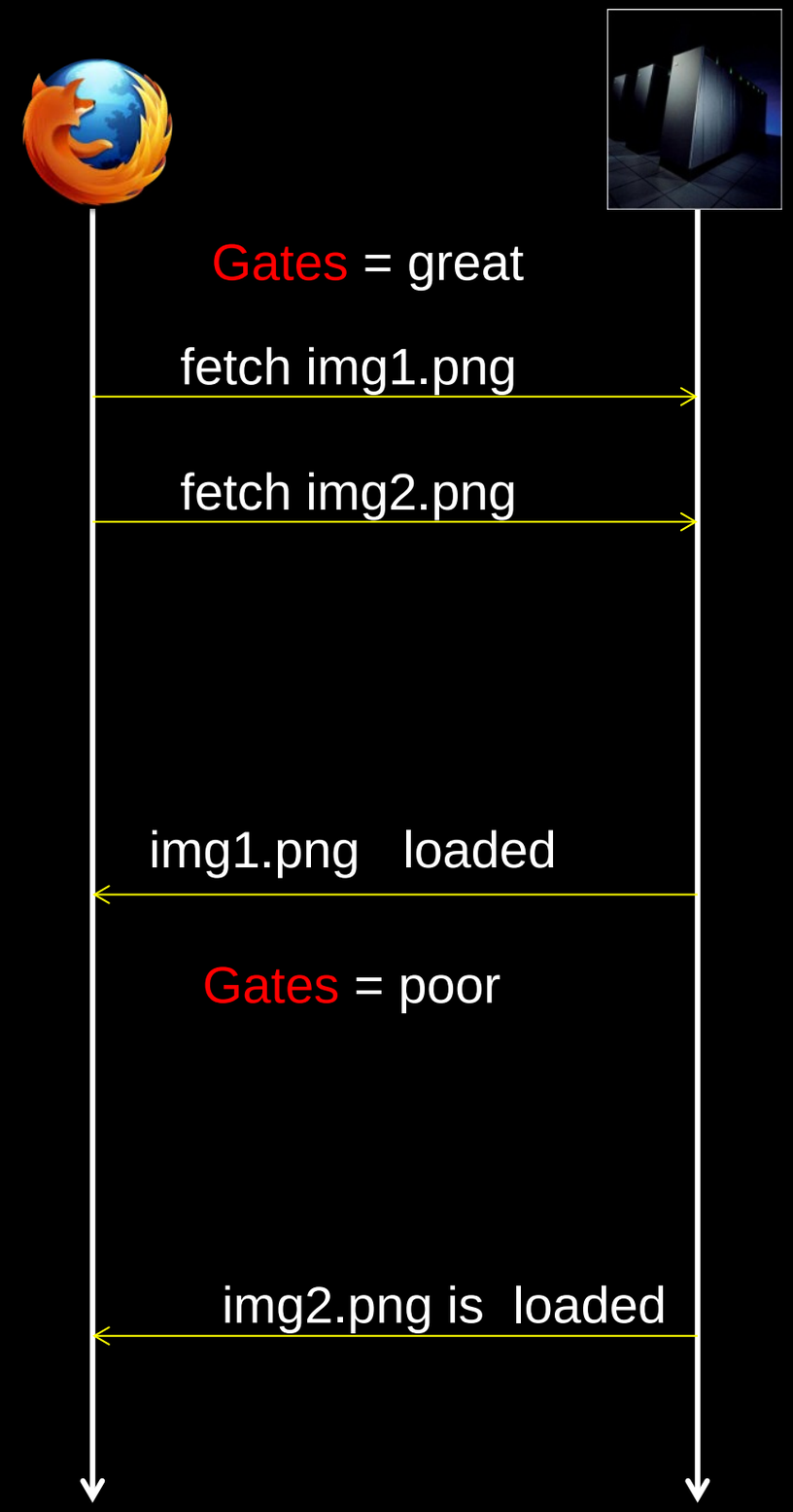
Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```



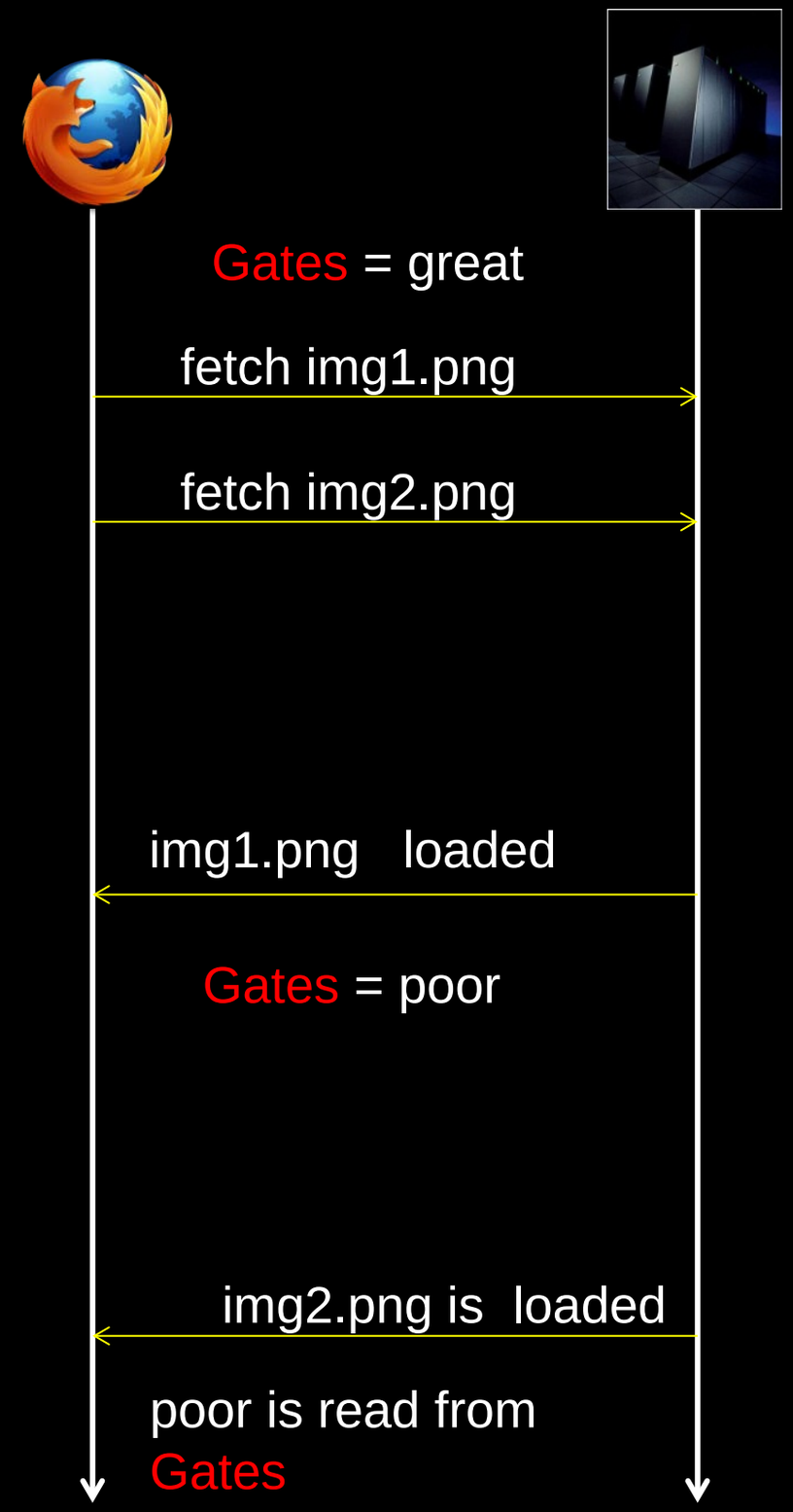
Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```



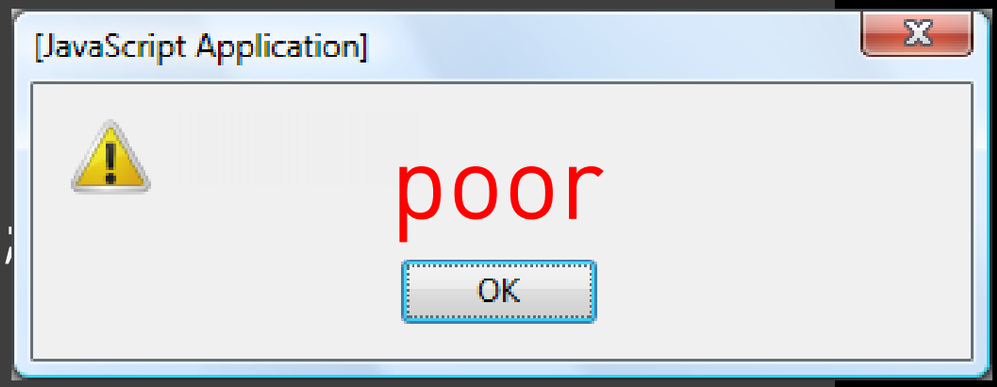
Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;</script>
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```



Data races: Web

```
<html>
<head></head>

<body>
<script>
var Gates = “great”;
</script>

<img src=“img1.png” onload=“Gates=‘poor’;”>
<img src=“img2.png” onload=“alert(Gates);”>

</body>
</html>
```



Gates = great

fetch img1.png

fetch img2.png

img1.png loaded

Gates = poor

img2.png is loaded

poor is read from Gates

EventRacer:
Concurrency Analysis of Event-Driven Apps

Publications

Effective Race Detection for Event-Driven Programs, OOPSLA'13

Race Detection for Web Applications, PLDI'12

Android Race Detection, M.Sc. Thesis, ETH

Stateless Model Checking for ER apps, PhD thesis, Aarhus

Tool Features

Formal Model of Concurrency

Scalable Graph Algorithms

Guaranteed Races

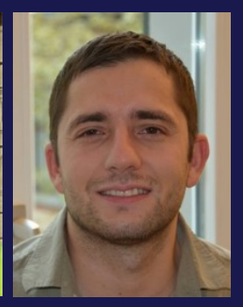
Bugs found and fixed

Works with Chromium/V8

People


Veselin Raychev


Pavol Bielik


Martin Vechev

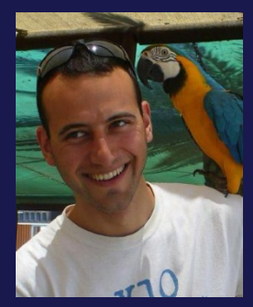

Manu Sridharan


Julian Dolby


Anders Moeller


Casper Jensen


Momchil Velikov


Boris Petrov

More info: http://www.eventracer.org
http://www.eventracer.org/android

Department of Computer Science
ETH zürich

Dynamic Race Detection

15+ years of algorithms and optimizations

Massive amount of work on data race detectors

exploit structure for improved asymptotic complexity, parallelization, sampling, hardware acceleration, static optimizations, ...

Key Theorems

- If the analysis reports a race on a given execution, then the race exists
- If the analysis does not report a race on a given execution, then there is no race for that input state (the input state of the execution)

Recommended Reading

MIT's Cilk Race Detector for SP-graphs, ACM SPAA'97, M.Feng, C. Leiserson
Race Detection for 2D partial orders, ACM SPAA'15, D. Dimitrov, M. Vechev, V. Sarkar

FastTrack, ACM PLDI'09, S. Freund, C. Flanagan
EventRacer, ACM OOPSLA'13, V. Raychev, M. Sridharan, M. Vechev

Trend:
Interference shifts to interfaces

Increasingly, low-level functionality captured inside high level objects

util.concurrent

The gap

Classic data race detection

Uses read-write notion of conflict

Many optimizations

Interference at the interface

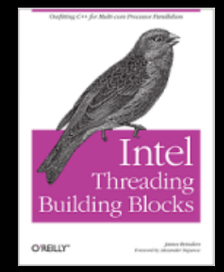
Current detectors increasingly ineffective in handling real-world programs

Wanted

New kind of race detection

Uses higher level notion of conflict

Many optimizations



Interference at the interface

How do we bridge the gap in an elegant and clean manner?

Most of what follows is based on:

Commutativity Race Detection, ACM PLDI'14

Dimitar Dimitrov

Veselin Raychev

Eric Koskinen

Martin Vechev

Commutativity Race

two high-level operations do not commute

and are not ordered by the program

capture

interference between
high-level operations

Commutativity Races

```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```

Commutativity Races

```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```

put(‘a’,‘tom’)/nil

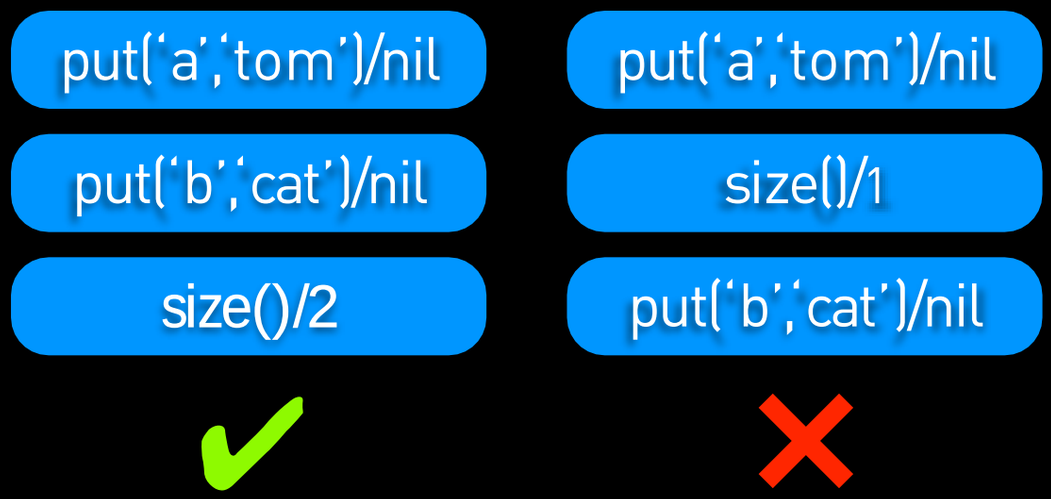
put(‘b’,‘cat’)/nil

size()/2

✔

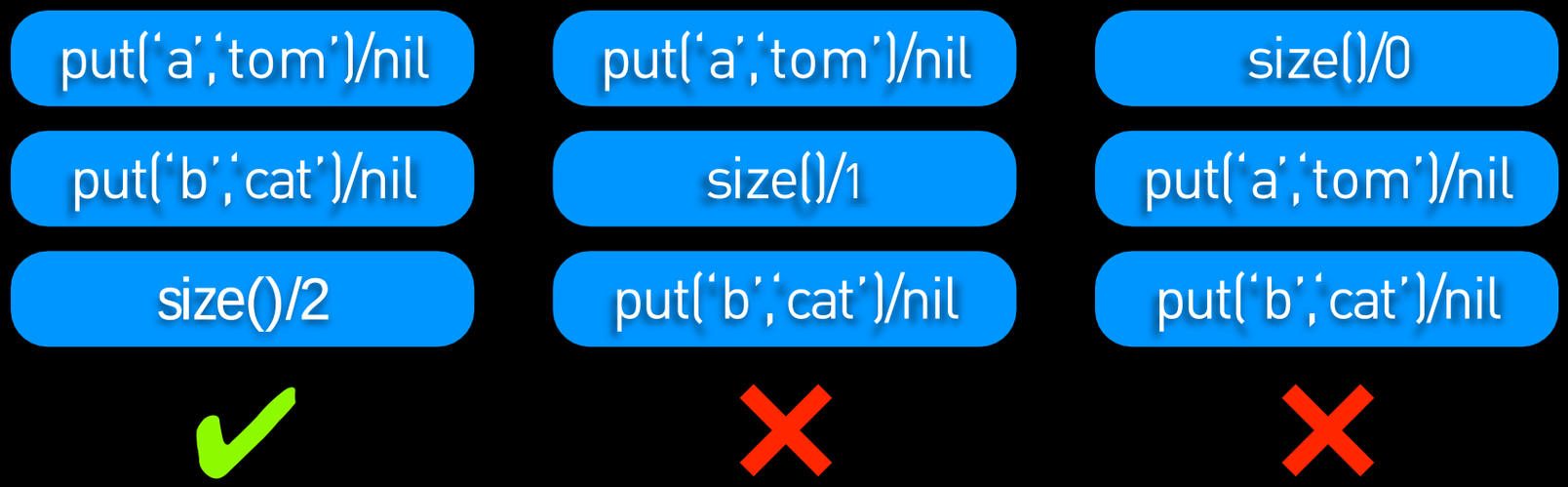
Commutativity Races

```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```



Commutativity Races

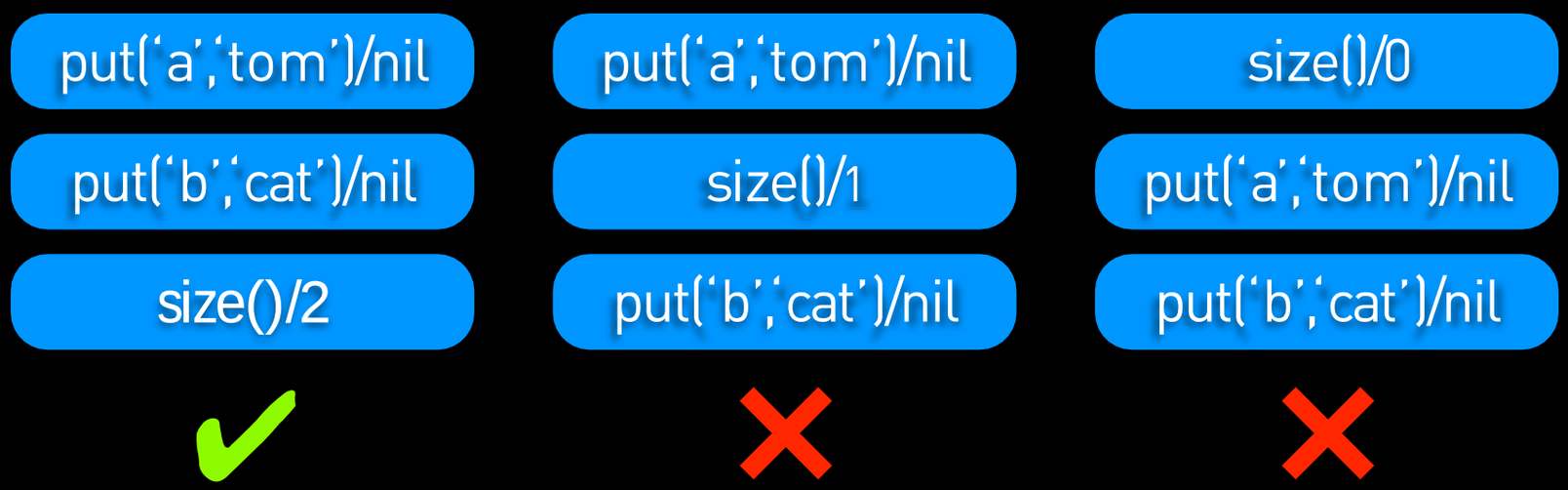
```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```



Commutativity Races

```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```

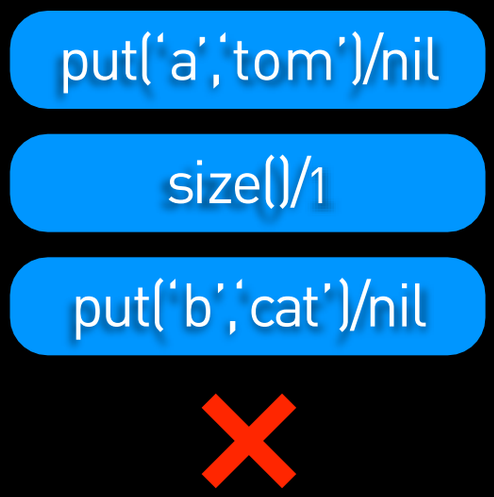
1. *put and size do not commute*
2. *are not ordered by the program*

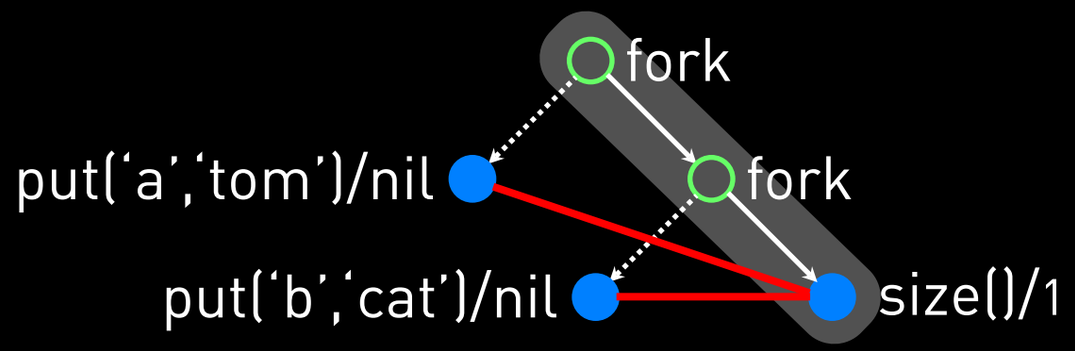


Commutativity Races

```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```

1. *put and size do not commute*
2. *are not ordered by the program*

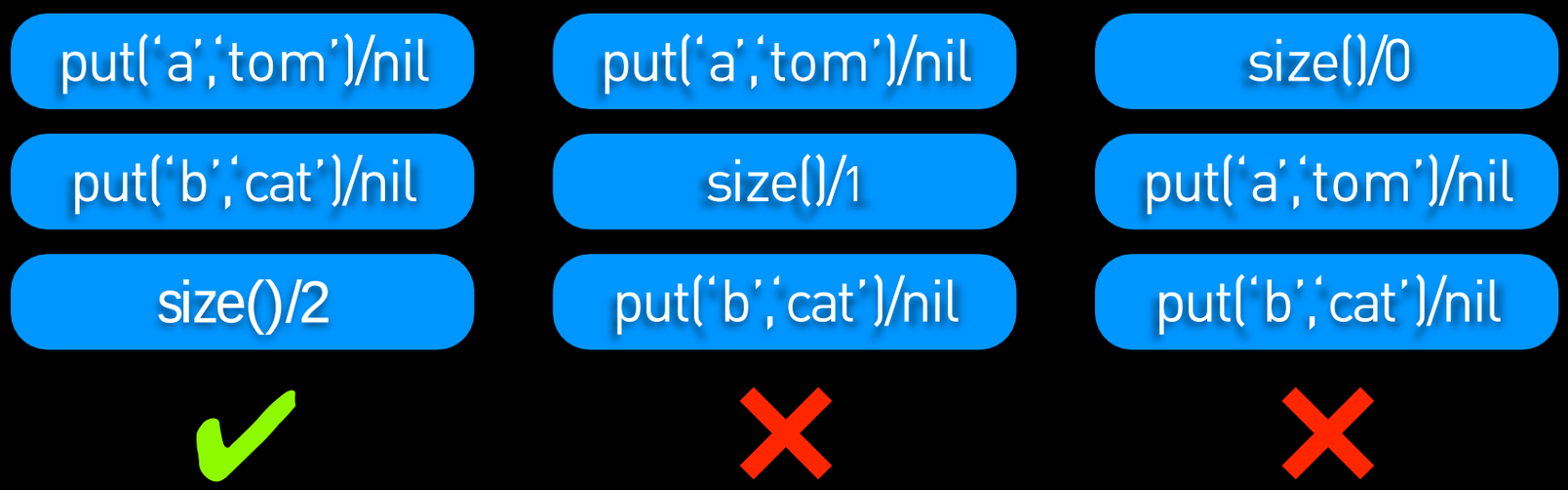




Commutativity Races

```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```

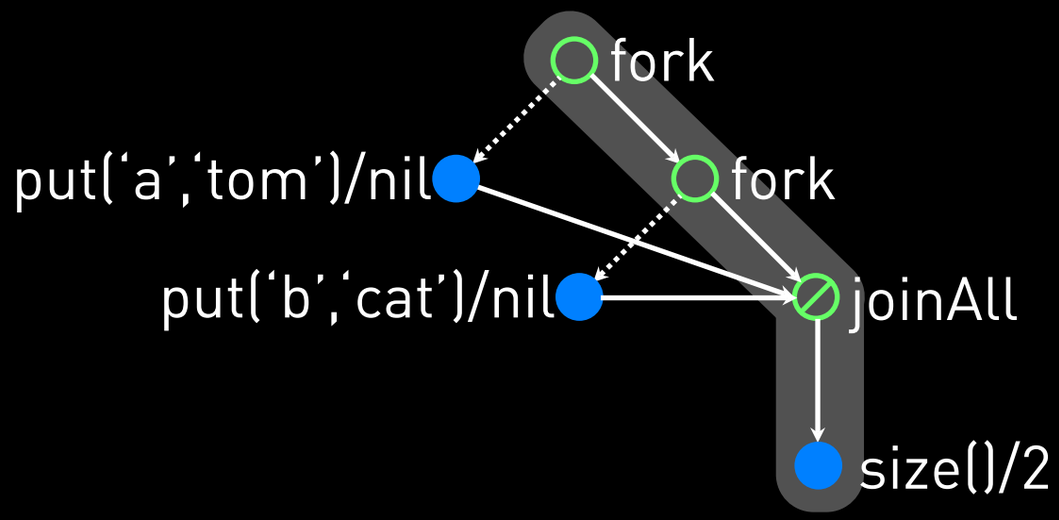
1. *put and size do not commute*
2. *are not ordered by the program*



No Commutativity Races

```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }

joinAll
new Array [data.size()];
```

1. *put and size do not commute*
2. *are ordered by the program*



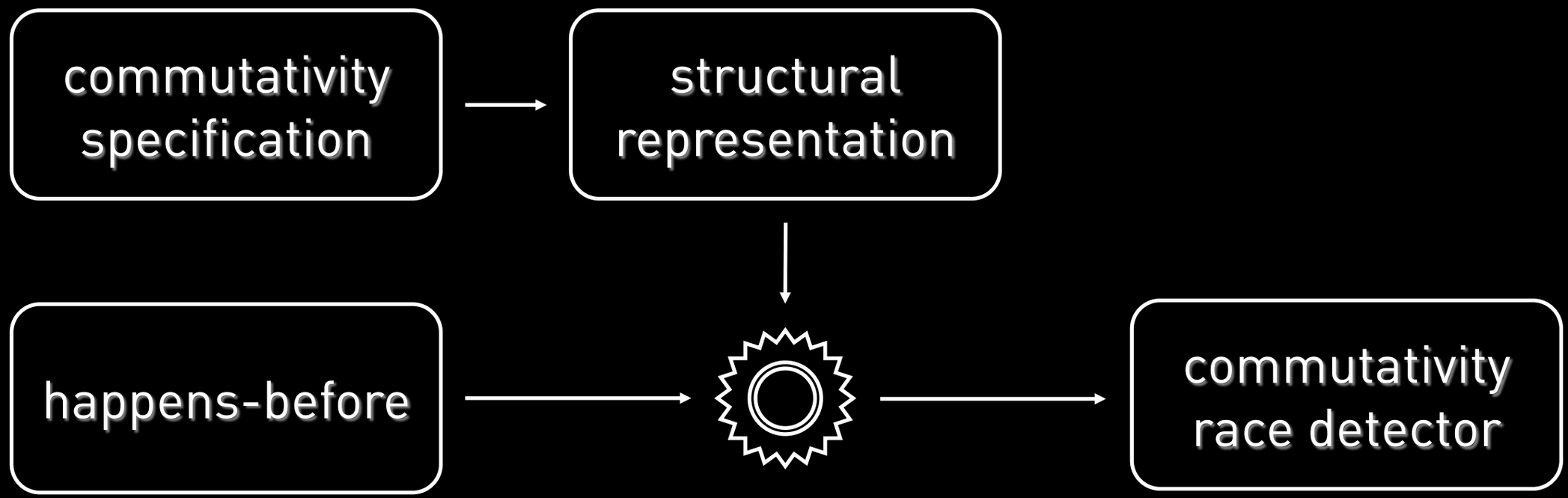
Principles of Analysis

To detect interference between concurrent events

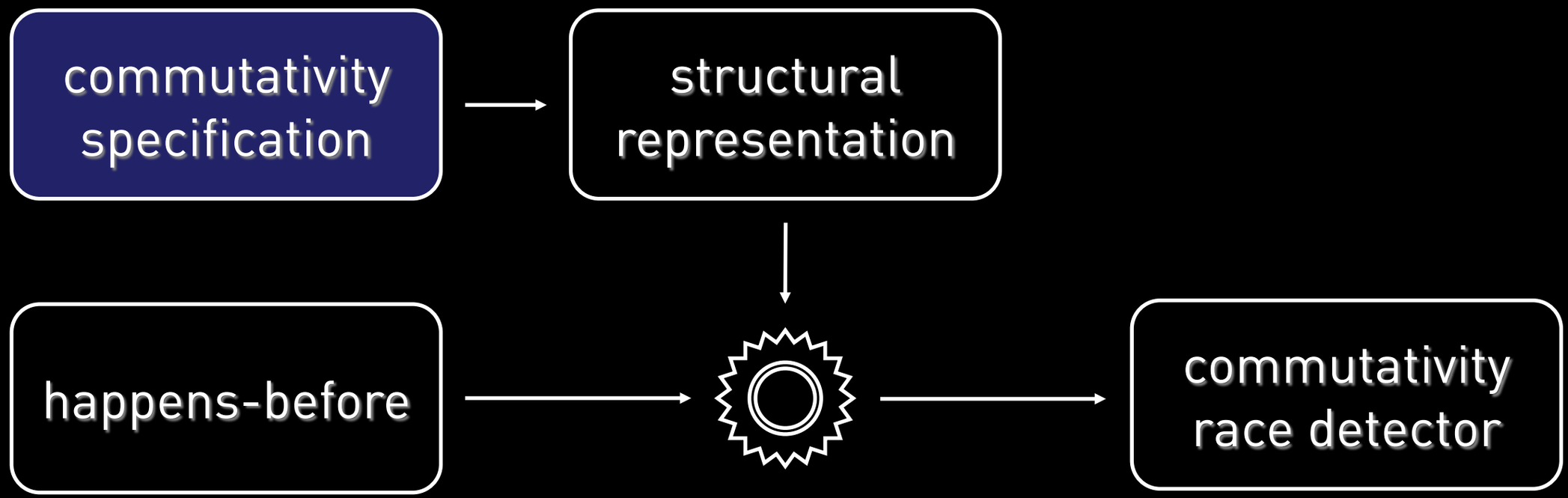
Time: Need to efficiently capture event ordering

Space: Interference between operations

Commutativity Race Detection



Commutativity Race Detection



Hashmap commutativity spec

$put(k_1,v_1)/r_1 \bowtie size()/n_2 \iff (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)$

$put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 \iff k_1 \neq k_2 \vee (v_1 = r_1 \wedge v_2 = r_2)$

$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \iff k_1 \neq k_2 \vee v_1 = r_1$

$size()/n_1 \bowtie size()/n_2 \iff true$

$size()/n_1 \bowtie get(k_2)/v_2 \iff true$

$get(k_1)/v_1 \bowtie get(k_2)/v_2 \iff true$

Hashmap commutativity spec

$put(k_1,v_1)/r_1 \bowtie size()/n_2 \iff (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)$

$put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 \iff k_1 \neq k_2 \vee (v_1 = r_1 \wedge v_2 = r_2)$

$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \iff k_1 \neq k_2 \vee v_1 = r_1$

$size()/n_1 \bowtie size()/n_2 \iff true$

$size()/n_1 \bowtie get(k_2)/v_2 \iff true$

$get(k_1)/v_1 \bowtie get(k_2)/v_2 \iff true$

Hashmap commutativity spec

$put(k_1,v_1)/r_1 \bowtie size()/n_2 \iff (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)$

$put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 \iff k_1 \neq k_2 \vee (v_1 = r_1 \wedge v_2 = r_2)$

$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \iff k_1 \neq k_2 \vee v_1 = r_1$

$size()/n_1 \bowtie size()/n_2 \iff true$

$size()/n_1 \bowtie get(k_2)/v_2 \iff true$

$get(k_1)/v_1 \bowtie get(k_2)/v_2 \iff true$

put(‘a’,‘tom’)/nil ⋈ size()/1

put(‘b’,‘lol’)/cat ⋈ size()/1

Commutativity Specifications

Can be large and complex

Example: ArrayList (part of the spec)

$r_1 = s_1.\texttt{remove\_at}(i_1)$	$s_2.\texttt{add\_at}(i_2, v_2)$	$(i_1 < i_2 \wedge s_1[i_2] = v_2) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = v_2) \vee$ $(i_1 > i_2 \wedge s_1[i_1 - 1] = s_1[i_1])$
	$r_2 = s_2.\texttt{get}(i_2)$	$(i_1 < i_2 \wedge s_1[i_2] = s_1[i_2 + 1]) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = s_1[i_2 + 1]) \vee$ $i_1 > i_2$
	$r_2 = s_2.\texttt{indexOf}(v_2)$	$\neg(\exists i : s_1[i] = v_2) \vee$ $(\exists i < i_1 : s_1[i] = v_2) \vee$ $(\neg(\exists i < i_1 : s_1[i] = v_2) \wedge s_1[i_1] = v_2 \wedge i_1 < \lvert s_1\rvert - 1 \wedge s_1[i_1 + 1] = v_2)$
	$r_2 = s_2.\texttt{lastIndexOf}(v_2)$	$\neg(\exists i : s_1[i] = v_2) \vee$ $((\exists i < i_1 : s_1[i] = v_2) \wedge \neg(\exists i \geq i_1 : s_1[i] = v_2))$
	$r_2 = s_2.\texttt{remove\_at}(i_2)$	$(i_1 < i_2 \wedge s_1[i_2] = s_1[i_2 + 1]) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = s_1[i_2 + 1]) \vee$ $(\lvert s_1\rvert - 1 > i_1 > i_2 \wedge s_1[i_1] = s_1[i_1 + 1])$
	$s_2.\texttt{remove\_at}(i_2)$	$(i_1 < i_2 \wedge s_1[i_2] = s_1[i_2 + 1]) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = s_1[i_2 + 1]) \vee$ $(\lvert s_1\rvert - 1 > i_1 > i_2 \wedge s_1[i_1] = s_1[i_1 + 1])$
	$r_2 = s_2.\texttt{set}(i_2, v_2)$	$(i_1 < i_2 \wedge s_1[i_2] = s_1[i_2 + 1] = v_2) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = s_1[i_2 + 1] = v_2) \vee$ $i_1 > i_2$
	$s_2.\texttt{set}(i_2, v_2)$	$(i_1 < i_2 \wedge s_1[i_2] = s_1[i_2 + 1] = v_2) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = s_1[i_2 + 1] = v_2) \vee$ $i_1 > i_2$

"Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures", Kim & Rinard, ACM PLDI'11

Commutativity Race Detection

Naïve Algorithm

Initially `Seen` = ε

Then during execution, for each operation `op` do:

```
foreach b ∈ Seen {
  if not(op ⋈ b) and (op || b)
    report ‘commutativity race’
}
Seen = Seen • op
```

Commutativity Race Detection

Naïve Algorithm

Initially `Seen = ε`

Commute?

Then during execution, for each operation `op` do:

```
foreach b ∈ Seen {
  if not(op ⋈ b) and (op || b)
    report ‘commutativity race’
}
Seen = Seen • op
```

Commutativity Race Detection

Naïve Algorithm

Initially `Seen` = ε

Commute?

Ordered?

Then during execution, for each operation `op` do:

```
foreach b ∈ Seen {
  if not(op ⋈ b) and (op || b)
    report ‘commutativity race’
}
Seen = Seen • op
```

Commutativity Race Detection

Naïve Algorithm

Initially `Seen` = ε

Commute?

Ordered?

Then during execution, for each operation `op` do:

```
foreach b ∈ Seen {
  if not(op ⋈ b) and (op || b)
    report ‘commutativity race’
}
Seen = Seen • op
```

Two problems:

1. Extremely inefficient: worst-case O(`Seen`) time per-operation

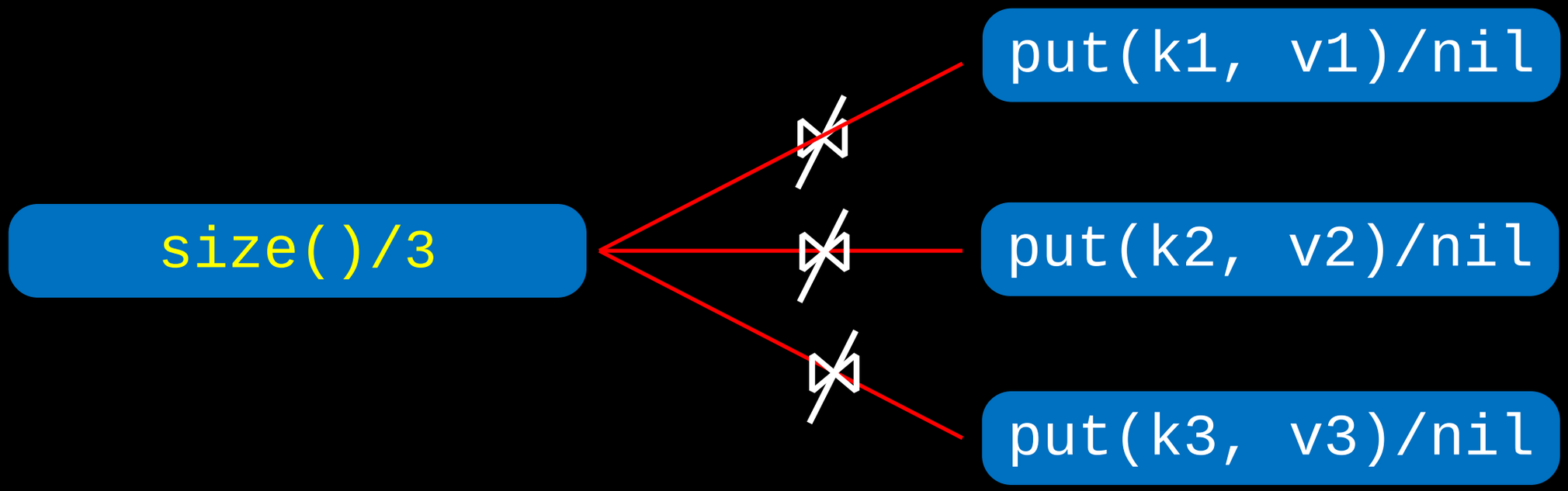
2. Does not explore commutativity sharing between operations

Commutativity Race Detection

Naïve Algorithm

For trace: `put(k1, v1)/nil • put(k2, v2)/nil • put(k3, v3)/nil • size()/3`

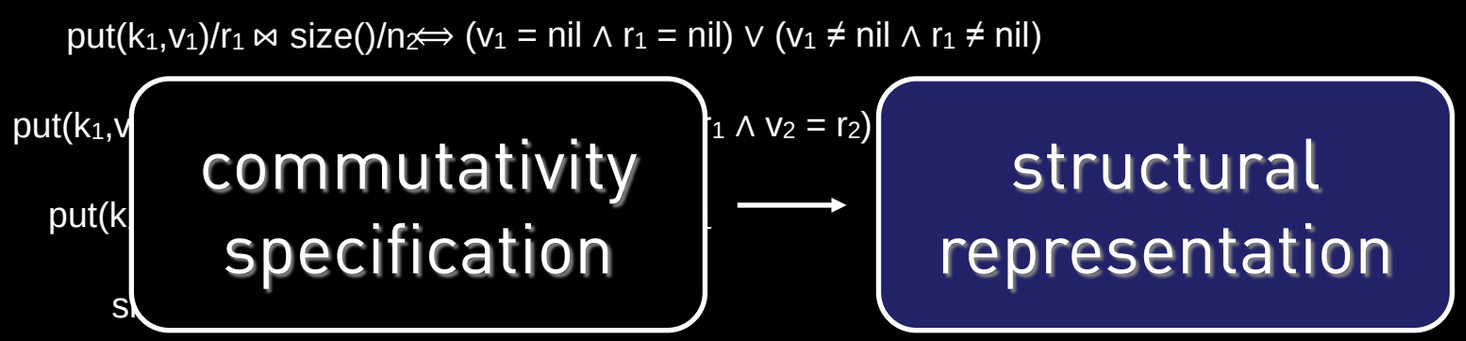
Upon encountering `size()/3` , we will perform 3 checks:



Yet, what matters is whether a concurrent resize has occurred

Wanted: leverage sharing between observed operations

Commutativity Race Detection



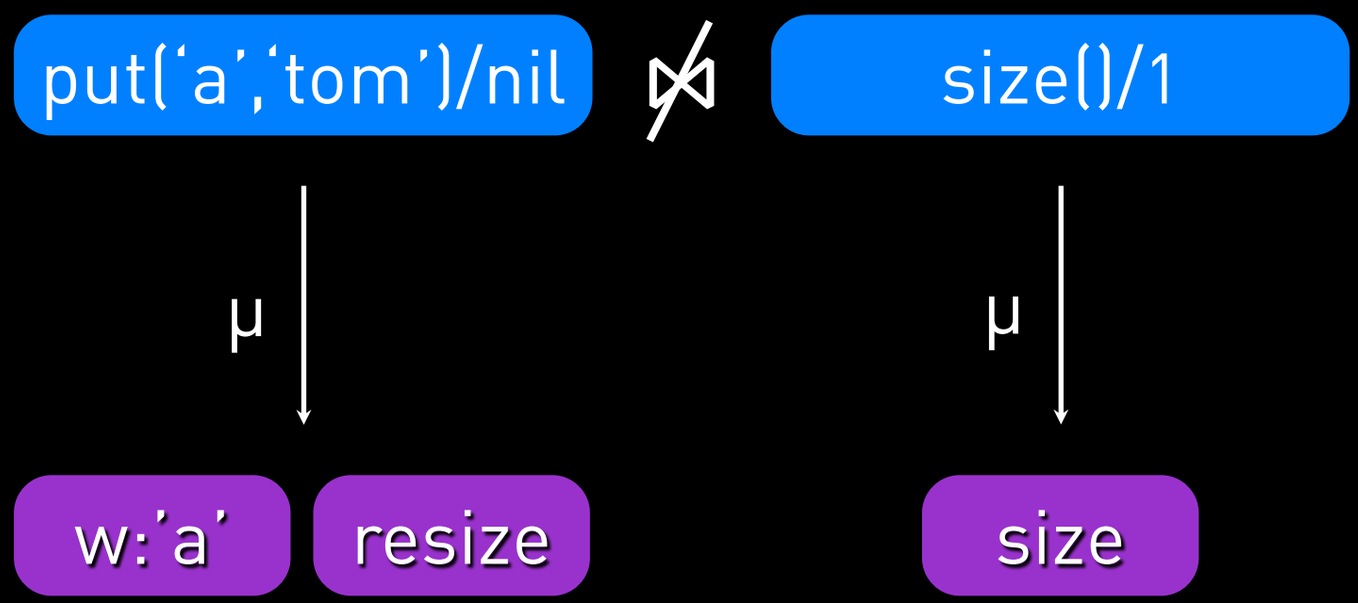
Micro-ops representation

operations

put(‘a’,‘tom’)/nil ⋈ size()/1

Micro-ops representation

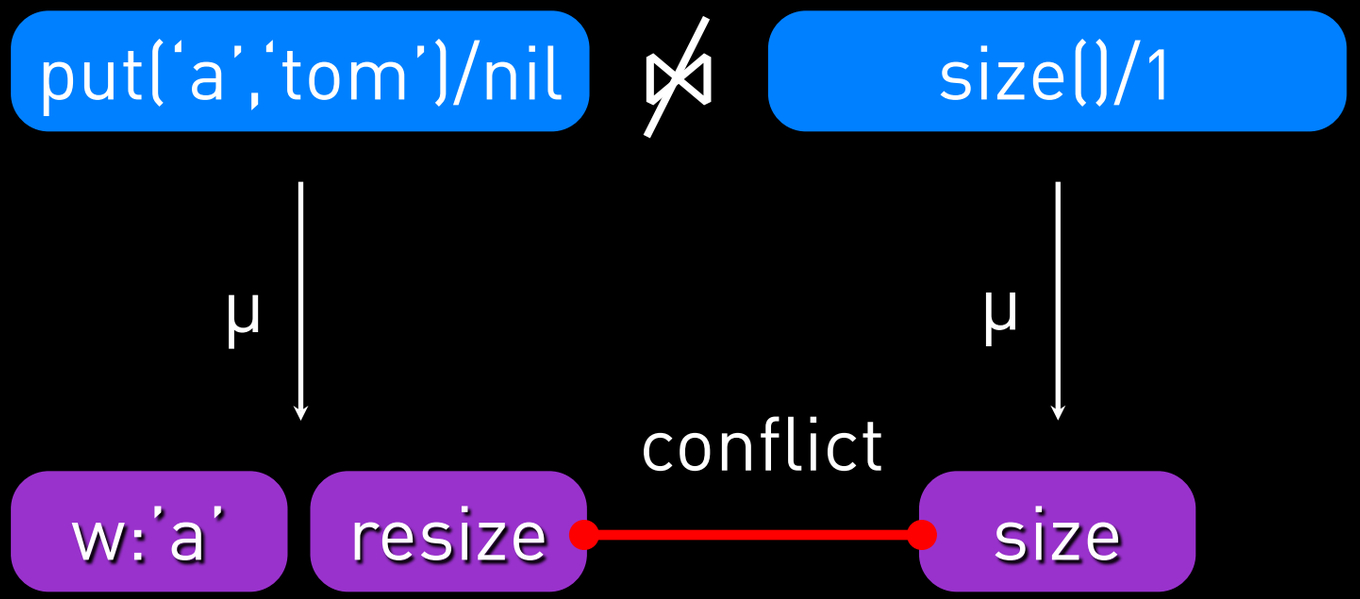
operations



μ-operations

Micro-ops representation

operations



μ-operations

From logic to μ-operations

$put(k_1,v_1)/r_1 \bowtie size()/n_2 \quad \Leftrightarrow (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)$

$put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 \quad \Leftrightarrow k1 \neq k2 \vee (v_1 = r_1 \wedge v_2 = r_2)$

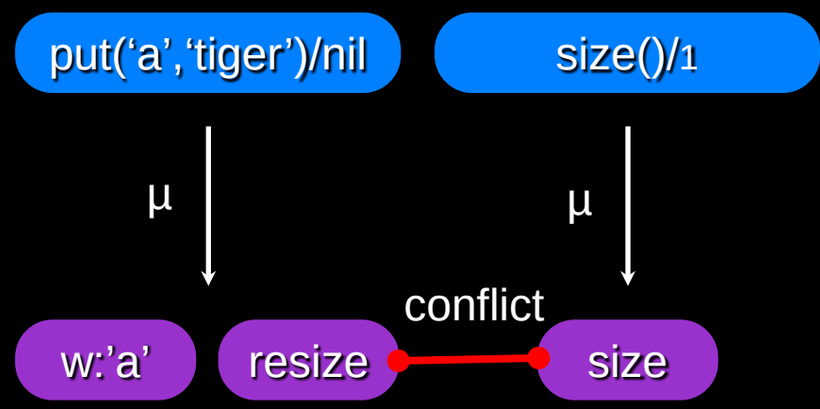
$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \quad \Leftrightarrow k1 \neq k2 \vee v_1 = r_1$

$size()/n_1 \bowtie size()/n_2 \quad \Leftrightarrow true$

$size()/n_1 \bowtie get(k_2)/v_2 \quad \Leftrightarrow true$

$get(k_1)/v_1 \bowtie get(k_2)/v_2 \quad \Leftrightarrow true$

VS.



From logic to μ-operations

$put(k_1,v_1)/r_1 \bowtie size()/n_2 \quad \Leftrightarrow (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)$

$put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 \quad \Leftrightarrow k1 \neq k2 \vee (v_1 = r_1 \wedge v_2 = r_2)$

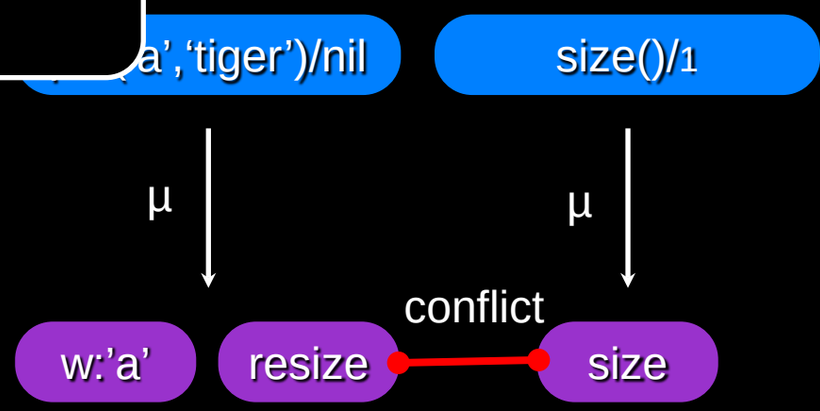
$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \quad \Leftrightarrow k1 \neq k2 \vee v_1 = r_1$

$size()/n_1 \bowtie size()/n_2$

$size()/n_1 \bowtie get(k_2)/v_2$

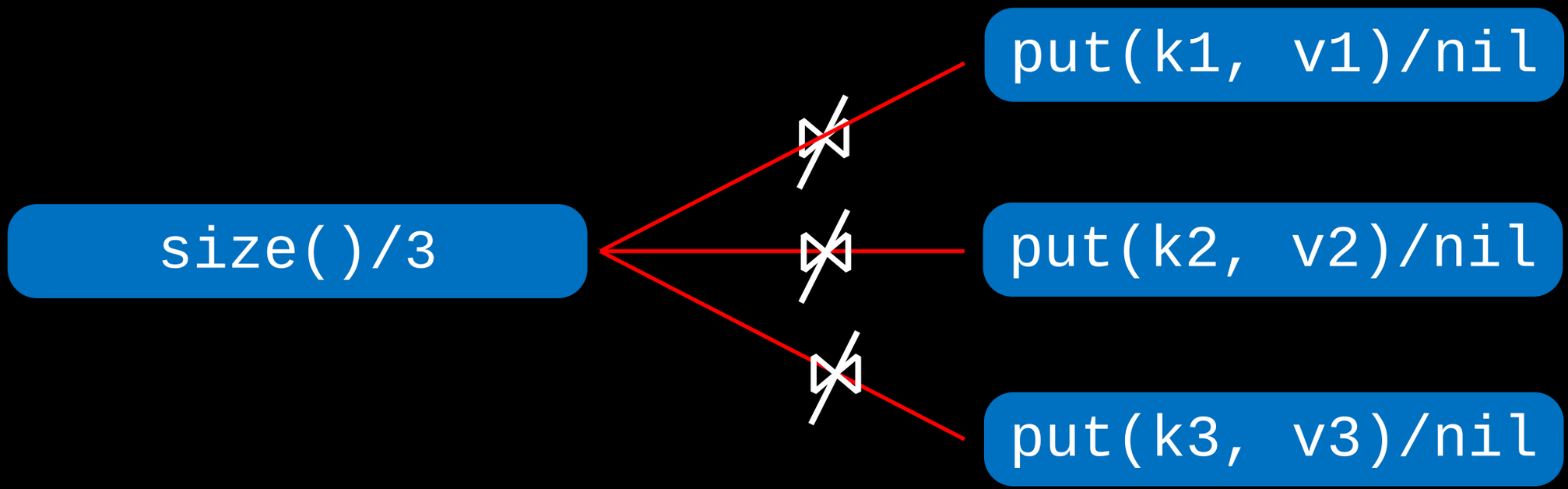
$get(k_1)/v_1 \bowtie get(k_2)/v_2$

commutativity compiler



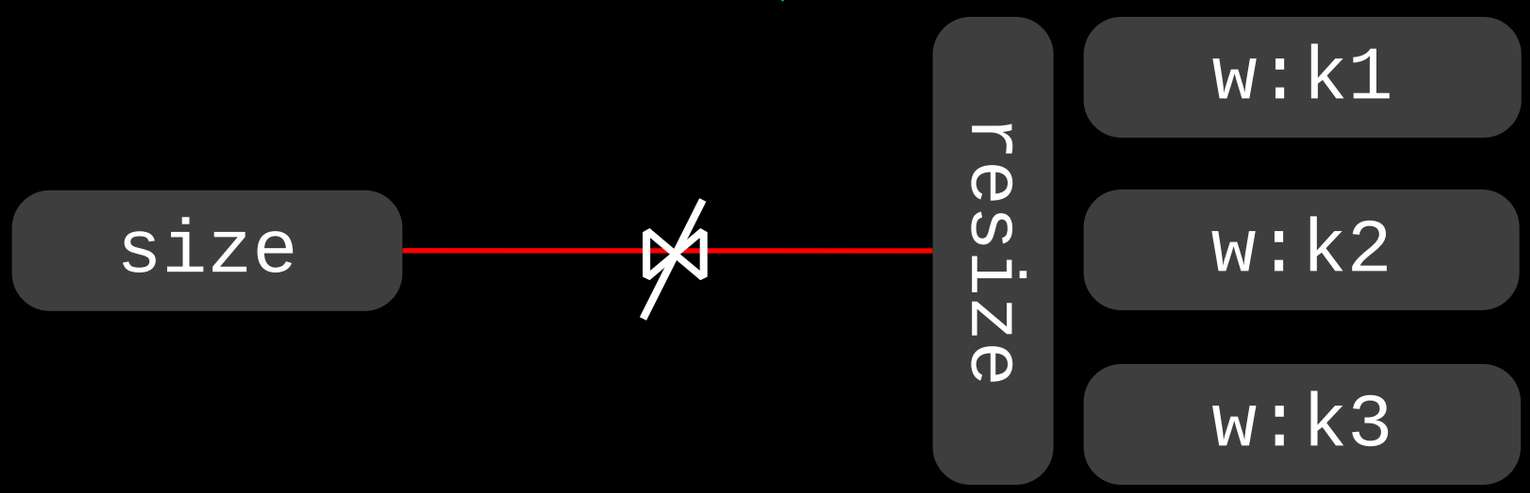
Commutativity Compiler

Benefits

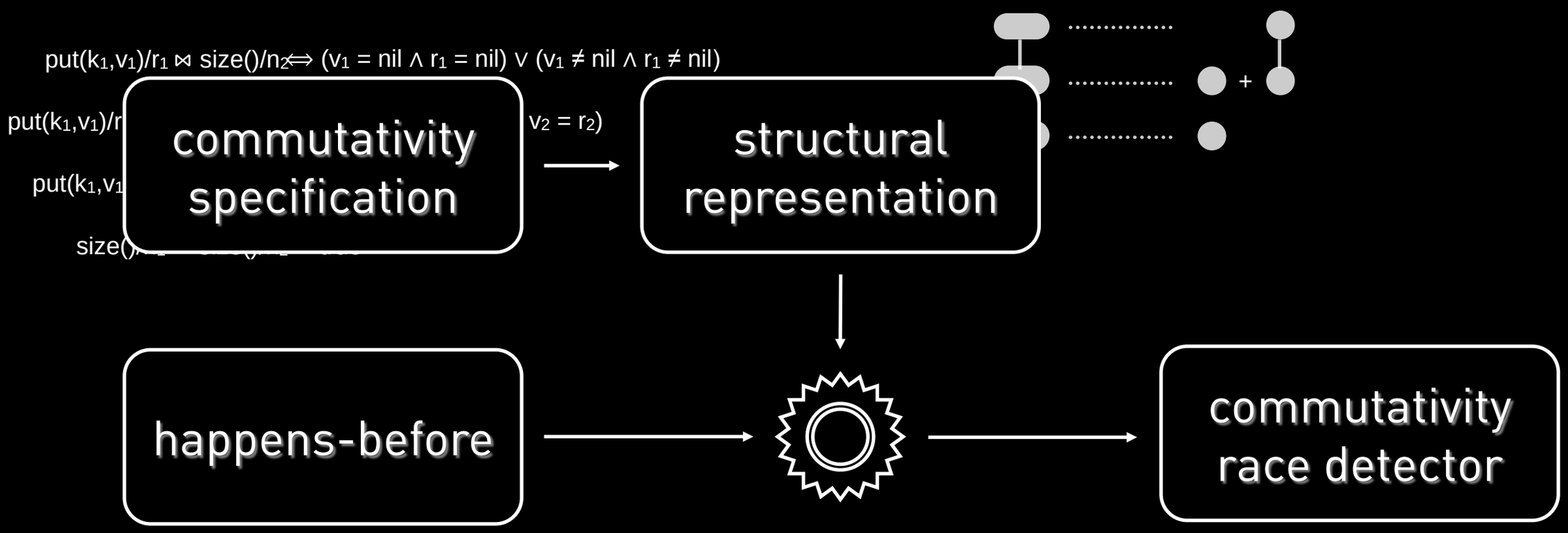


conflict on operations

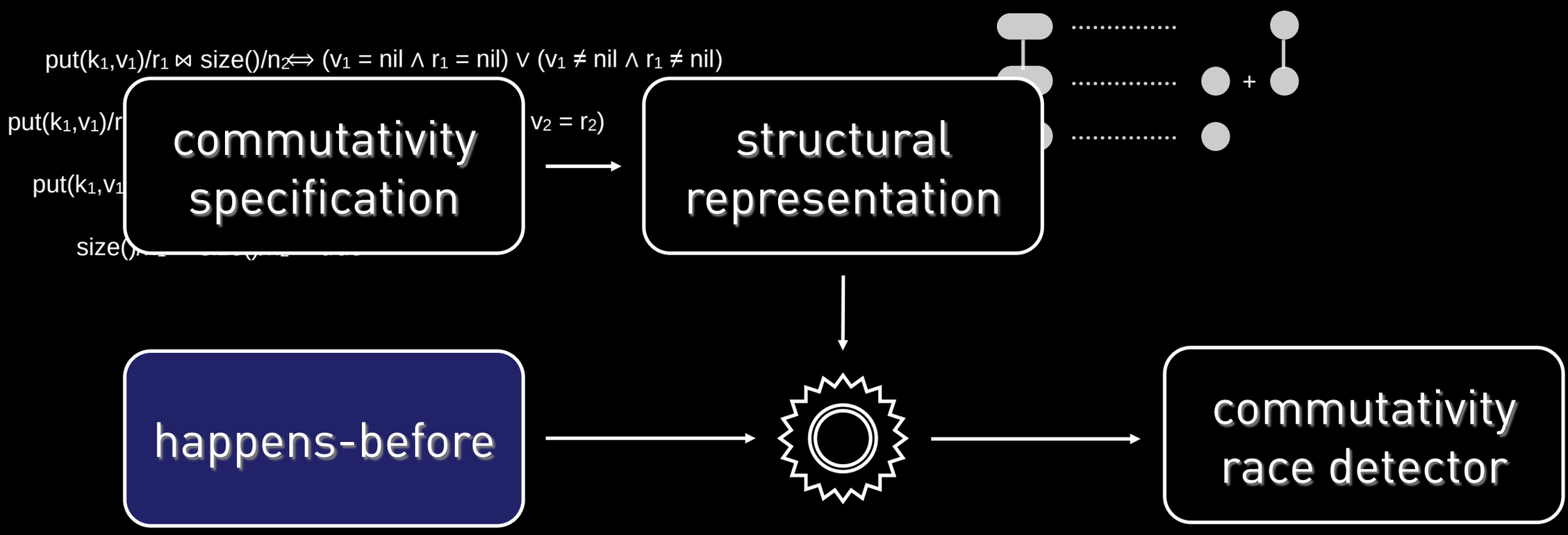
conflict on μ-operations



Commutativity Race Detection



Commutativity Race Detection



Next: brief review of vector clocks

Vector Clocks

Review

A vector clock VC is a map: $VC \in Threads \rightarrow Nat$

Example: if we have 3 threads, a vector clock can be: $\langle 2,0,1 \rangle$

Intuitively, vector clocks are associated with operations and capture ordering (or lack of) between these operations

Vector Clocks

Operations

Compare

$\langle 2,0,1 \rangle \sqsubseteq \langle 2,0,2 \rangle$: operation with clock $\langle 2,0,1 \rangle$ happens before operation with clock $\langle 2,0,2 \rangle$

$\langle 2,0,1 \rangle \not\sqsubseteq \langle 1,0,2 \rangle$: operation with clock $\langle 2,0,1 \rangle$ is unordered with operation with clocks $\langle 1,0,2 \rangle$

Vector Clocks

Operations

Compare

$\langle 2,0,1 \rangle \sqsubseteq \langle 2,0,2 \rangle$: operation with clock $\langle 2,0,1 \rangle$ happens before operation with clock $\langle 2,0,2 \rangle$
$\langle 2,0,1 \rangle \not\sqsubseteq \langle 1,0,2 \rangle$: operation with clock $\langle 2,0,1 \rangle$ is unordered with operation with clocks $\langle 1,0,2 \rangle$

Join

$\langle 2,0,1 \rangle \sqcup \langle 1,0,2 \rangle = \langle 2,0,2 \rangle$: used to obtain the most up to date information when combining effects of two operations

Vector Clocks

Operations

Compare

$\langle 2,0,1\rangle \sqsubseteq \langle 2,0,2\rangle$: operation with clock $\langle 2,0,1\rangle$ happens before operation with clock $\langle 2,0,2\rangle$
$\langle 2,0,1\rangle \not\sqsubseteq \langle 1,0,2\rangle$: operation with clock $\langle 2,0,1\rangle$ is unordered with operation with clocks $\langle 1,0,2\rangle$

Join

$\langle 2,0,1\rangle \sqcup \langle 1,0,2\rangle = \langle 2,0,2\rangle$: used to obtain the most up to date information when combining effects of two operations

Increment

`inc` ($\langle 2,0,1\rangle$, 2) results in vector clocks $\langle 2,0,2\rangle$: it advances time for thread 2. Done upon encountering synchronization operations (e.g., fork, lock, etc)

Vector Clocks

Operations

Compare

$\langle 2,0,1 \rangle \sqsubseteq \langle 2,0,2 \rangle$: operation with clock $\langle 2,0,1 \rangle$ happens before operation with clock $\langle 2,0,2 \rangle$
$\langle 2,0,1 \rangle \not\sqsubseteq \langle 1,0,2 \rangle$: operation with clock $\langle 2,0,1 \rangle$ is unordered with operation with clocks $\langle 1,0,2 \rangle$

Join

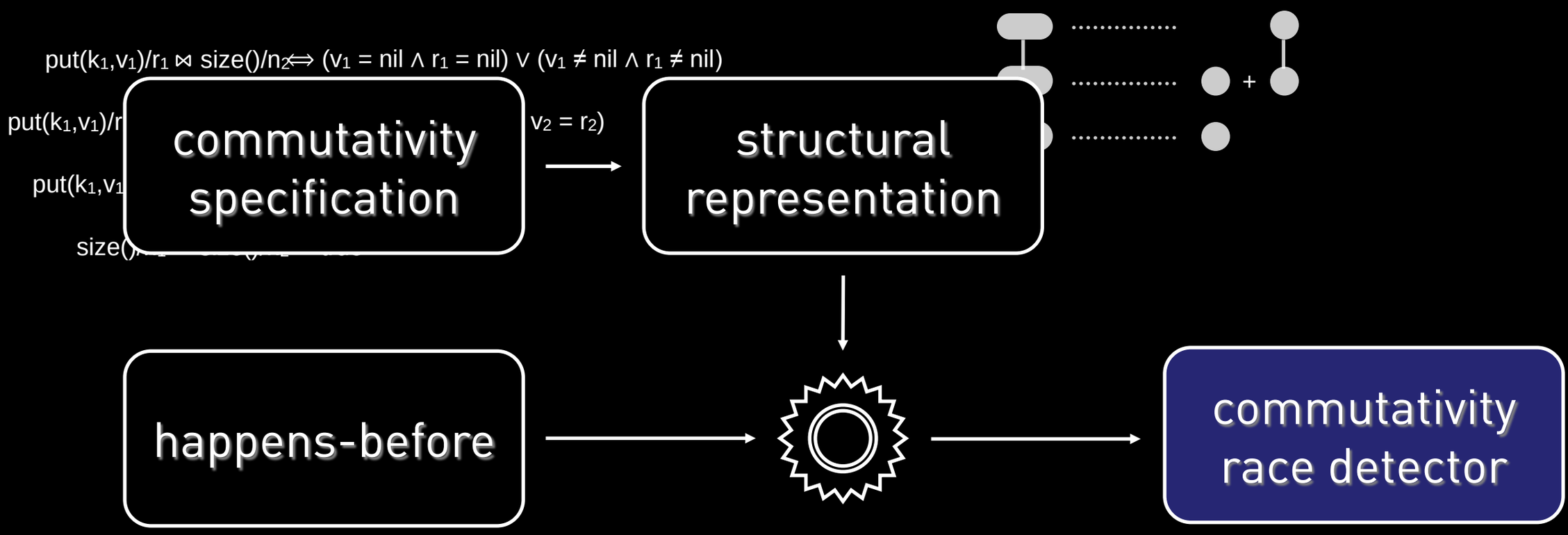
$\langle 2,0,1 \rangle \sqcup \langle 1,0,2 \rangle = \langle 2,0,2 \rangle$: used to obtain the most up to date information when combining effects of two operations

Increment

`inc ( ⟨2,0,1⟩ , 2 )` results in vector clocks $\langle 2,0,2 \rangle$: it advances time for thread 2. Done upon encountering synchronization operations (e.g., fork, lock, etc)

Vector clocks form a lattice

Commutativity Race Detection

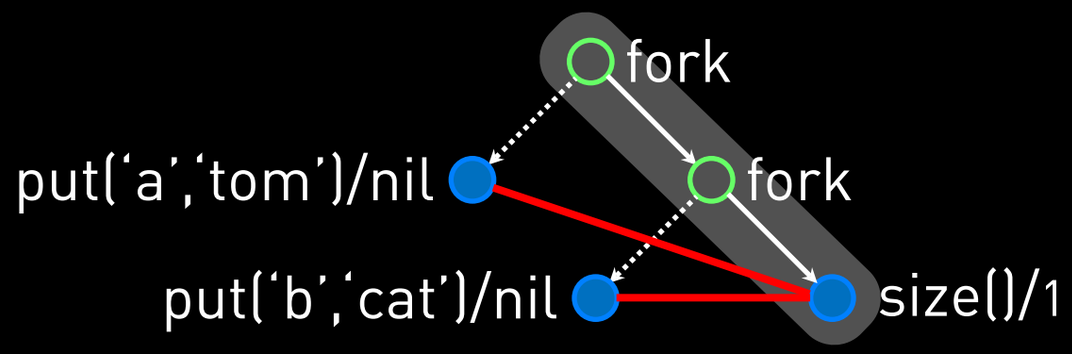


Key idea: associate vector clocks with μ-operations

Commutativity Races

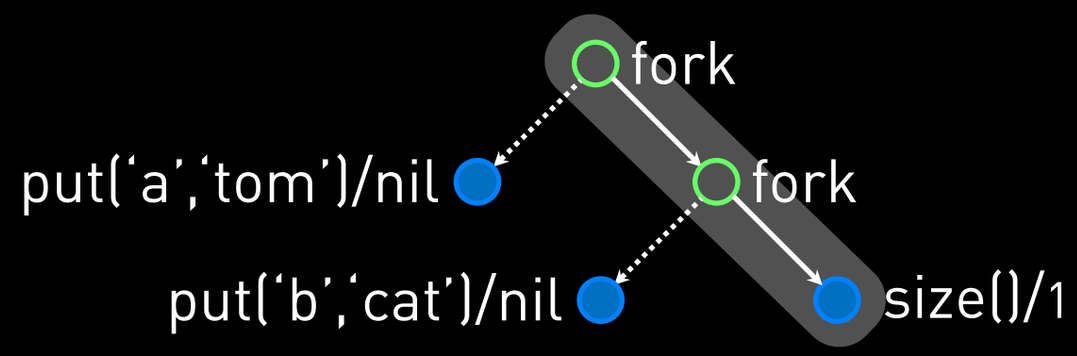
```
ConcurrentHashMap data = new ConcurrentHashMap();

for (String key : keys)
   fork {
      data.put(key, Value.compute(key));
   }


new Array [data.size()];
```

1. *put and size do not commute*
2. *are not ordered by the program*



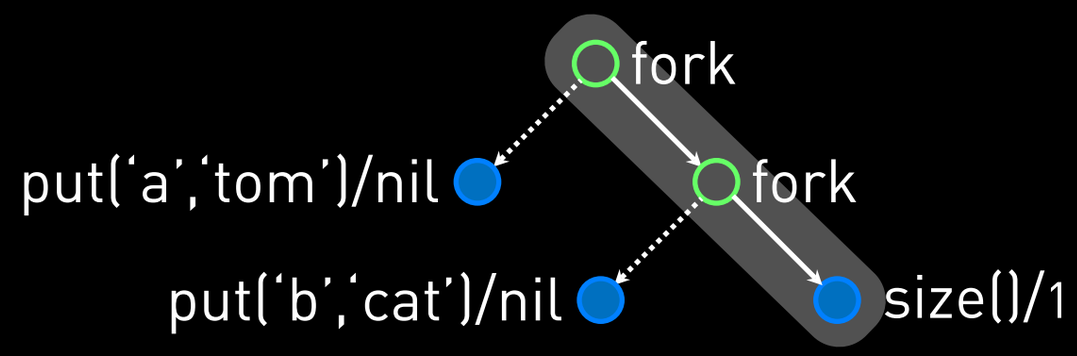
An example

trace

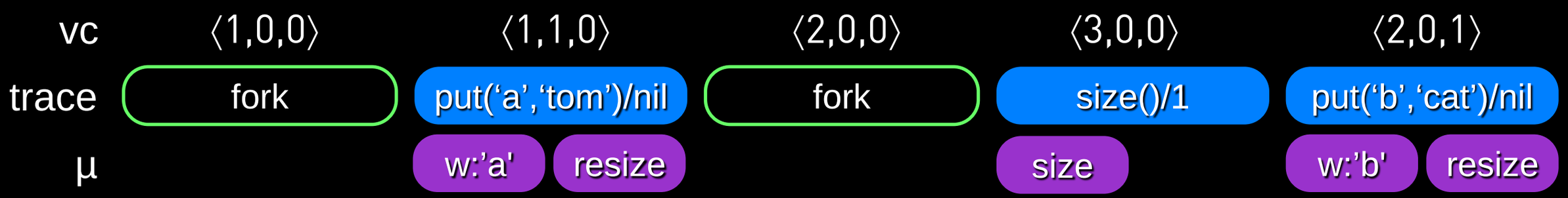


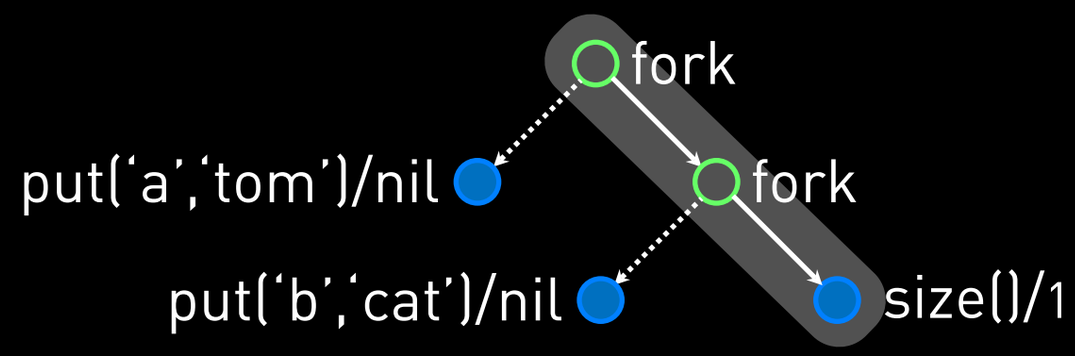
An example


trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

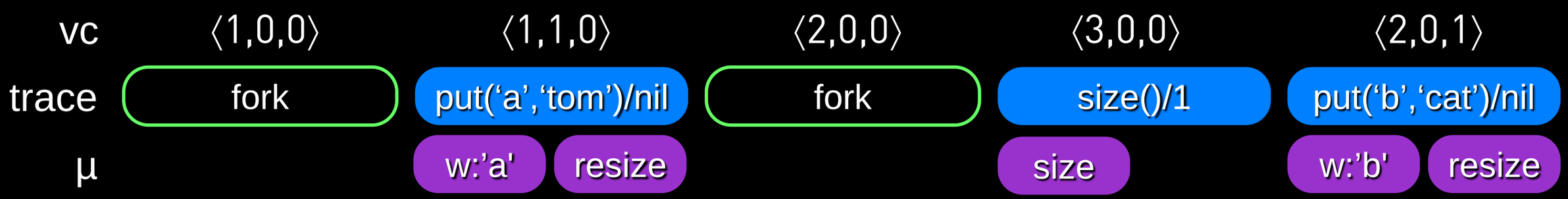

fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

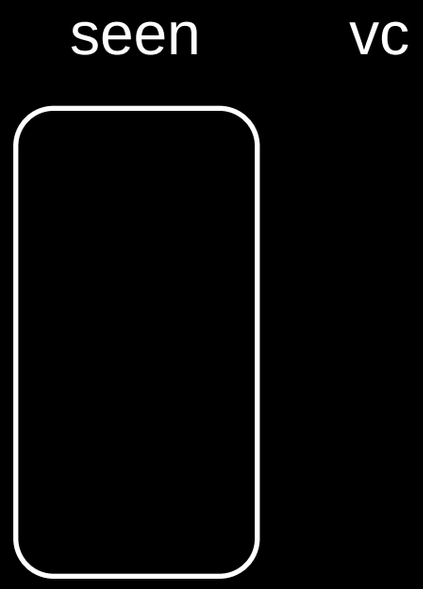
An example

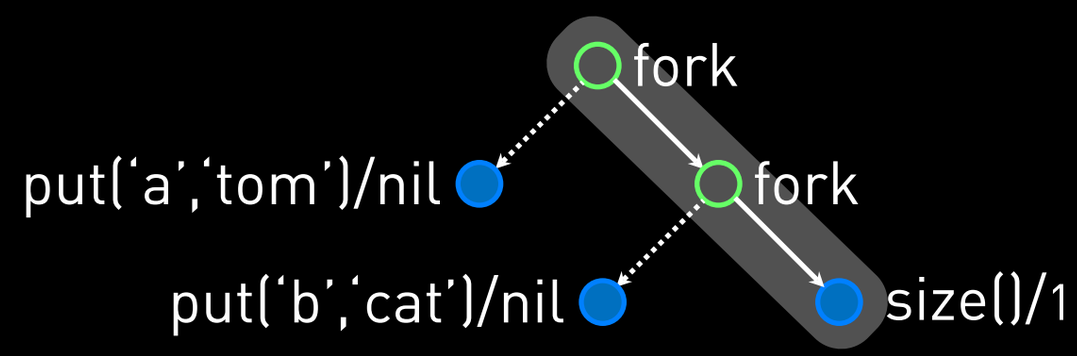

vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

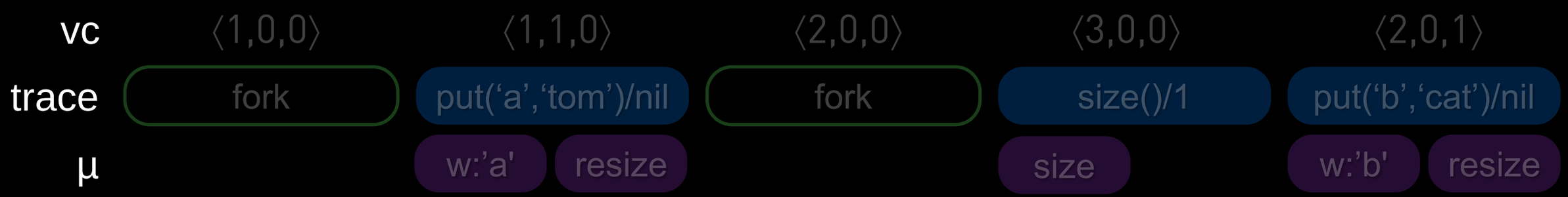
An example

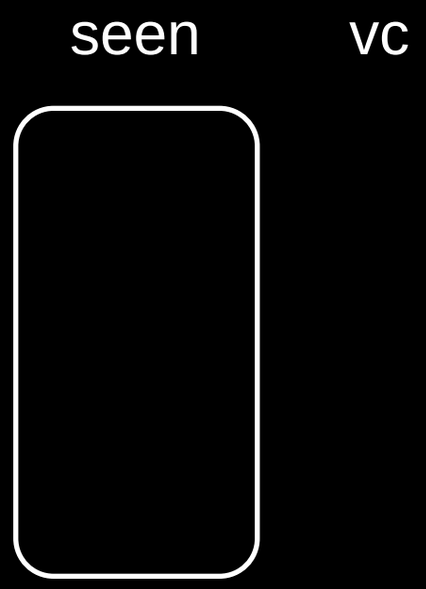

vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

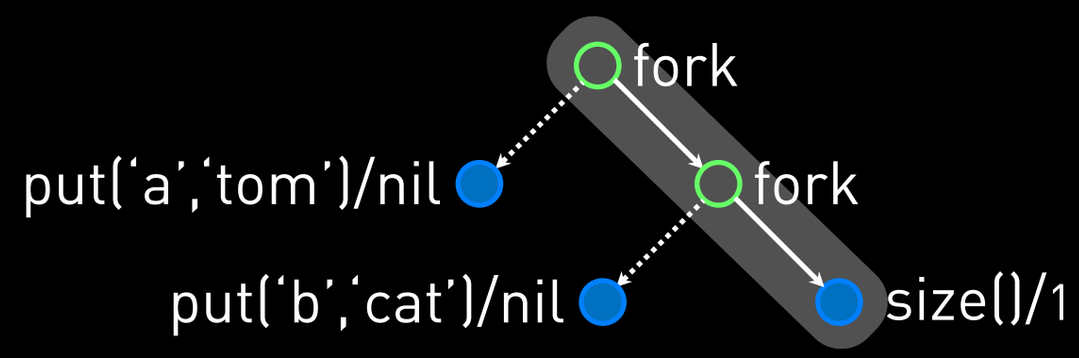

seen
vc


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

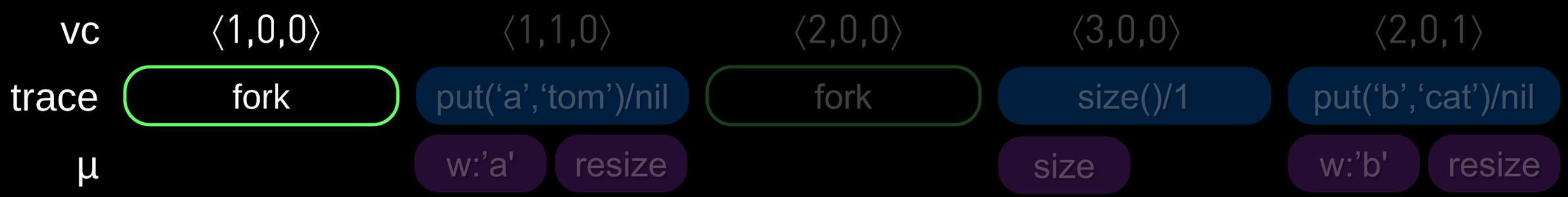
An example

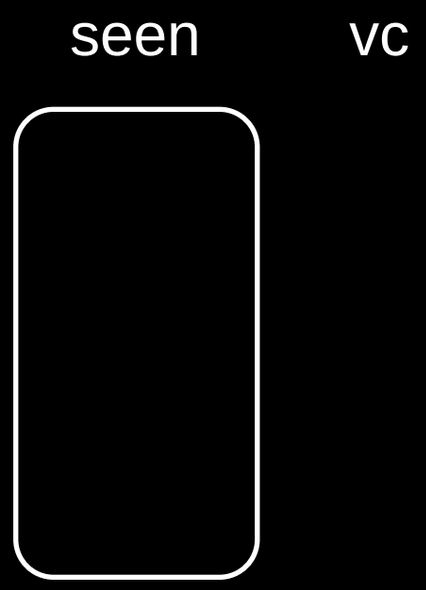

vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

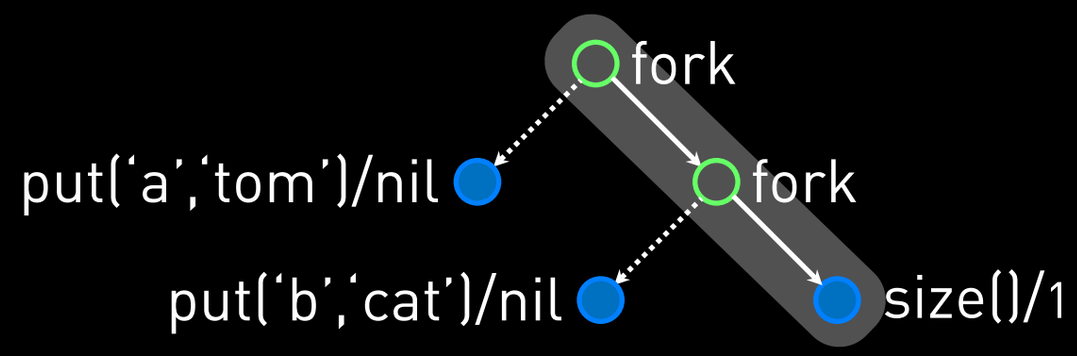

seen
vc


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

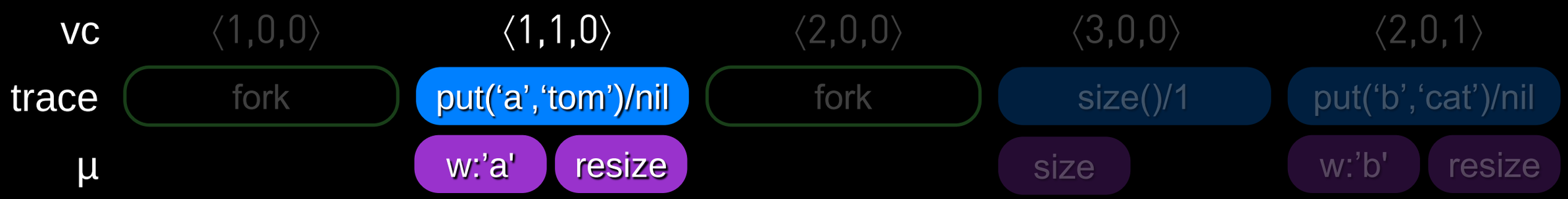
An example

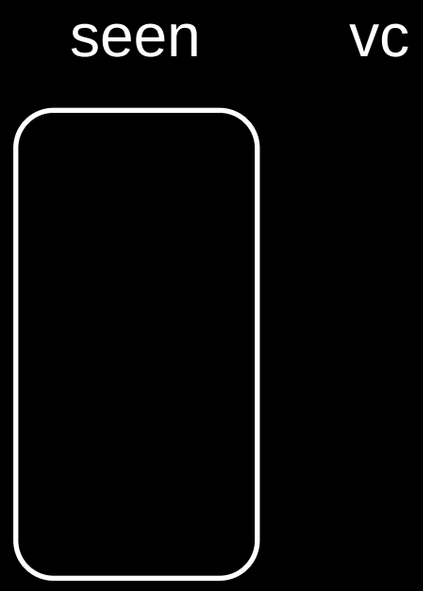

vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

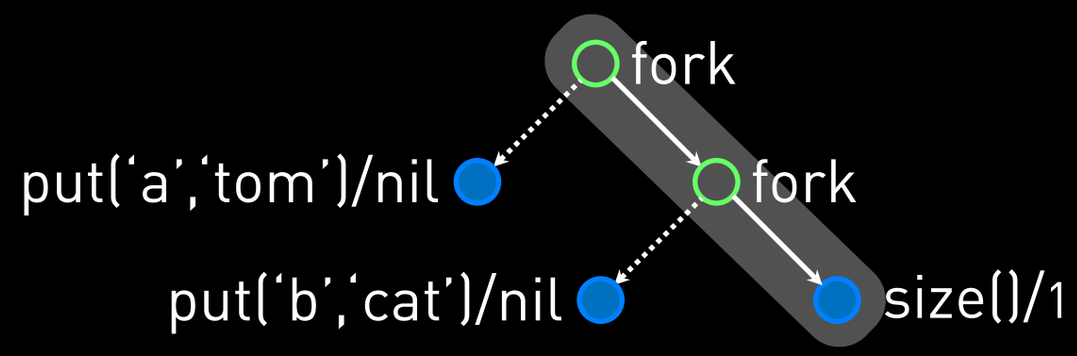

seen
vc


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

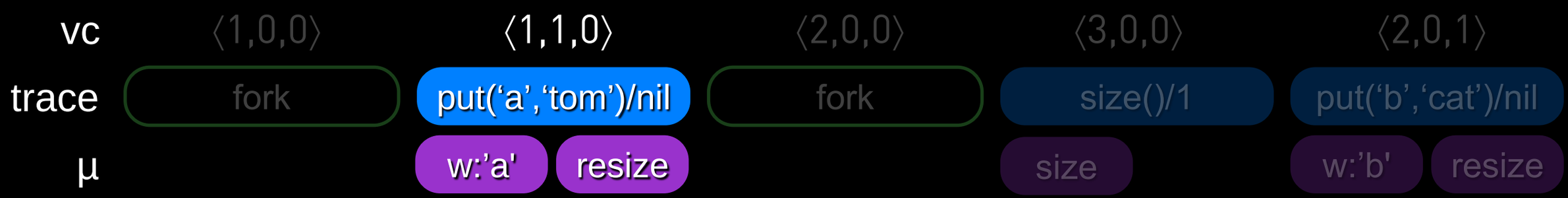
An example

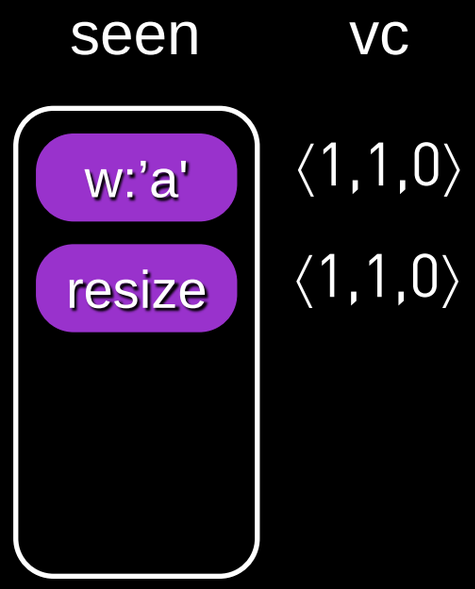

vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

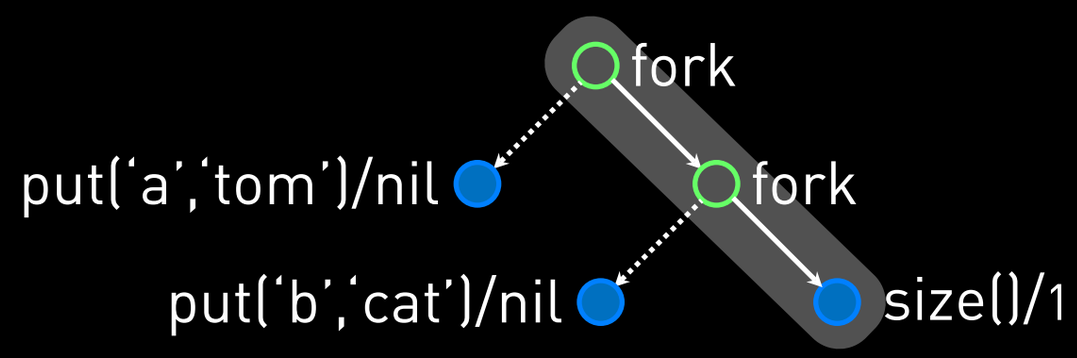

seen
vc


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

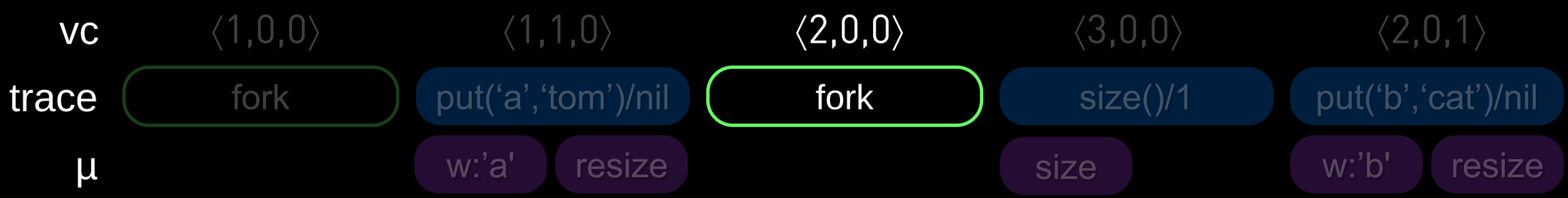
An example

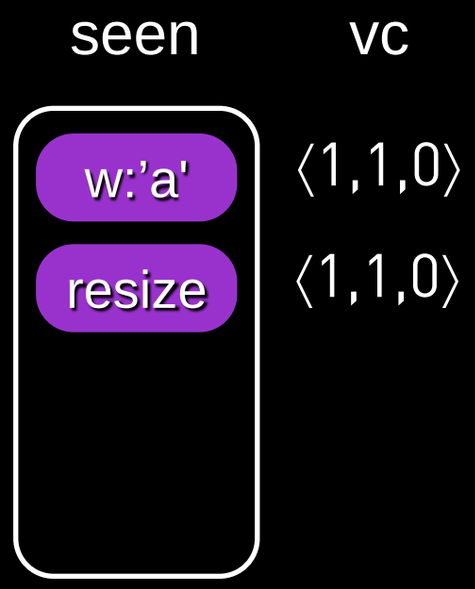

vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

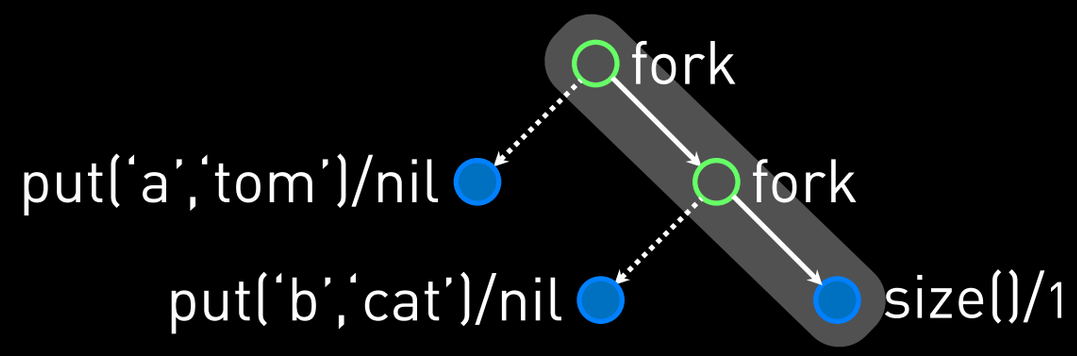

seen
vc
w:’a'
⟨1,1,0⟩
resize
⟨1,1,0⟩


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

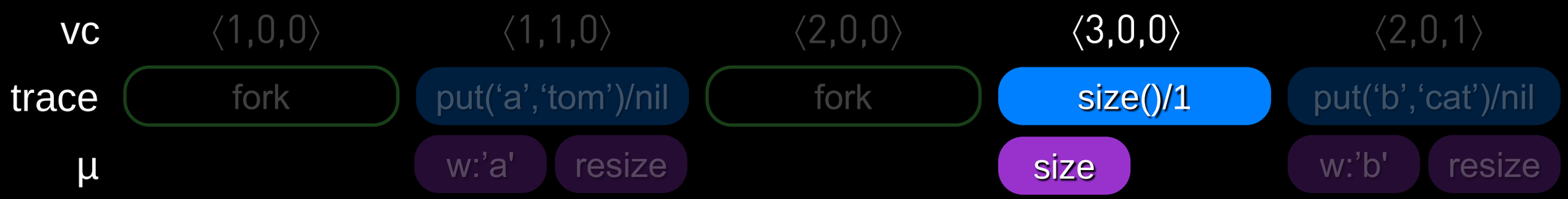
An example

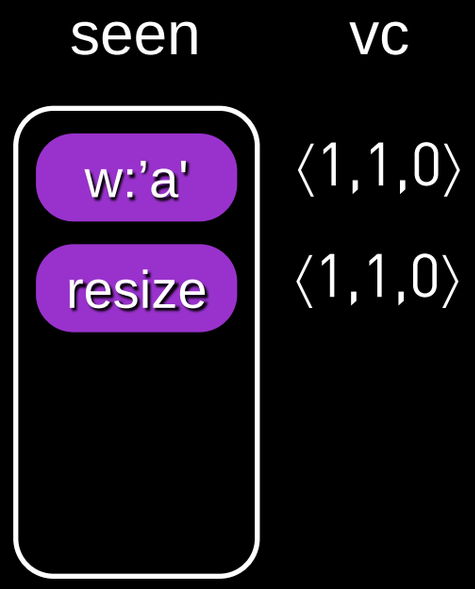

vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

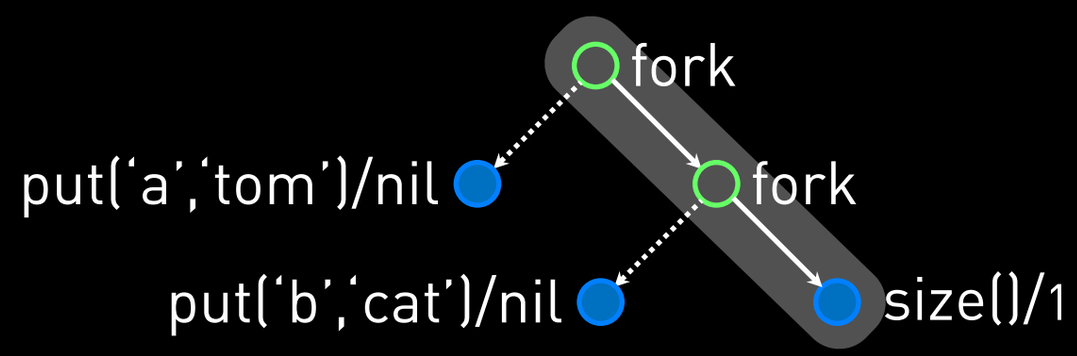

seen
vc
w:’a'
⟨1,1,0⟩
resize
⟨1,1,0⟩


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

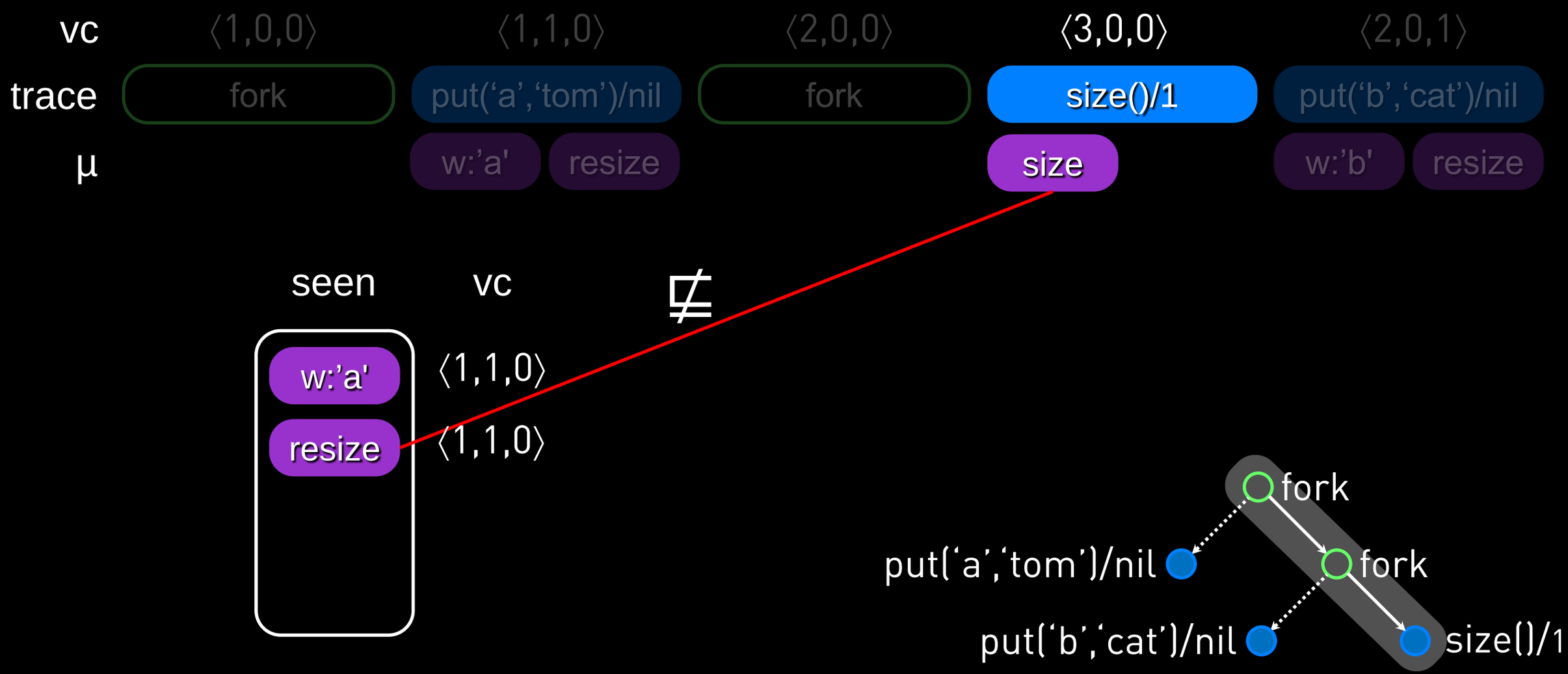
An example


vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize

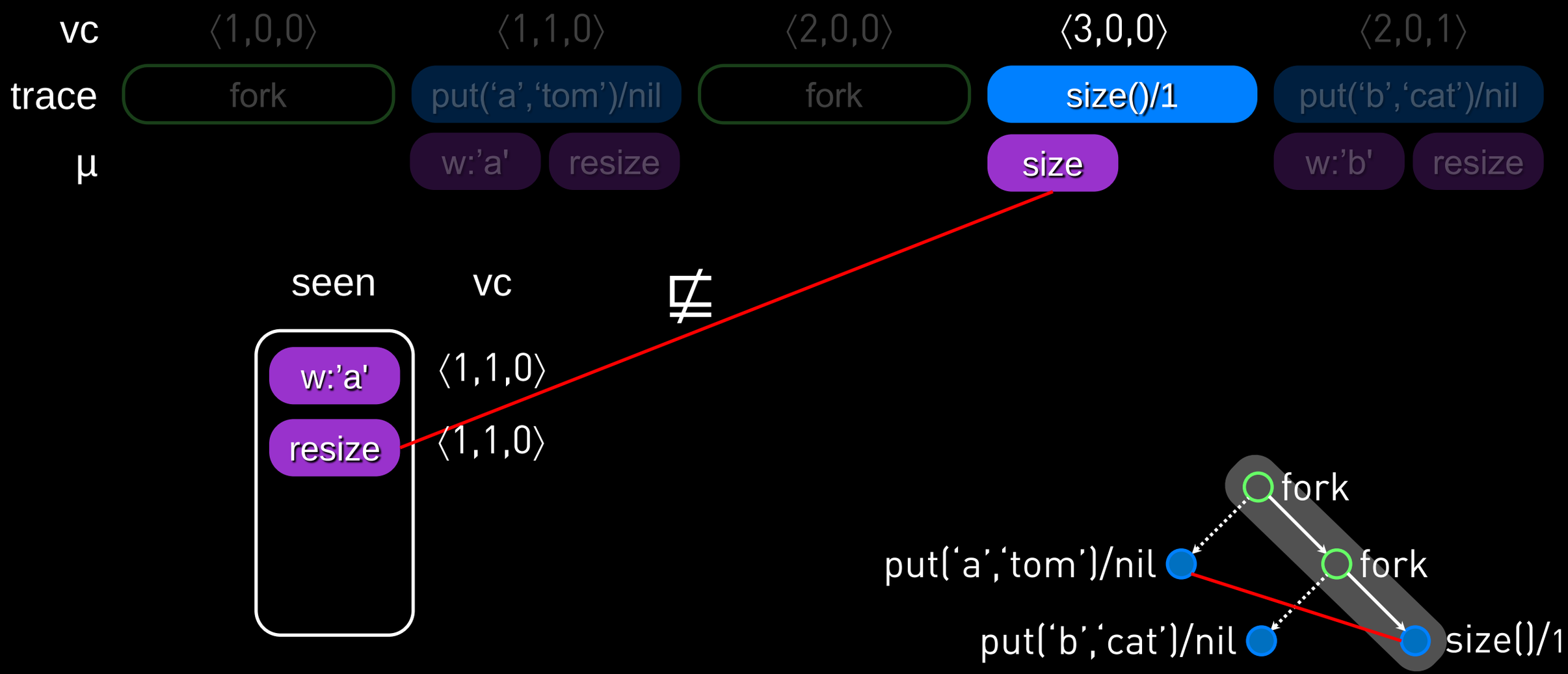

seen
vc
w:’a'
⟨1,1,0⟩
resize
⟨1,1,0⟩


fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

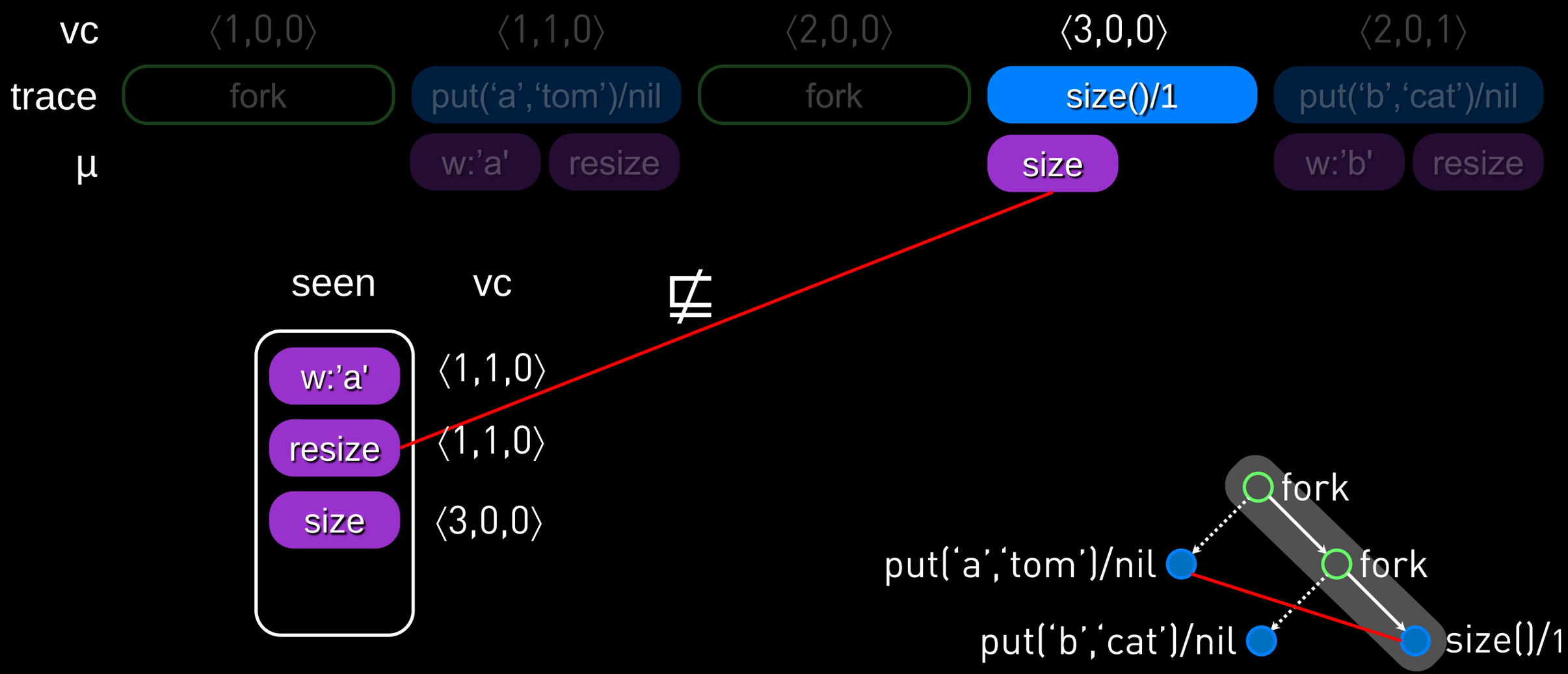
An example


vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize
seen
vc
⋤
w:’a'
resize
⟨1,1,0⟩
⟨1,1,0⟩
fork
put(‘a’,‘tom’)/nil
fork
put(‘b’,‘cat’)/nil
size()/1

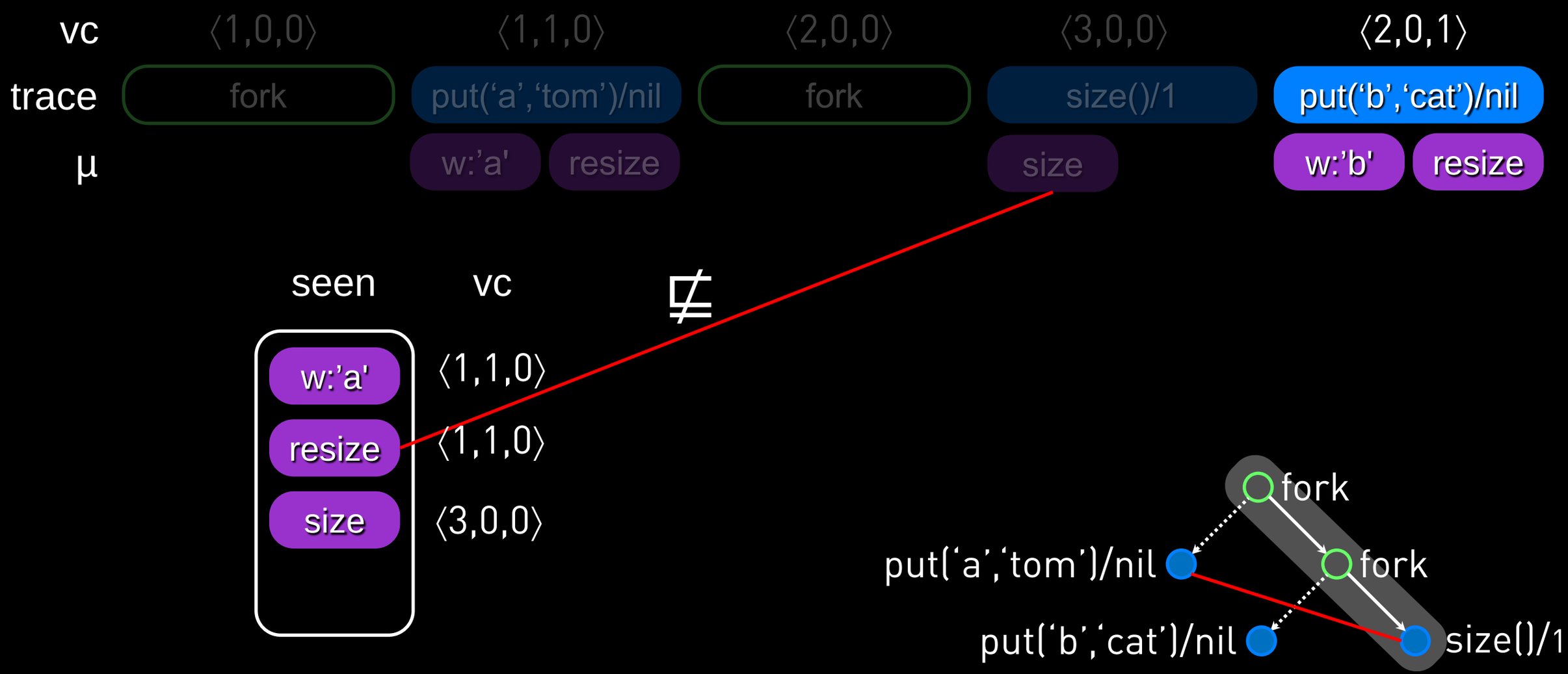
An example


vc
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
trace
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
μ
w:’a'
resize
size
w:’b'
resize
seen
vc
⋤
w:’a'
resize
⟨1,1,0⟩
⟨1,1,0⟩
fork
fork
put(‘a’,‘tom’)/nil
put(‘b’,‘cat’)/nil
size()/1

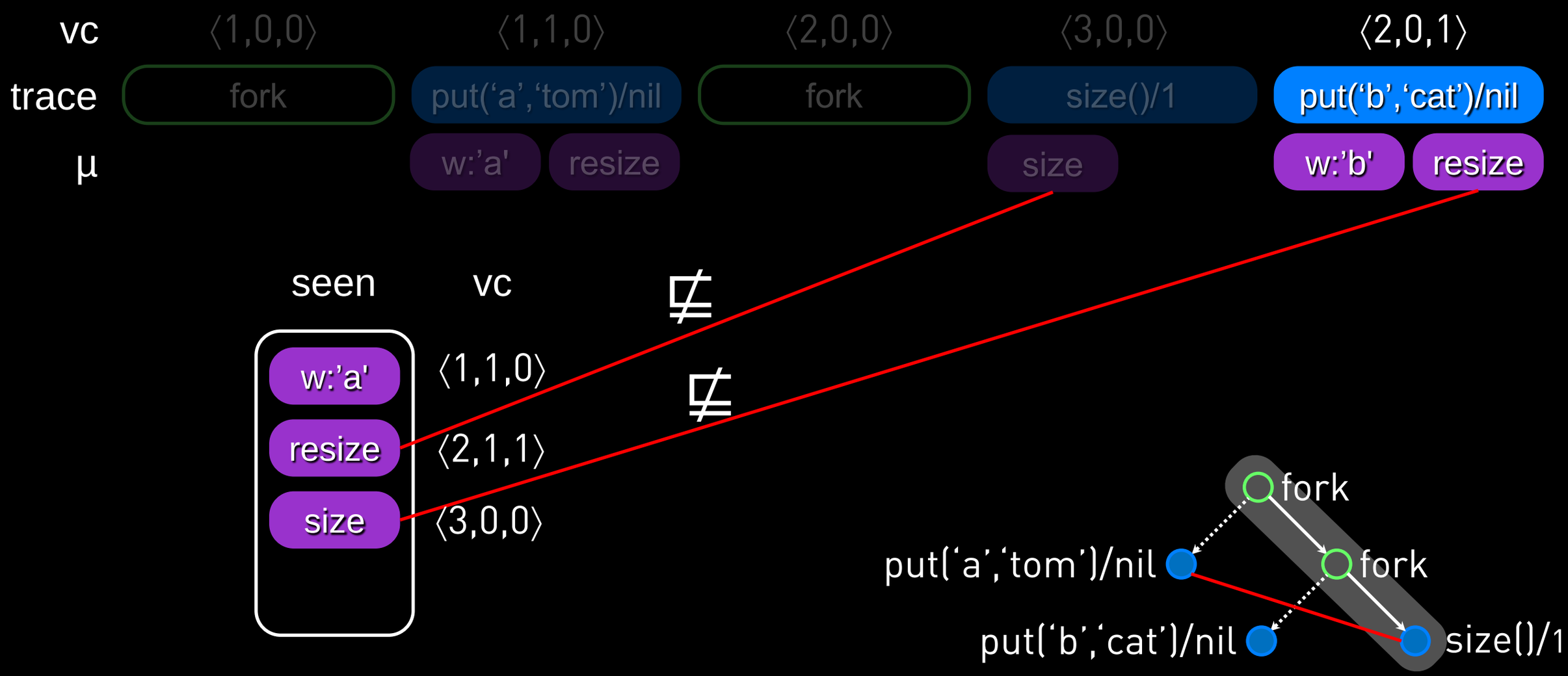
An example


vc
trace
μ
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
w:’a'
resize
size
w:’b'
resize
seen
vc
⋤
w:’a'
resize
size
⟨1,1,0⟩
⟨1,1,0⟩
⟨3,0,0⟩
fork
fork
put(‘a’,‘tom’)/nil
put(‘b’,‘cat’)/nil
size()/1

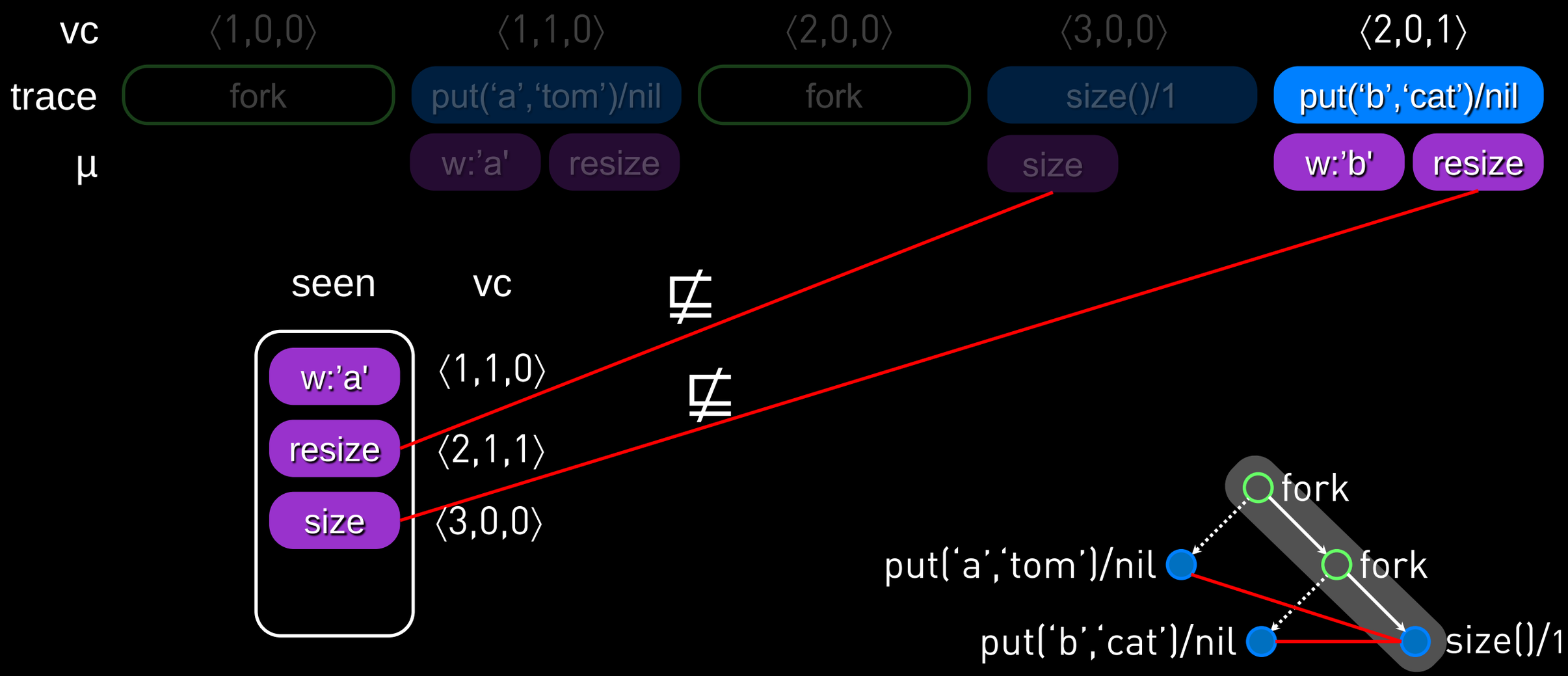
An example


vc
trace
μ
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
w:’a'
resize
size
w:’b'
resize
seen
vc
⋤
w:’a'
resize
size
⟨1,1,0⟩
⟨1,1,0⟩
⟨3,0,0⟩
fork
fork
put(‘a’,‘tom’)/nil
put(‘b’,‘cat’)/nil
size()/1

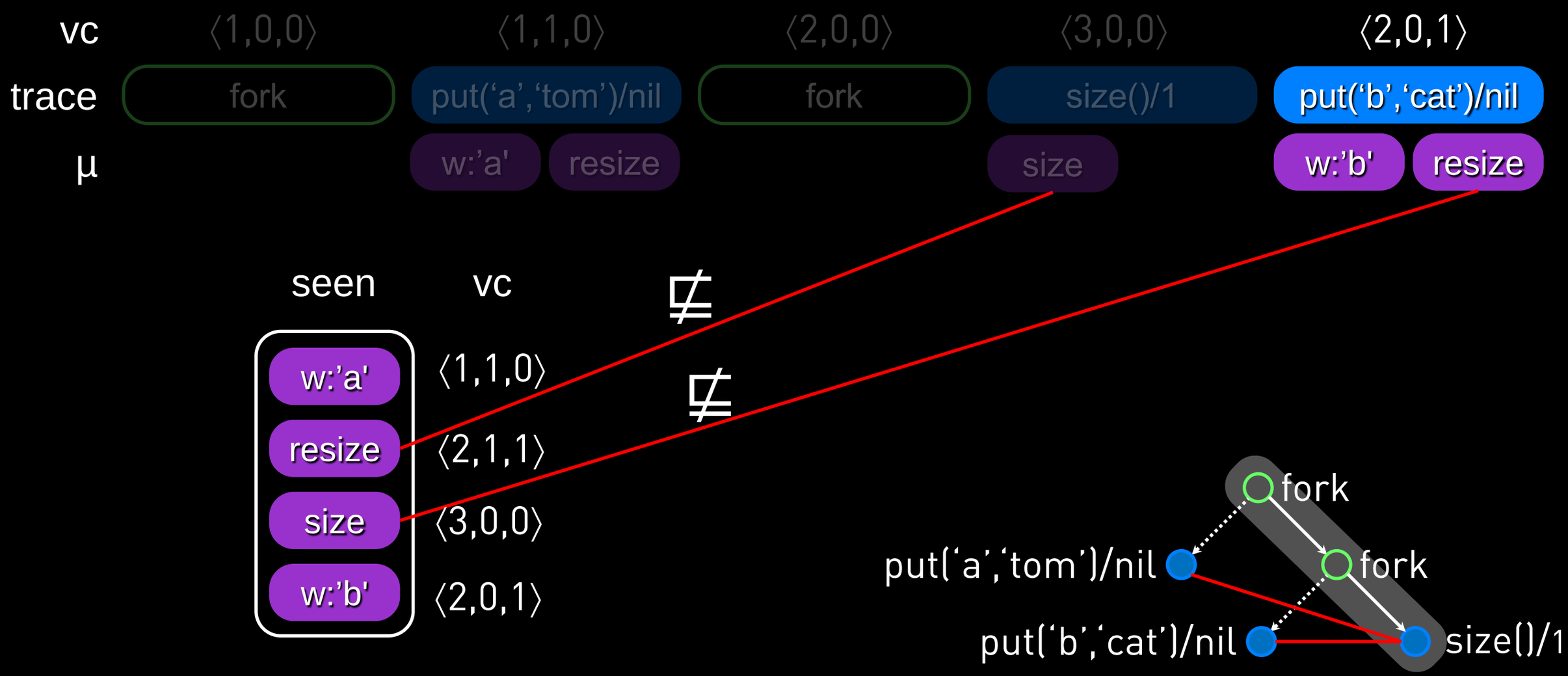
An example


vc
trace
μ
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
w:’a'
resize
size
w:’b'
resize
seen
vc
w:’a'
resize
size
⟨1,1,0⟩
⟨2,1,1⟩
⟨3,0,0⟩
⋢
⋢
fork
fork
put(‘a’,‘tom’)/nil
put(‘b’,‘cat’)/nil
size()/1

An example


vc
trace
μ
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
w:’a'
resize
size
w:’b'
resize
seen
vc
w:’a'
resize
size
⟨1,1,0⟩
⟨2,1,1⟩
⟨3,0,0⟩
⋢
⋢
fork
fork
put(‘a’,‘tom’)/nil
put(‘b’,‘cat’)/nil
size()/1

An example


vc
trace
μ
⟨1,0,0⟩
⟨1,1,0⟩
⟨2,0,0⟩
⟨3,0,0⟩
⟨2,0,1⟩
fork
put(‘a’,‘tom’)/nil
fork
size()/1
put(‘b’,‘cat’)/nil
w:’a'
resize
size
w:’b'
resize
seen
vc
w:’a'
resize
size
w:’b'
⟨1,1,0⟩
⟨2,1,1⟩
⟨3,0,0⟩
⟨2,0,1⟩
⊈
⊈
fork
fork
put(‘a’,‘tom’)/nil
put(‘b’,‘cat’)/nil
size()/1

Asymptotic complexity

O(n) conflict checks per encountered operation

Can we do better?

Extended Logical Fragment (ECL)

A logical fragment which ensures O(1) checks per operation

Extended Logical Fragment (ECL)

Variables of first method

Variables of second method

$$S ::= V_1 \neq V_2 \mid S \wedge S \mid \texttt{true} \mid \texttt{false}$$

Extended Logical Fragment (ECL)

Variables of first method

Variables of second method

$$S ::= V_1 \neq V_2 \mid S \wedge S \mid \texttt{true} \mid \texttt{false}$$

Example:

$$put(k_1, v_1)/r_1 \bowtie get(k_2)/v_2 \iff k1 \neq k2 \vee v_1 = r_1$$

Extended Logical Fragment (ECL)

Variables of first method

Variables of second method

$S ::= V_1 \neq V_2 \mid S \wedge S \mid \texttt{true} \mid \texttt{false}$

Example:

$\texttt{put}(k_1, v_1)/r_1 \bowtie \texttt{get}(k_2)/v_2 \iff \texttt{k1} \neq \texttt{k2} \vee v_1 = r_1$

Limited relationship between variables of different methods.
For instance, $V_1 = V_2$ not allowed

Extended Logical Fragment (ECL)

$S ::= V_1 \neq V_2 \mid S \wedge S \mid \text{true} \mid \text{false}$

$B ::= P_{V1} \mid P_{V2} \mid \neg B \mid B \wedge B \mid B \vee B \mid \text{true} \mid \text{false}$

Any predicate over V_1

Any predicate over V_2

Extended Logical Fragment (ECL)

$S ::= V_1 \neq V_2 \mid S \wedge S \mid true \mid false$

$B ::= P_{V1} \mid P_{V2} \mid \neg B \mid B \wedge B \mid B \vee B \mid true \mid false$

Any predicate over v_1

Any predicate over v_2

Example:

$put(k_1, v_1)/r_1 \bowtie get(k_2)/v_2 \iff k1 \neq k2 \vee v_1 = r_1$

Extended Logical Fragment (ECL)

$S ::= V_1 \neq V_2 \mid S \wedge S \mid \text{true} \mid \text{false}$

$B ::= P_{V1} \mid P_{V2} \mid \neg B \mid B \wedge B \mid B \vee B \mid \text{true} \mid \text{false}$

$X ::= S \mid B \mid X \wedge X \mid X \vee B$

Not $X \vee X$

Extended Logical Fragment (ECL)

$S ::= V_1 \neq V_2 \mid S \wedge S \mid true \mid false$

$B ::= P_{V1} \mid P_{V2} \mid \neg B \mid B \wedge B \mid B \vee B \mid true \mid false$

$X ::= S \mid B \mid X \wedge X \mid X \vee B$

Not $X \vee X$

Example:

$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \iff k1 \neq k2 \vee v_1 = r_1$

Comes from S

Comes from B

Extended Logical Fragment (ECL)

$S ::= V_1 \neq V_2 \mid S \wedge S \mid \text{true} \mid \text{false}$

$B ::= P_{V1} \mid P_{V2} \mid \neg B \mid B \wedge B \mid B \vee B \mid \text{true} \mid \text{false}$

$X ::= S \mid B \mid X \wedge X \mid X \vee B$

ECL ensures O(1) time per operation

Extends SIMPLE in Kulkarni et al. PLDI ’11

Classic R/W Race Detection...

...is now a special case:

$\text{read}()/r_1 \bowtie \text{read}()/r_2 \quad \Leftrightarrow \quad \text{true}$

$\text{read}()/r_1 \bowtie \text{write}(v_2) \quad \Leftrightarrow \quad \text{false}$

$\text{write}(v_1) \bowtie \text{write}(v_2) \quad \Leftrightarrow \quad \text{false}$

Fits inside ECL

What about this?

…a more precise spec:

$\text{read}()/r_1 \bowtie \text{read}()/r_2 \quad \Leftrightarrow \quad \text{true}$

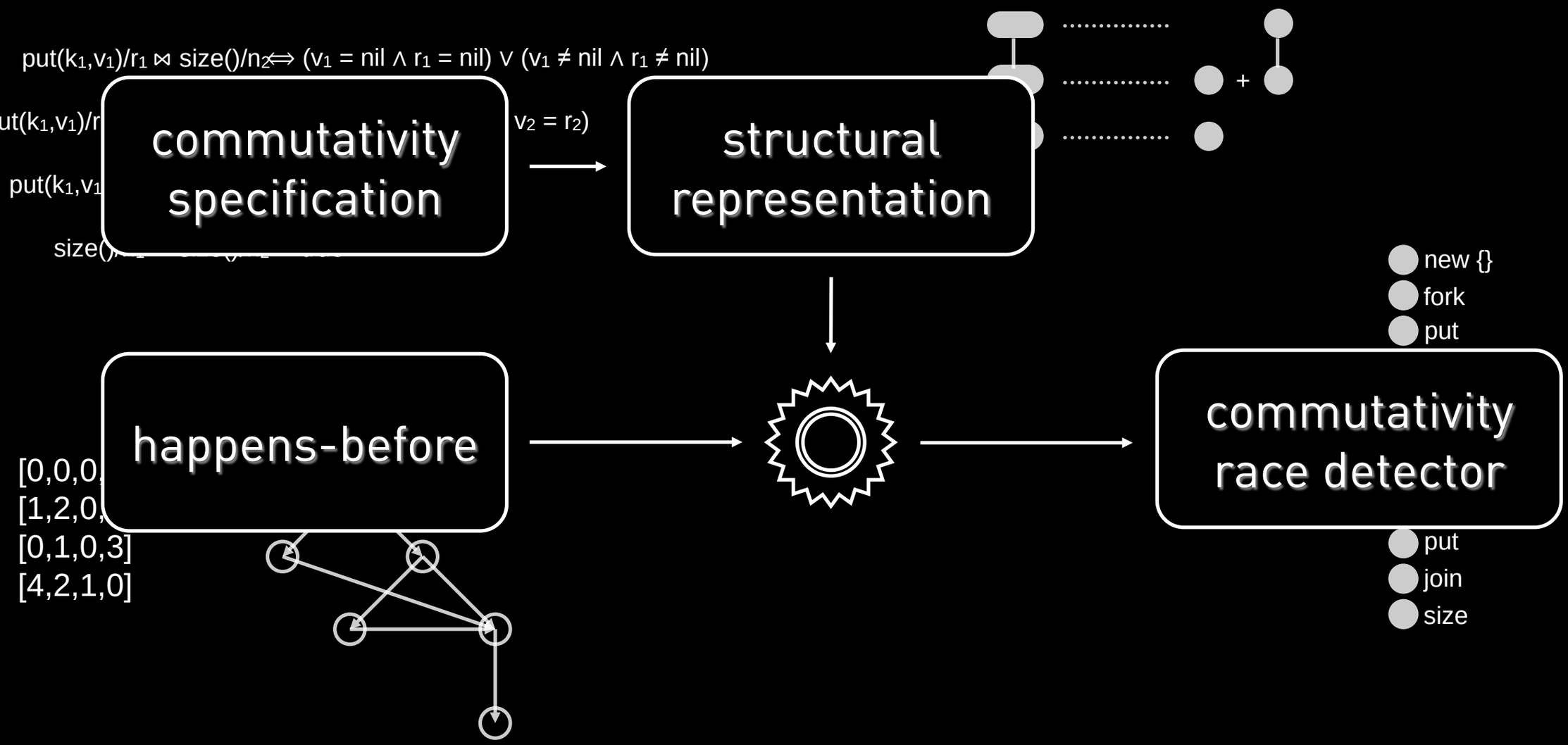
$\text{read}()/r_1 \bowtie \text{write}(v_2) \quad \Leftrightarrow \quad r_1 = v_2$

$\text{write}(v_1) \bowtie \text{write}(v_2) \quad \Leftrightarrow \quad v_1 = v_2$

ECL does not allow = between variables of different operations

NO!

Commutativity race detection



Experimental Results

Implemented a commutativity detector for Java

Found both correctness and performance bugs in large-scale apps (e.g., H2 Database)

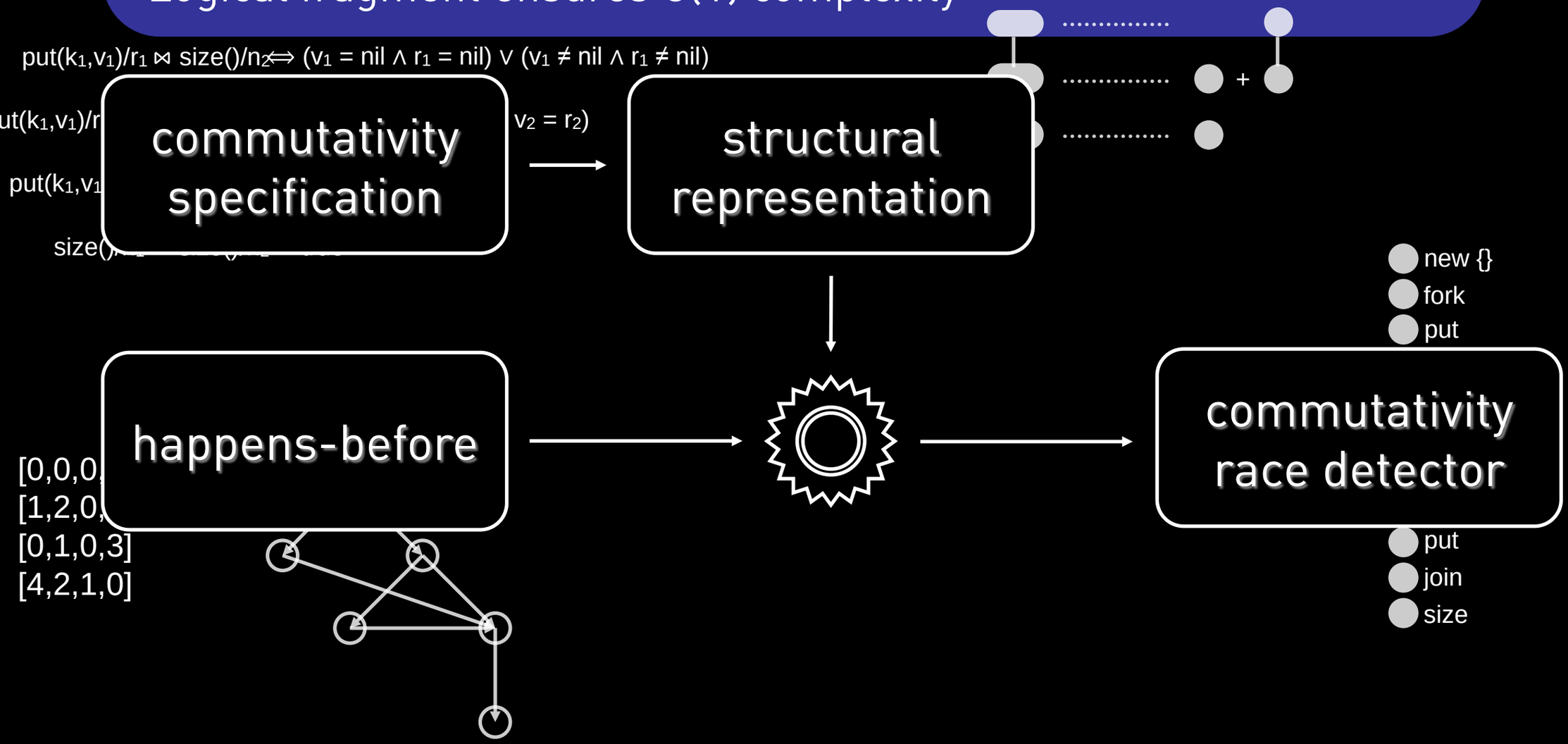
Commutativity race detection

Parametric Concurrency Analysis Framework

Generalizes classic read-write race detection

Enables concurrency analysis for clients of high-level abstractions

Logical fragment ensures O(1) complexity



Applications

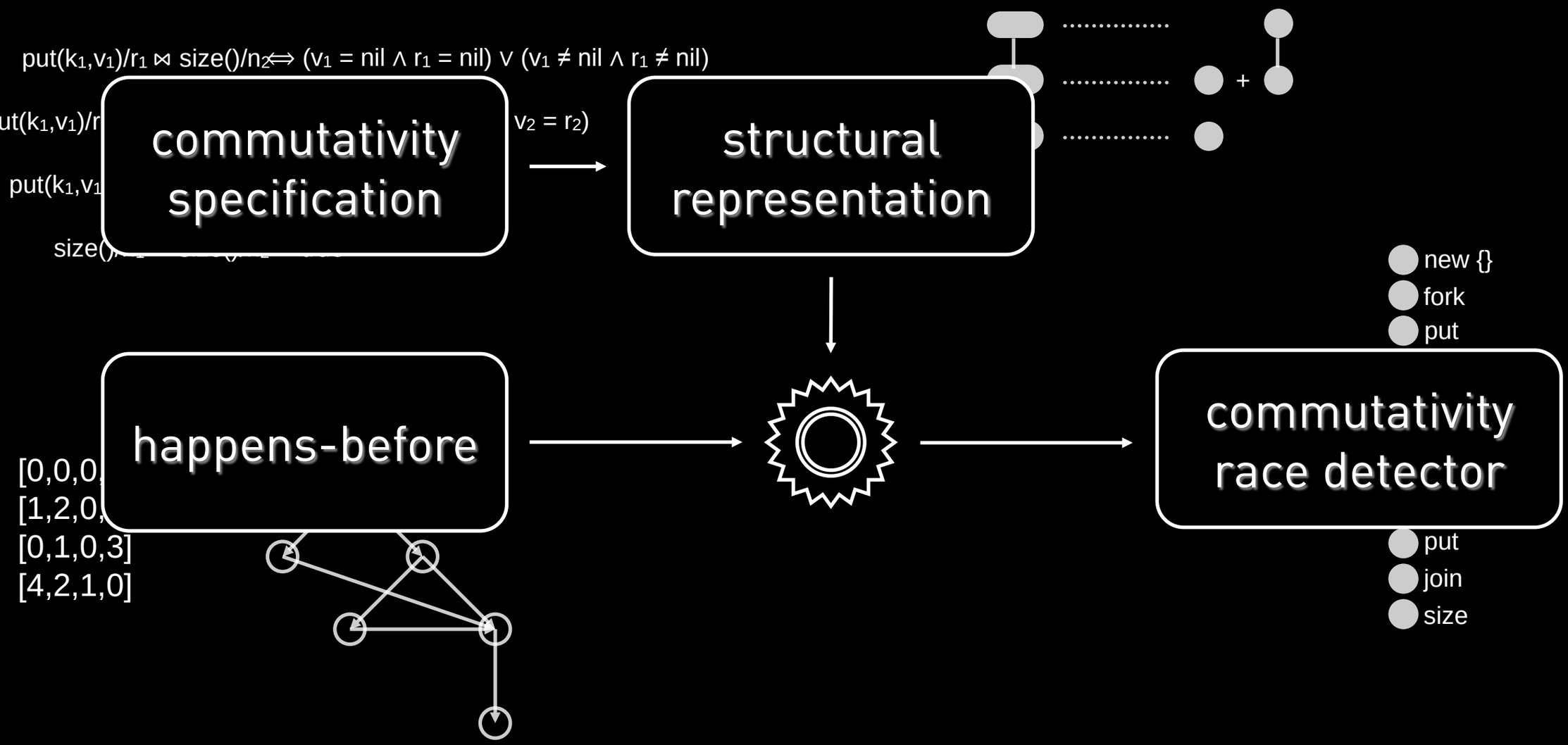
Software Defined Networks

Stateless Model Checking

Eventual Consistency

Optimistic Concurrency Control

Open Problems

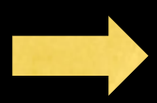


Question 1:
Can we learn the commutativity spec?

Specification of a Map ADT

?

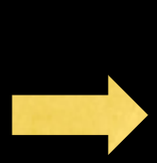
Automatic Inference



$$
\begin{aligned}
put(k_1,v_1)/r_1 \bowtie size()/n_2 &\iff (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)\\
put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 &\iff k_1 \neq k_2 \vee (v_1 = r_1 \wedge v_2 = r_2)\\
put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 &\iff k_1 \neq k_2 \vee v_1 = r_1\\
size()/n_1 \bowtie size()/n_2 &\iff true\\
size()/n_1 \bowtie get(k_2)/v_2 &\iff true\\
get(k_1)/v_1 \bowtie get(k_2)/v_2 &\iff true
\end{aligned}
$$

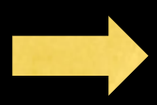
Question 1:
Can we learn the commutativity spec?

Specification of a Map ADT



?

Automatic Inference



$$
\begin{aligned}
put(k_1,v_1)/r_1 \bowtie size()/n_2 &\iff (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)\\
put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 &\iff k_1 \neq k_2 \vee (v_1 = r_1 \wedge v_2 = r_2)\\
put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 &\iff k_1 \neq k_2 \vee v_1 = r_1\\
size()/n_1 \bowtie size()/n_2 &\iff true\\
size()/n_1 \bowtie get(k_2)/v_2 &\iff true\\
get(k_1)/v_1 \bowtie get(k_2)/v_2 &\iff true
\end{aligned}
$$

Learning Commutativity Specifications, CAV'15
Black Box Learning of specifications from examples

Many open problems: state, quantifiers, minimization

Timon Gehr

Dimitar Dimitrov

Open Problems

can we learn it?

put(k_1,v_1)/r_1 ⋈ size()/n_2 ⇔ (v_1 = nil ∧ r_1 = nil) ∨ (v_1 ≠ nil ∧ r_1 ≠ nil)

put(k_1,v_1)/r

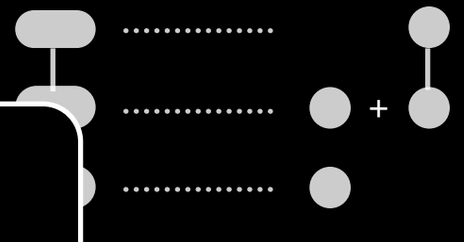
put(k_1,v_1

size()/

commutativity specification

v_2 = r_2)

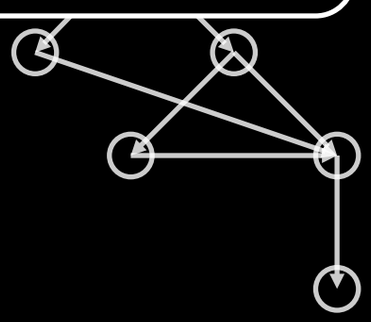
structural representation



new {}
fork
put

happens-before

[0,0,0,
[1,2,0,
[0,1,0,3]
[4,2,1,0]



commutativity race detector

put
join
size

Question 2:
Logic Expressivity vs. Asymptotic Complexity

Recall that ECL allows for O(1) checks per operation:

$$S ::= V_1 \neq V_2 \mid S \wedge S \mid \texttt{true} \mid \texttt{false}$$

$$B ::= P_{V1} \mid P_{V2} \mid \neg B \mid B \wedge B \mid B \vee B \mid \texttt{true} \mid \texttt{false}$$

$$X ::= S \mid B \mid X \wedge X \mid X \vee B$$

What is the richest logical fragment which guarantees O(1)?

What about O(logN)?

Open Problems

can we learn it?

richest fragment to obtain O(1) time?

put(k_1,v_1)/r_1 ⋈ size()/n_2 ⇔ (v_1 = nil ∧ r_1 = nil) ∨ (v_1 ≠ nil ∧ r_1 ≠ nil)

put(k_1,v_1)/r

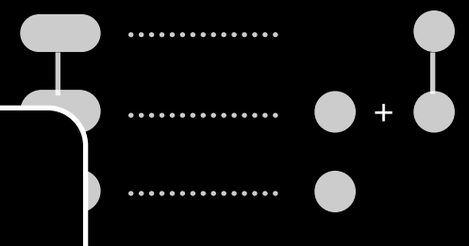
put(k_1,v_1

size()/

v_2 = r_2)

commutativity specification

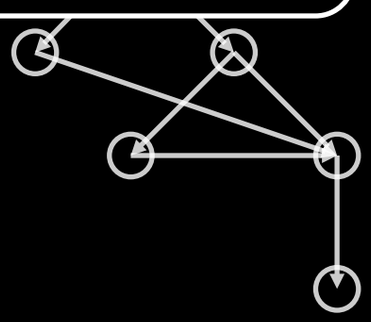
structural representation



new {}
fork
put

happens-before

[0,0,0,
[1,2,0,
[0,1,0,3]
[4,2,1,0]



commutativity race detector

put
join
size

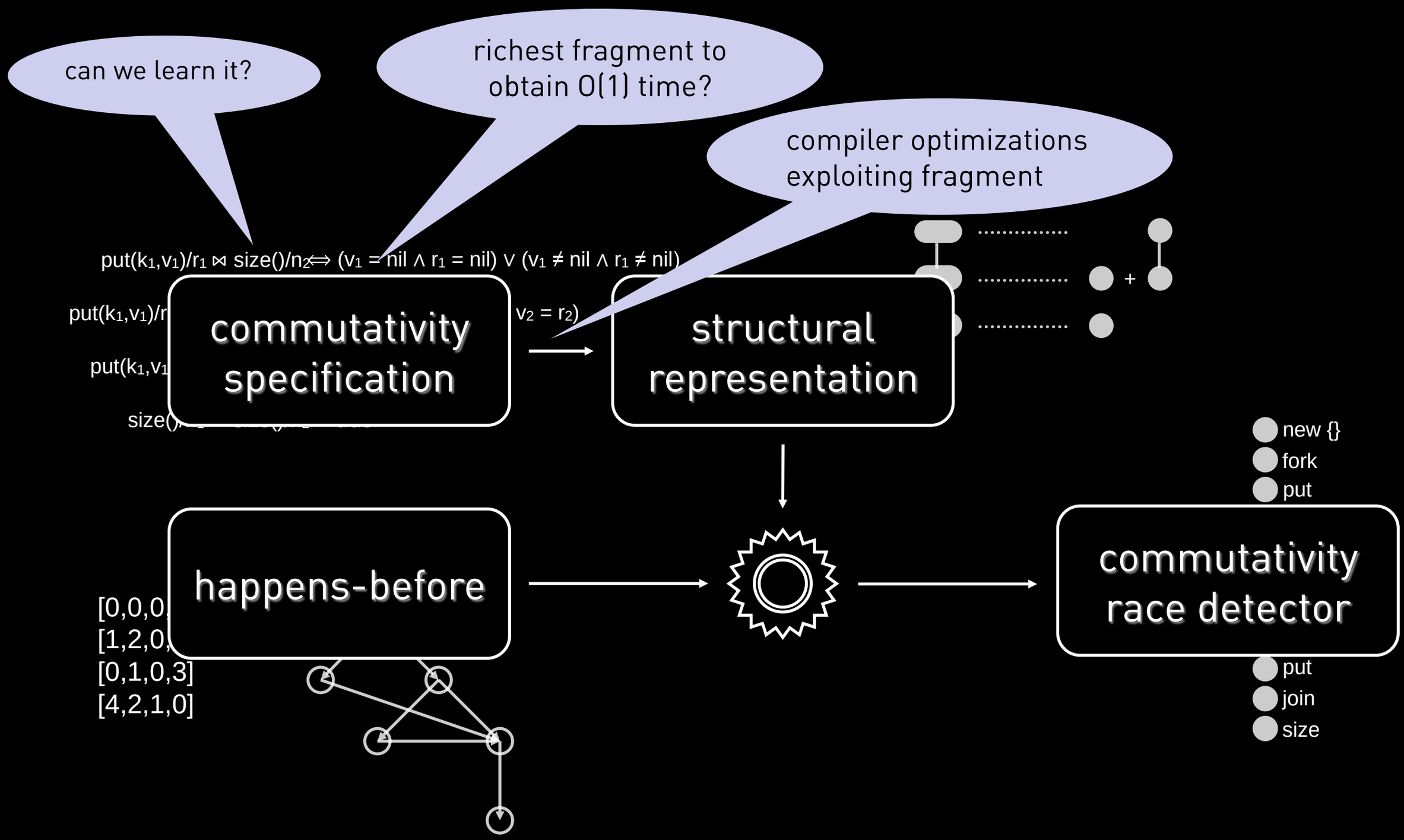
Question 3:
Commutativity Compiler Optimizations

Obtaining a succinct micro-operation representation can be tricky

New compiler optimizations which depend on the logical fragment

and on the property we are checking

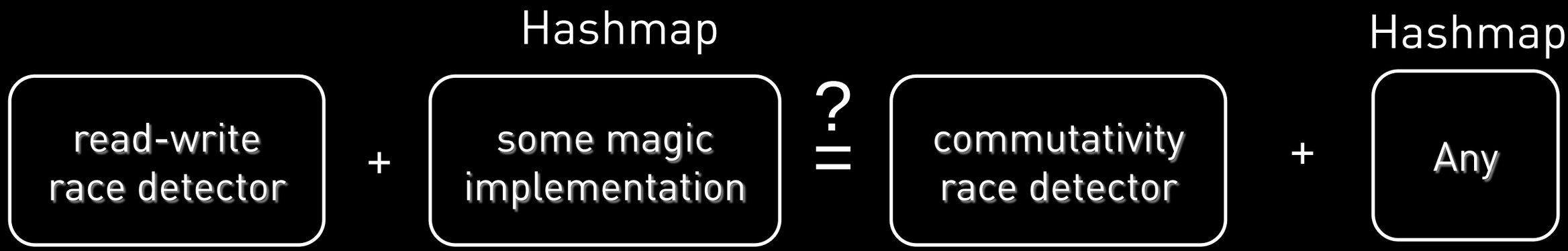
Open Problems



Question 4 :
Impossibility questions

Can a read-write race detector precisely detect commutativity races?

At what cost?



What is the “consensus-like” hierarchy of concurrency analyzers?

Open Problems

can we learn it?

richest fragment to obtain O(1) time?

compiler optimizations exploiting fragment

impossibility questions

put(k_1,v_1)/r_1 ⋈ size()/n_2 ⟺ (v_1 = nil ∧ r_1 = nil) ∨ (v_1 ≠ nil ∧ r_1 ≠ nil)

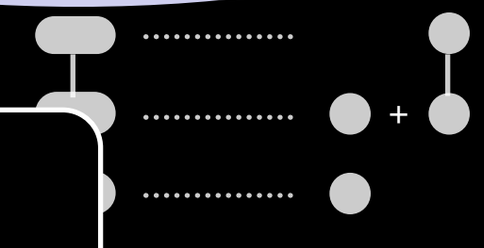
put(k_1,v_1)/r … v_2 = r_2)

put(k_1,v_1…

size()/…

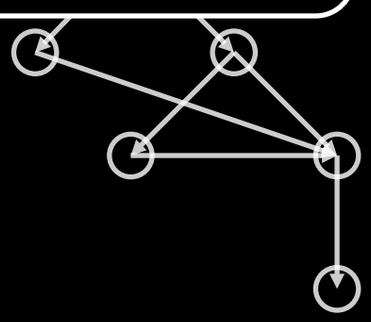
commutativity specification

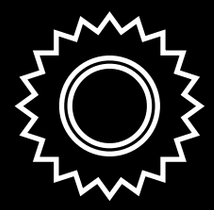
structural representation



happens-before

[0,0,0,…
[1,2,0,…
[0,1,0,3]
[4,2,1,0]





commutativity race detector

Open Problems

can we learn it?

richest fragment to obtain O(1) time?

compiler optimizations exploiting fragment

impossibility questions

put(k_1,v_1)/r_1 ⋈ size()/n_2 ⇔ (v_1 = nil ∧ r_1 = nil) ∨ (v_1 ≠ nil ∧ r_1 ≠ nil)

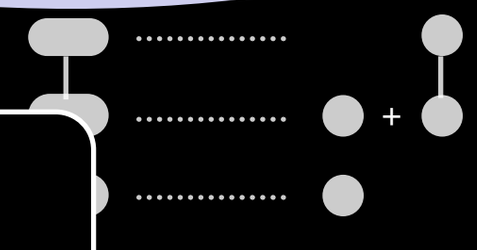
put(k_1,v_1)/r … v_2 = r_2)

put(k_1,v_1…

size()/…

commutativity specification

structural representation



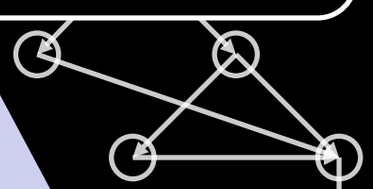
new {}
fork
put

happens-before

commutativity race detector

put
join
size

[0,0,0,…
[1,2,0,…
[0,1,0,3]
[4,2,1,0]



different instantiations (e.g. VCs, SP-graphs)

other analyzers (e.g. atomicity)

Commutativity race detection

Parametric Concurrency Analysis Framework

Generalizes classic read-write race detection

Enables concurrency analysis for clients of high-level abstractions

Logical fragment ensures O(1) complexity

Many open problems