Commutativity Race Detection

Concepts, Algorithms and Open Problems

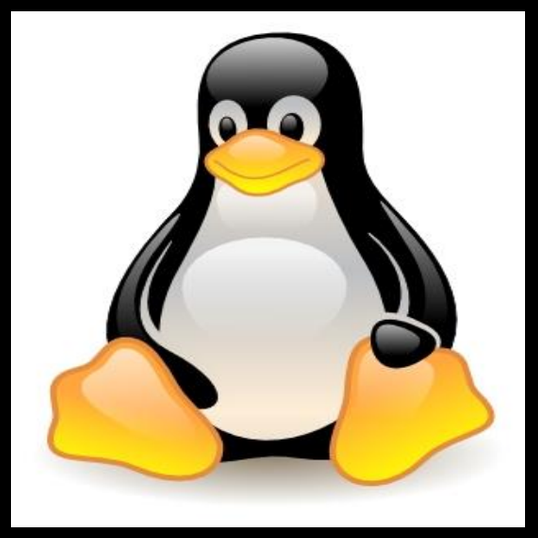
Martin Vechev

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 bad (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

```
Object o = new Object();

fork {
   o.talk = “Martin”
}

o.talk = “Marco”
```

1. two writes to o.talk
2. not ordered by program

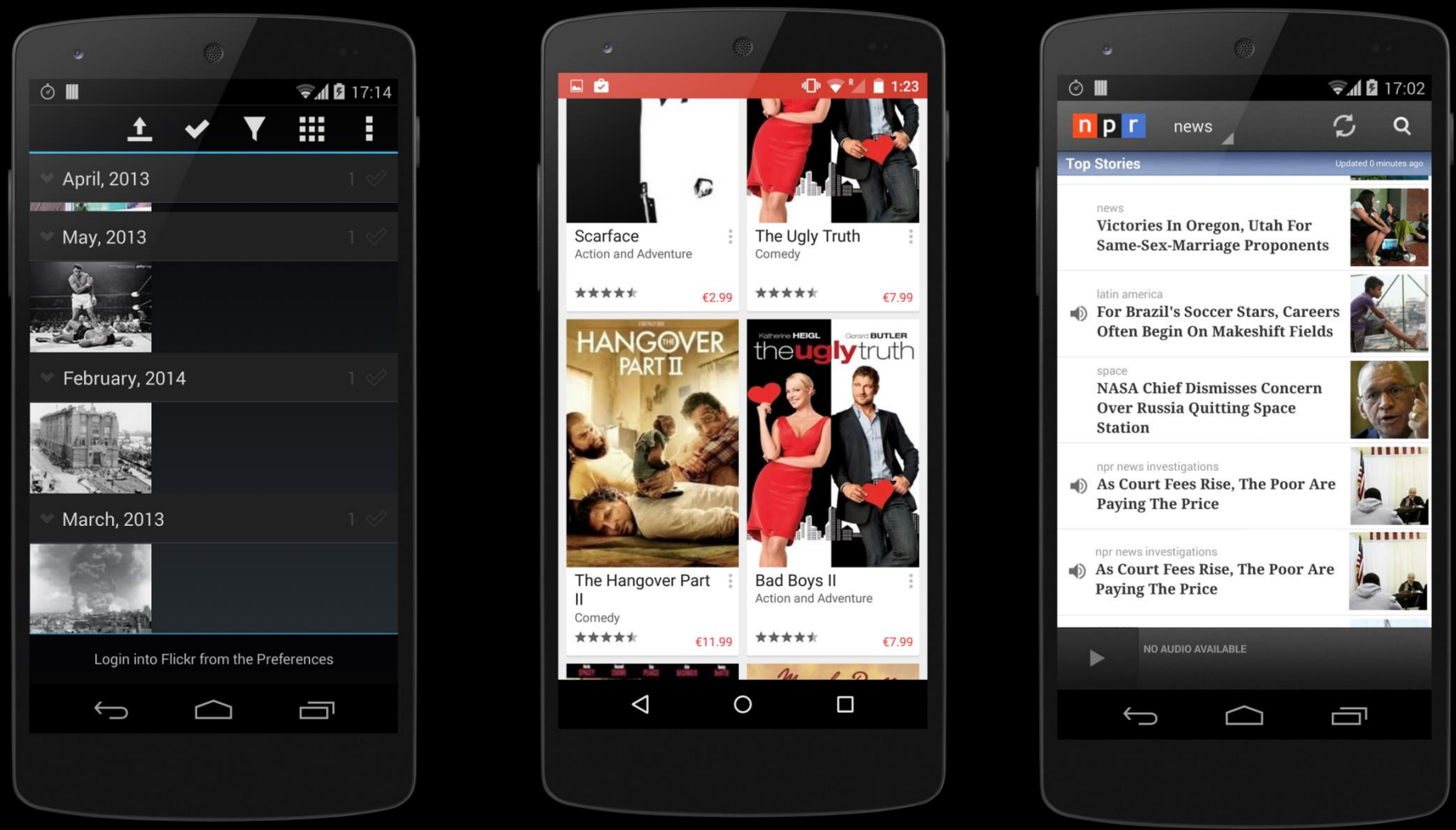
Android Data Races

Scalable Race Detection for Android Applications, ACM OOPSLA 2015

Pavol Bielik

Veselin Raychev

Races cause memory leaks, crashes, null dereferences, bad UI



http://www.eventracer.org/android

Dynamic Race Detection

20+ years of algorithms and optimizations

Lots of work on the subect:

improve asymptotic complexity, parallelization, sampling, hardware, optimizations, ...

<u>Guarantees:</u> proves program free of races for a given input state

Dynamic Race Detection

20+ years of algorithms and optimizations

Lots of work on the subect:

improve asymptotic complexity, parallelization, sampling, hardware, optimizations, ...

<u>Guarantees:</u> proves program free of races for a given input state

Recommended reading:

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

} Practice

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. V., V. Sarkar

} Theory

Trend:
Interference shifts to interfaces

Increasingly, low-level functionality captured inside high level objects

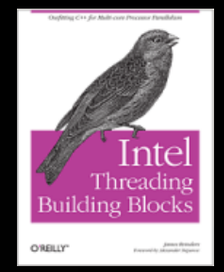
util.concurrent

The gap

Classic data race detection

Read-write notion of conflict

Many optimizations



Interference at the interface

Current detectors ineffective in handling real-world programs

Wanted

New kind of race detection

Higher level notion of conflict

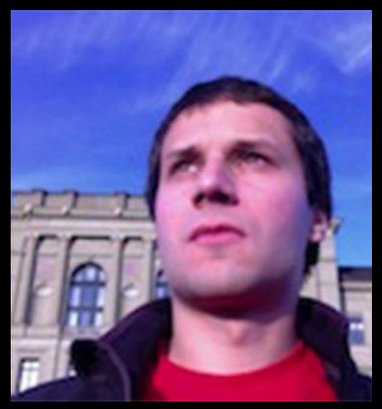
New optimizations

Interference at the interface

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

Commutativity Race Detection, ACM PLDI'14

Dimitar Dimitrov, Veselin Raychev, Eric Koskinen, M.V.



Dimitar Dimitrov

Veselin Raychev

Commutativity Race

two high-level operations do not commute

and are not ordered by the program

capture

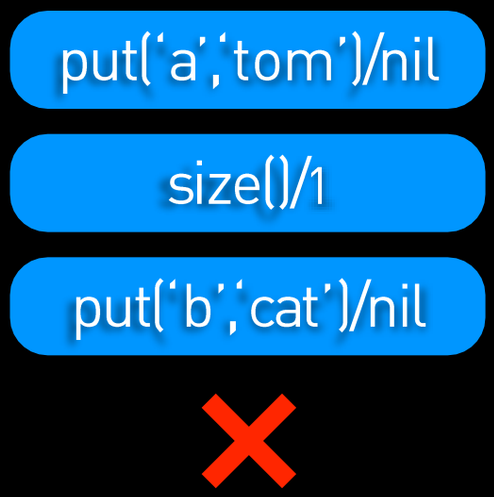
interference between
high-level operations

Commutativity Race

```
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

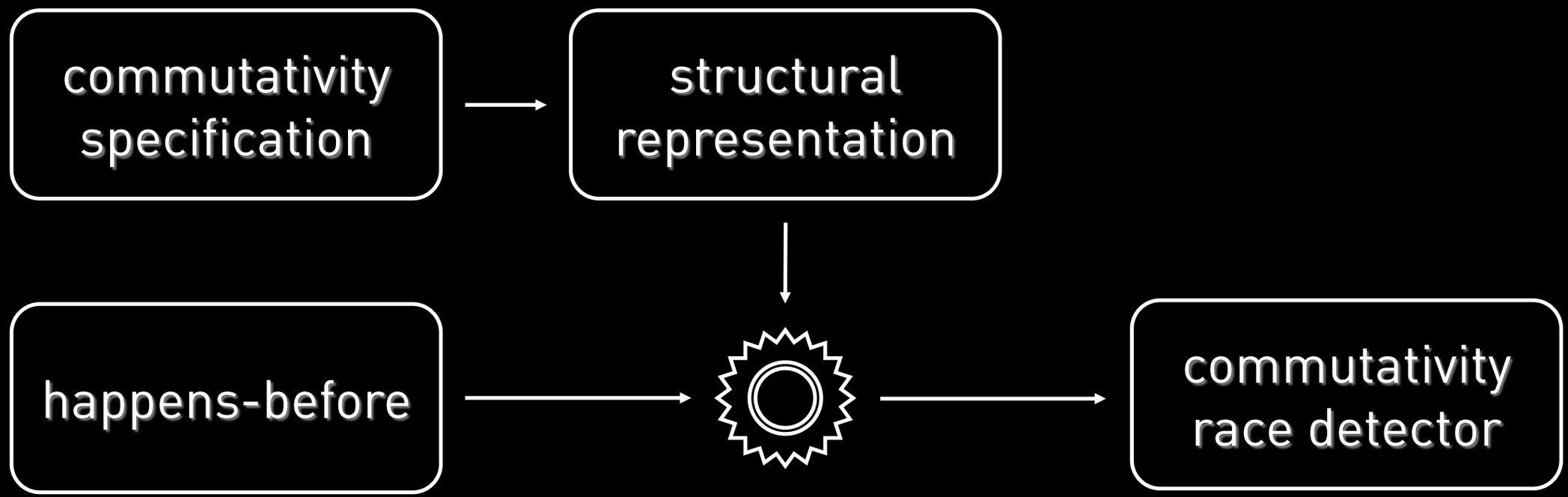


No Commutativity Races

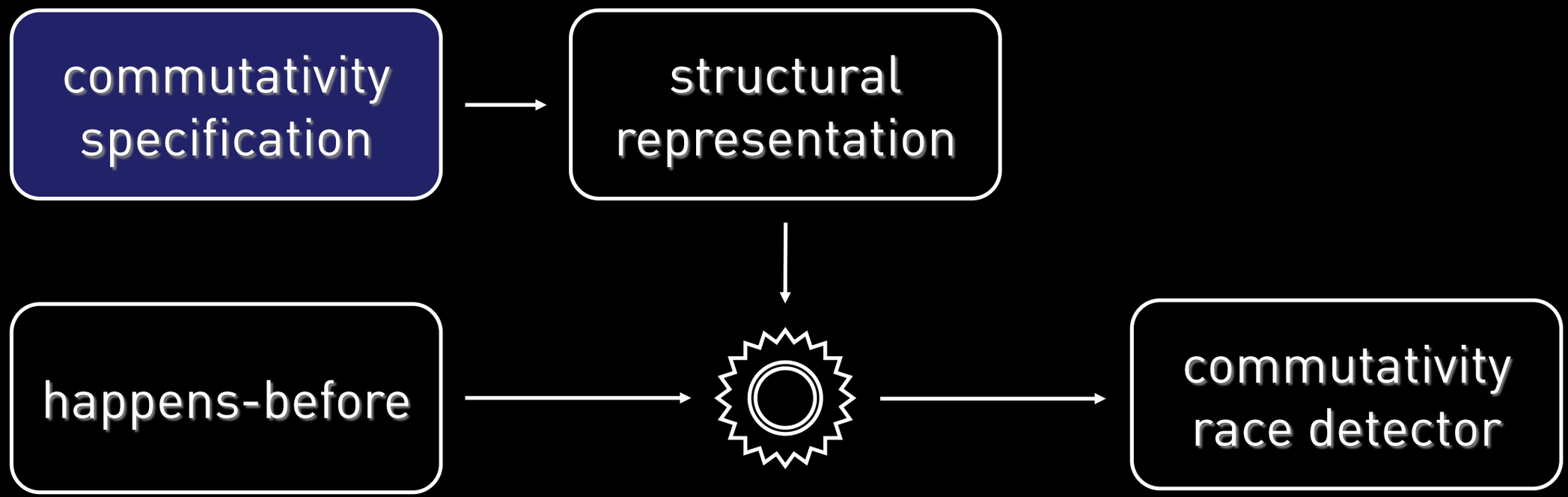
```
ConcurrentHashMap data = new ConcurrentHashMap();

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

Commutativity Race Detection



Commutativity Race Detection



Hashmap commutativity spec

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

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

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

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

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

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

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

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

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

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

put(‘a’,‘tom’)/nil ⋈ size()/1

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

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

put(‘b’,‘lol’)/cat ⋈ size()/1

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

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", Deokhwan Kim & Martin Rinard, ACM PLDI'11

Commutativity Race Detection

Naïve Algorithm

Initially `Seen` $= \varepsilon$

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

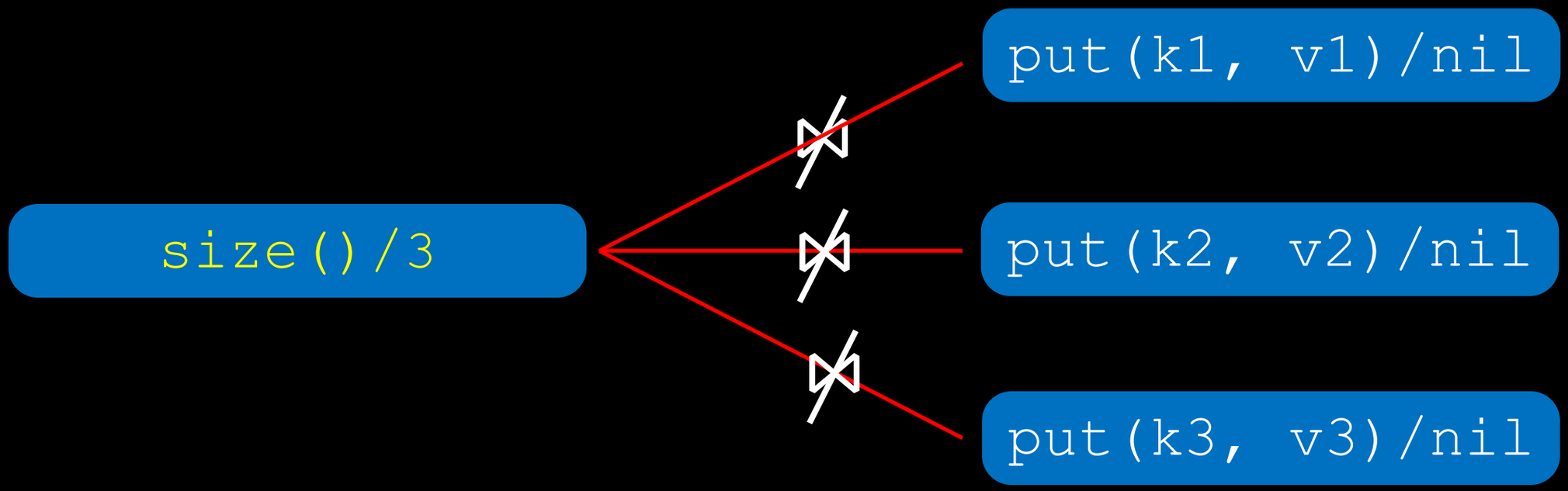
2. Does not explore commutativity sharing between operations

Commutativity Race Detection

Naïve Algorithm

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

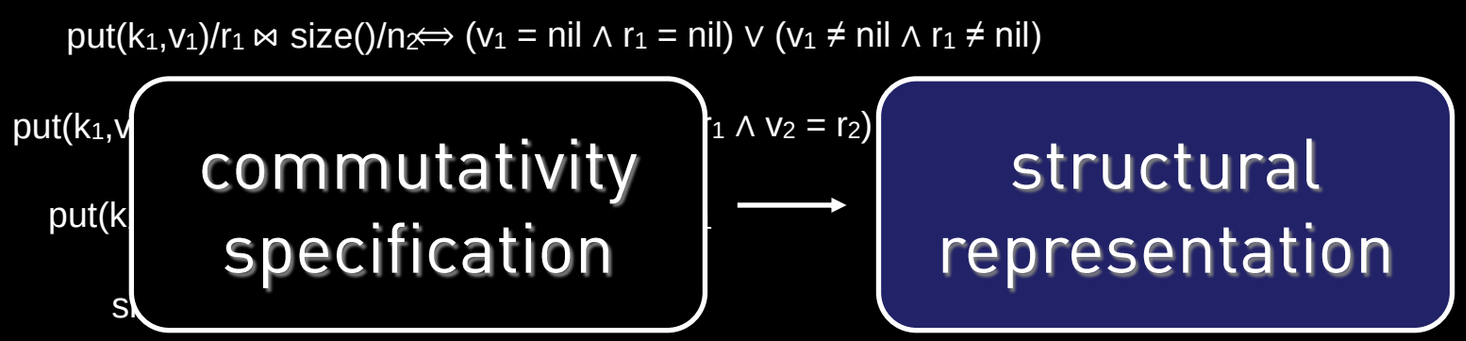
Upon encountering `size()/3` , we will perform 3 checks:



Yet, what matters is whether a concurrent resize has occurred

Wanted: leverage sharing between observed operations

Commutativity Race Detection



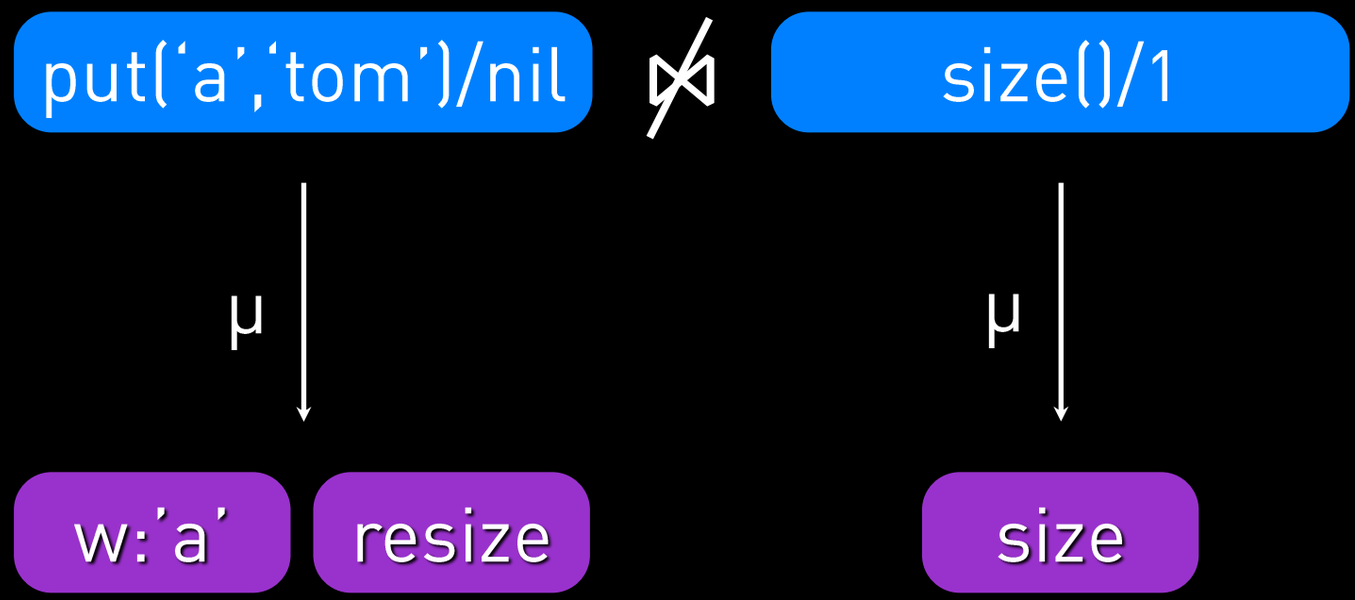
Micro-ops representation

operations

put(‘a’,‘tom’)/nil ⋪ size()/1

Micro-ops representation

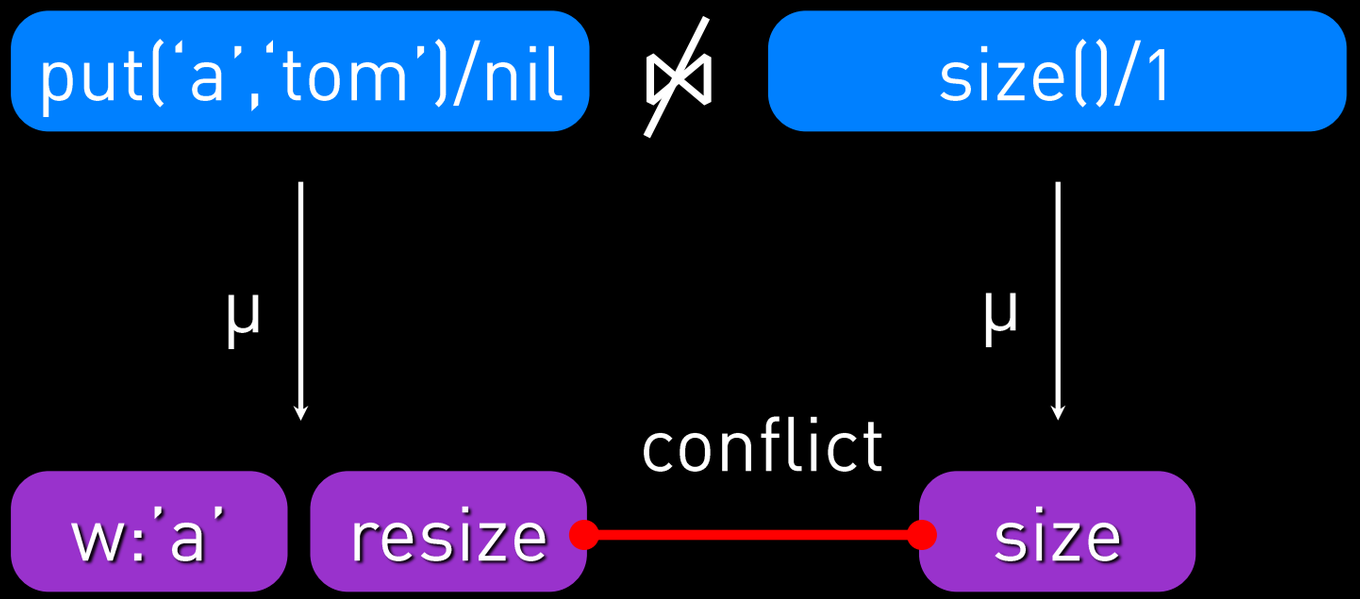
operations



μ-operations

Micro-ops representation

operations



μ-operations

From logic to μ-operations

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

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

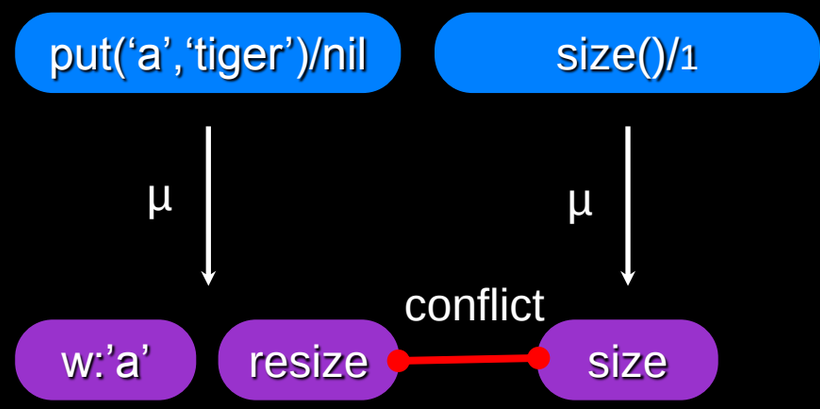
$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \quad \Leftrightarrow k1 \neq k2 \vee v_1 = r_1$

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

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

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

VS.



From logic to μ-operations

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

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

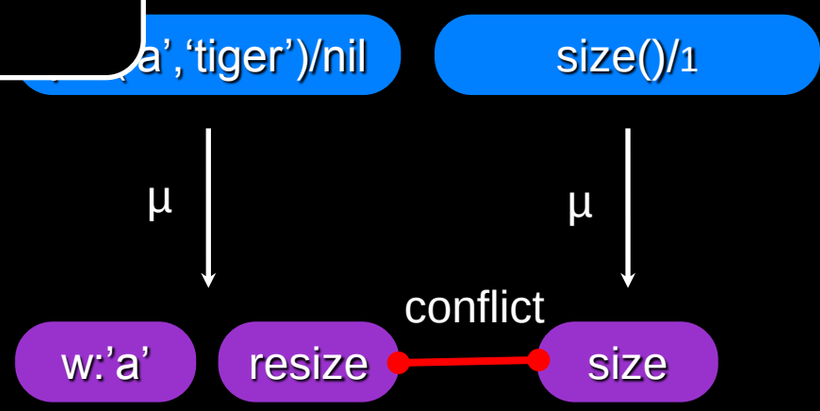
$put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 \quad \Leftrightarrow k1 \neq k2 \vee v_1 = r_1$

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

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

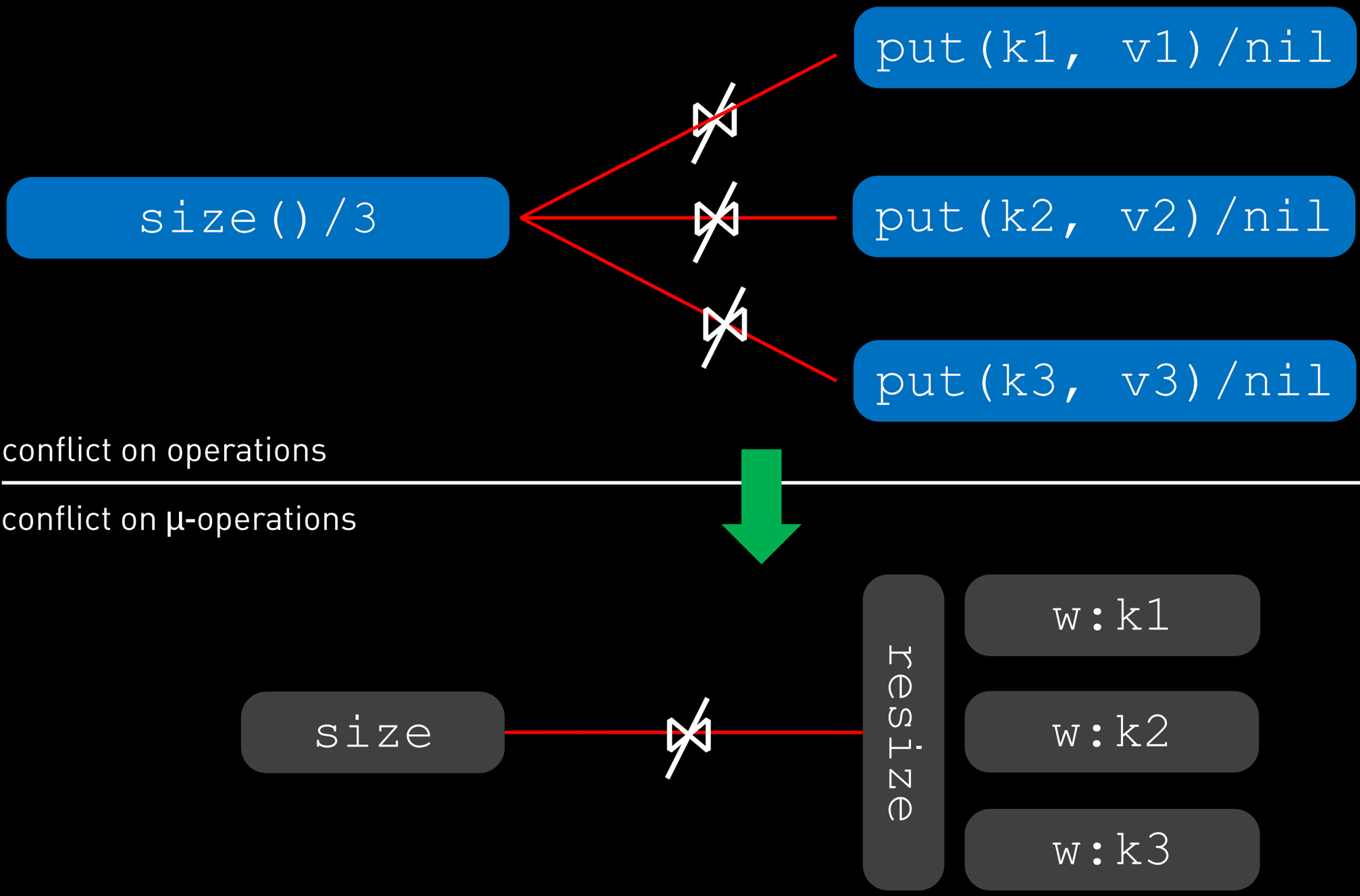
$get(k_1)/v_1 \bowtie get(k_2)/v_2$

commutativity compiler

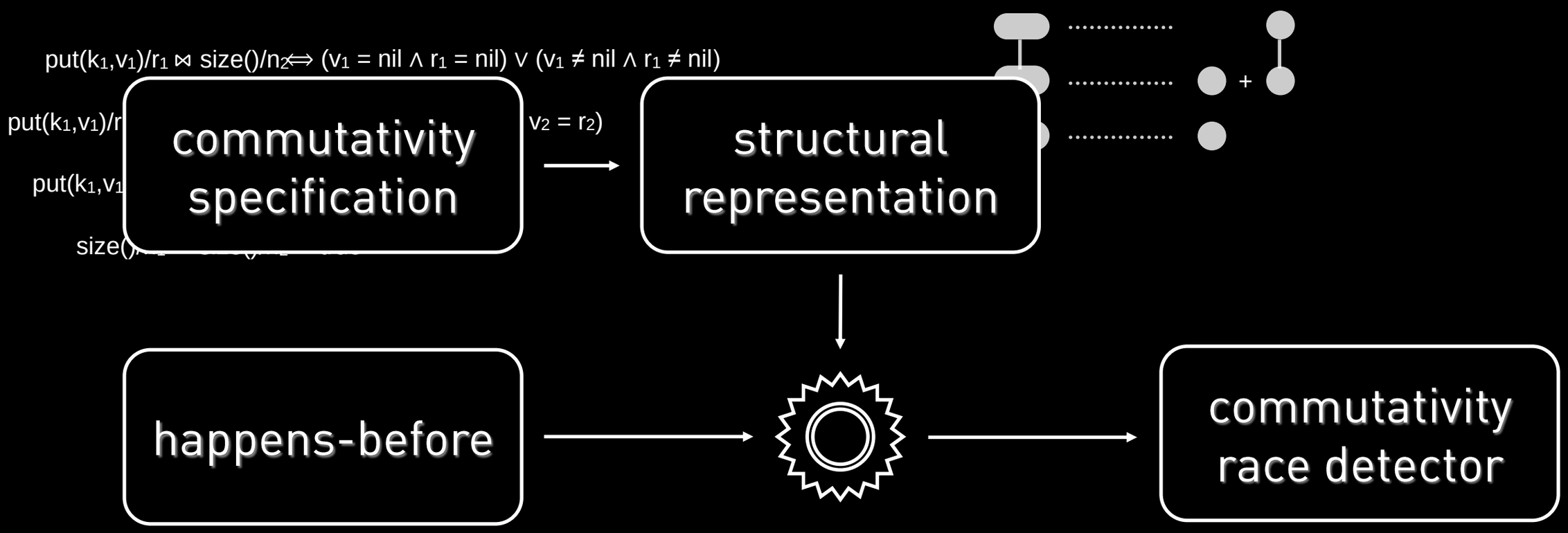


Commutativity Compiler

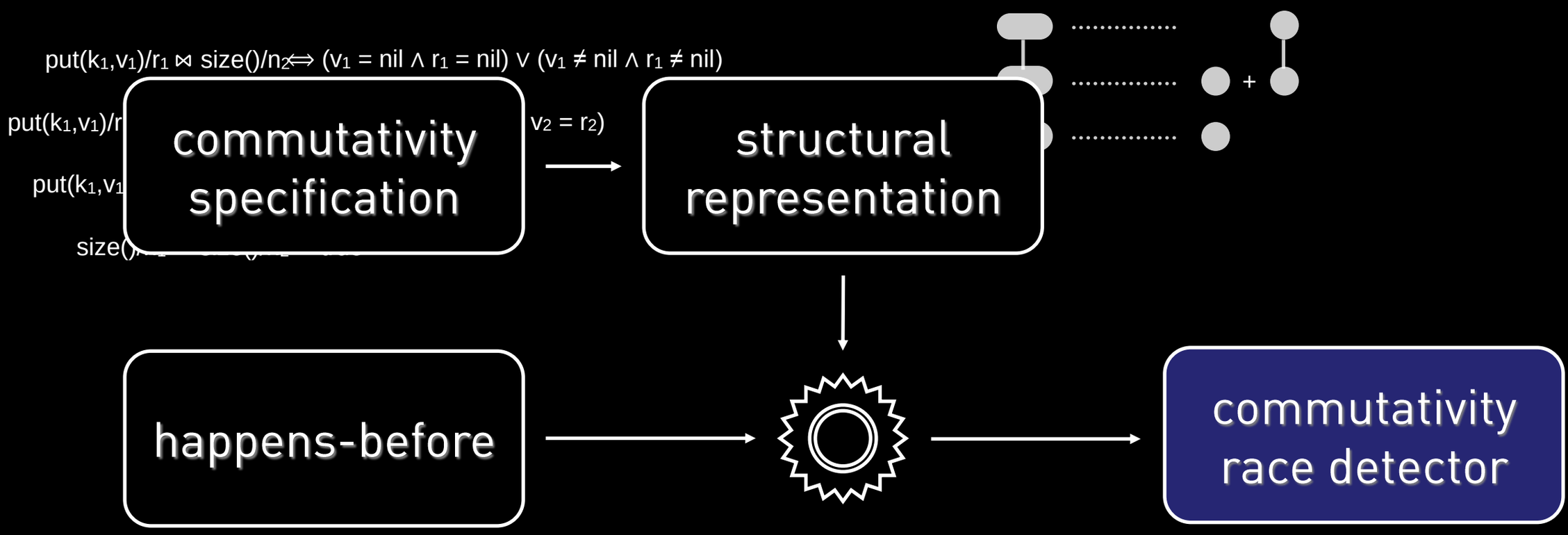
Benefits



Commutativity Race Detection



Commutativity Race Detection



Key idea: associate vector clocks with μ-operations

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

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

Example:

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

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 $V1$

Any predicate over $V2$

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)

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

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

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

Not $X \vee X$

Example:

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

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 from:

Exploiting the commutativity lattice, Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, Keshav Pingali, ACM PLDI 2011

Classic Read-Write 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:

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

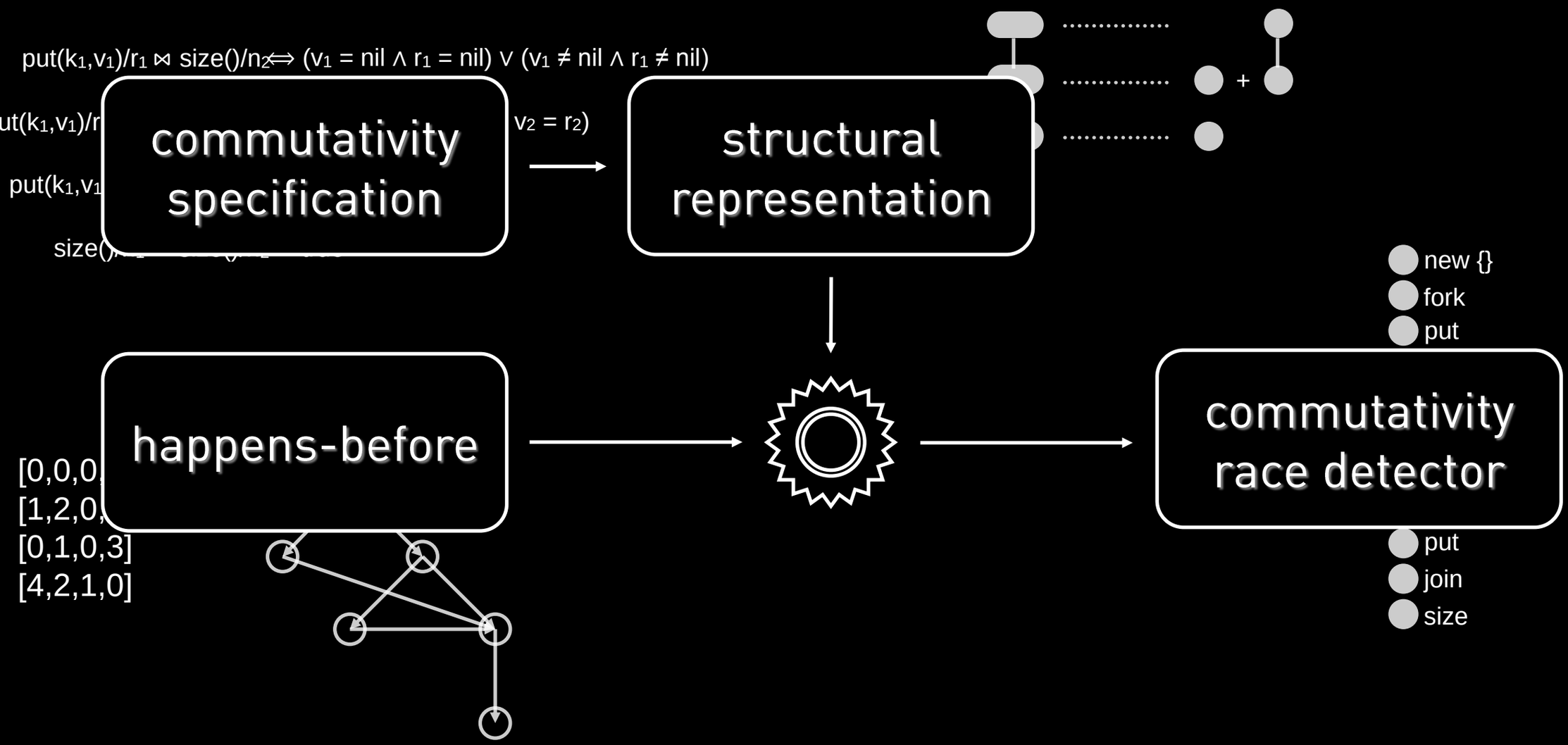
$read()/r_1 \bowtie write(v_2) \quad \Leftrightarrow \quad r_1 = v_2$

$write(v_1) \bowtie write(v_2) \quad \Leftrightarrow \quad v_1 = v_2$

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)

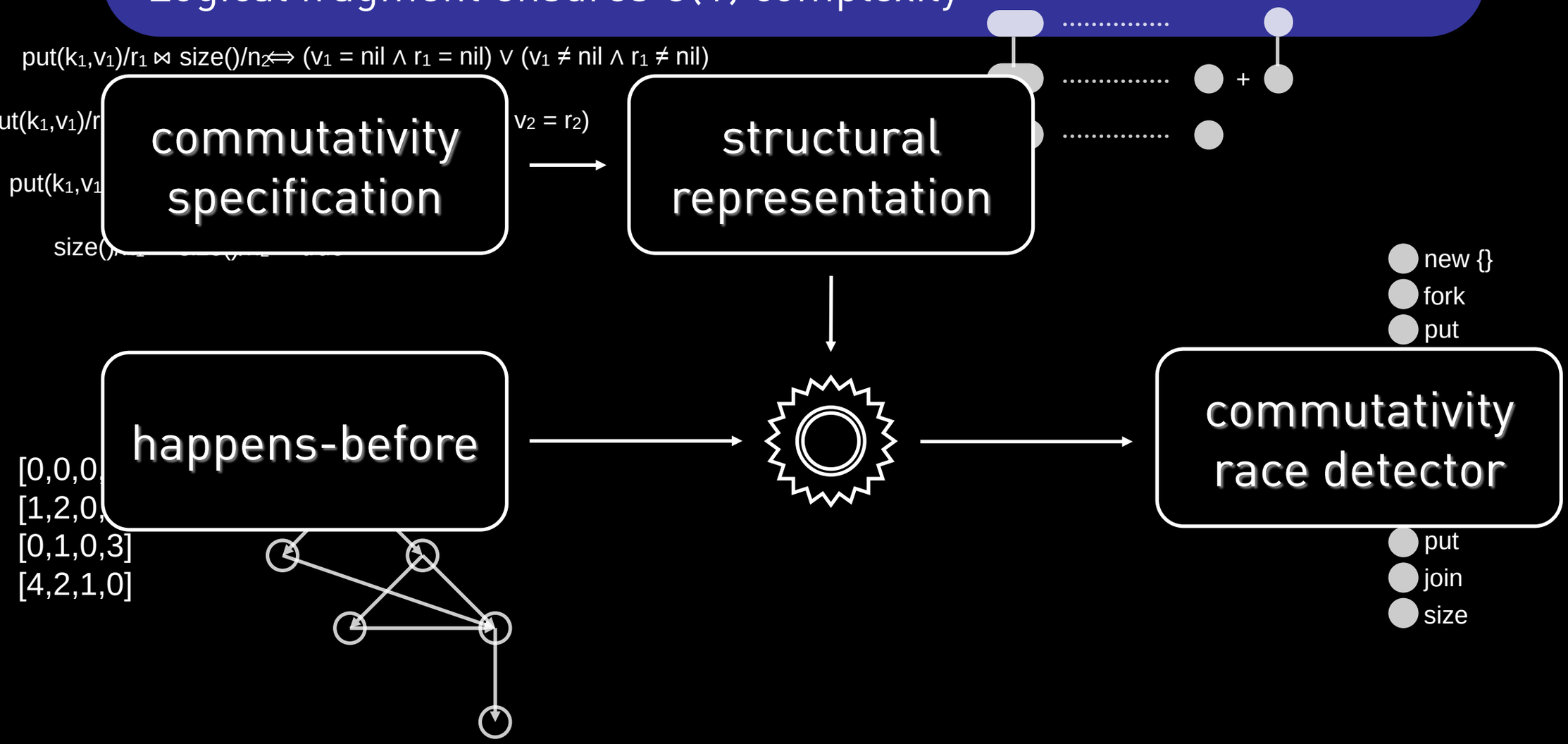
Commutativity race detection

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



More Applications

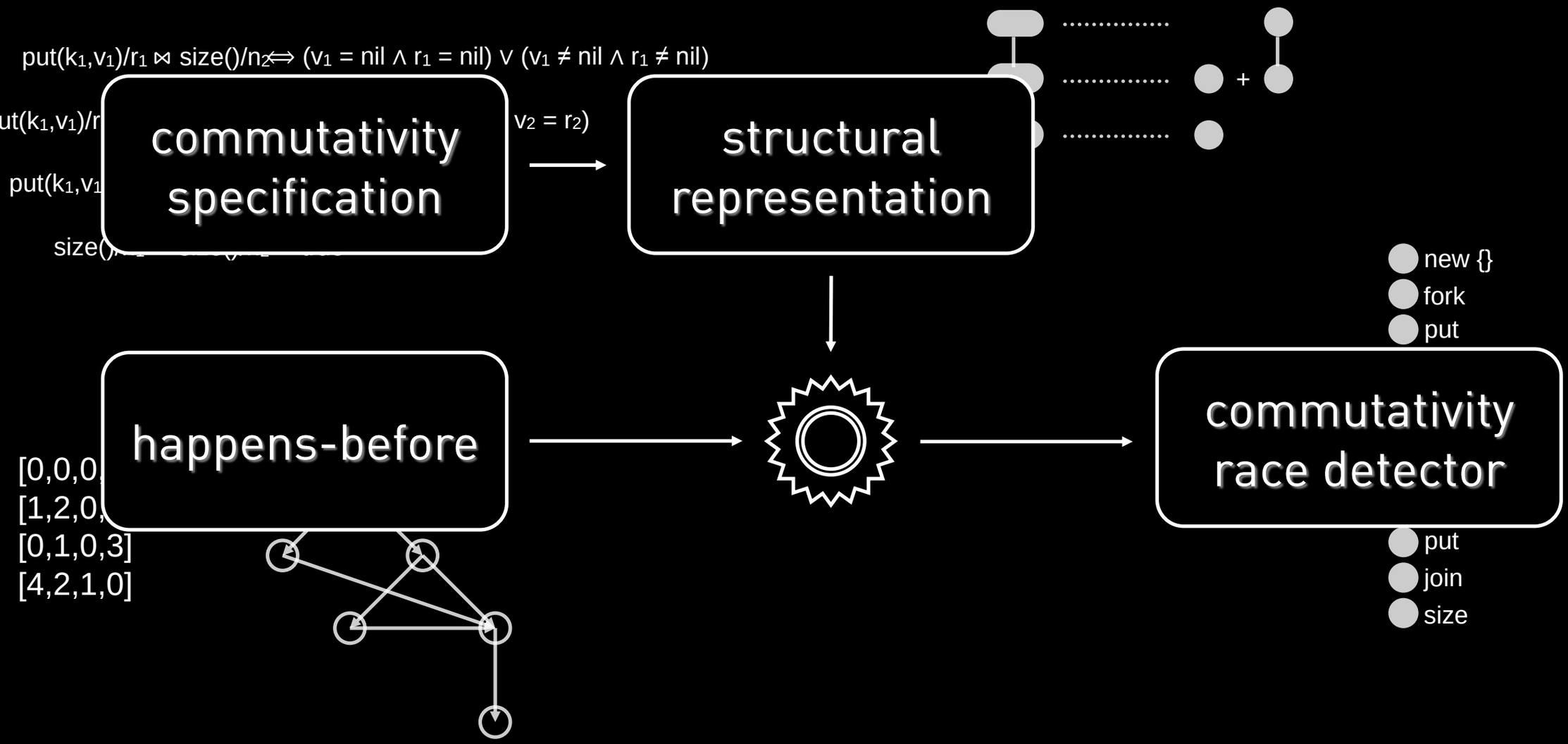
Software Defined Networks

Partial Order Reduction

Optimistic Concurrency Control

Eventual Consistency

Open Problems

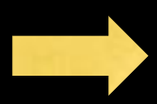


Question 1:
Can we learn the commutativity spec?

Specification of a Map ADT

?

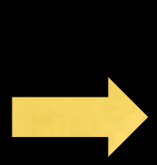
Automatic Inference



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

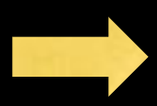
Question 1:
Can we learn the commutativity spec?

Specification of a Map ADT



?

Automatic Inference

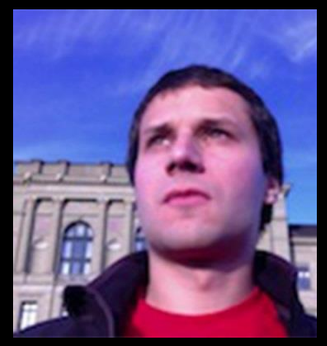


$$
\begin{aligned}
put(k_1,v_1)/r_1 \bowtie size()/n_2 &\iff (v_1 = nil \wedge r_1 = nil) \vee (v_1 \neq nil \wedge r_1 \neq nil)\\
put(k_1,v_1)/r_1 \bowtie put(k_2,v_2)/r_2 &\iff k_1 \neq k_2 \vee (v_1 = r_1 \wedge v_2 = r_2)\\
put(k_1,v_1)/r_1 \bowtie get(k_2)/v_2 &\iff k_1 \neq k_2 \vee v_1 = r_1\\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

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

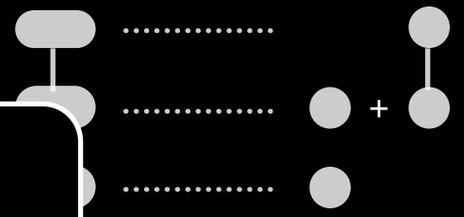
put(k_1,v_1

size()

commutativity specification

v_2 = r_2)

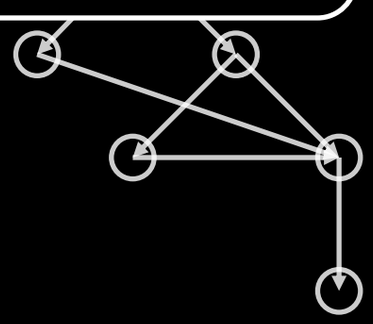
structural representation



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:

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

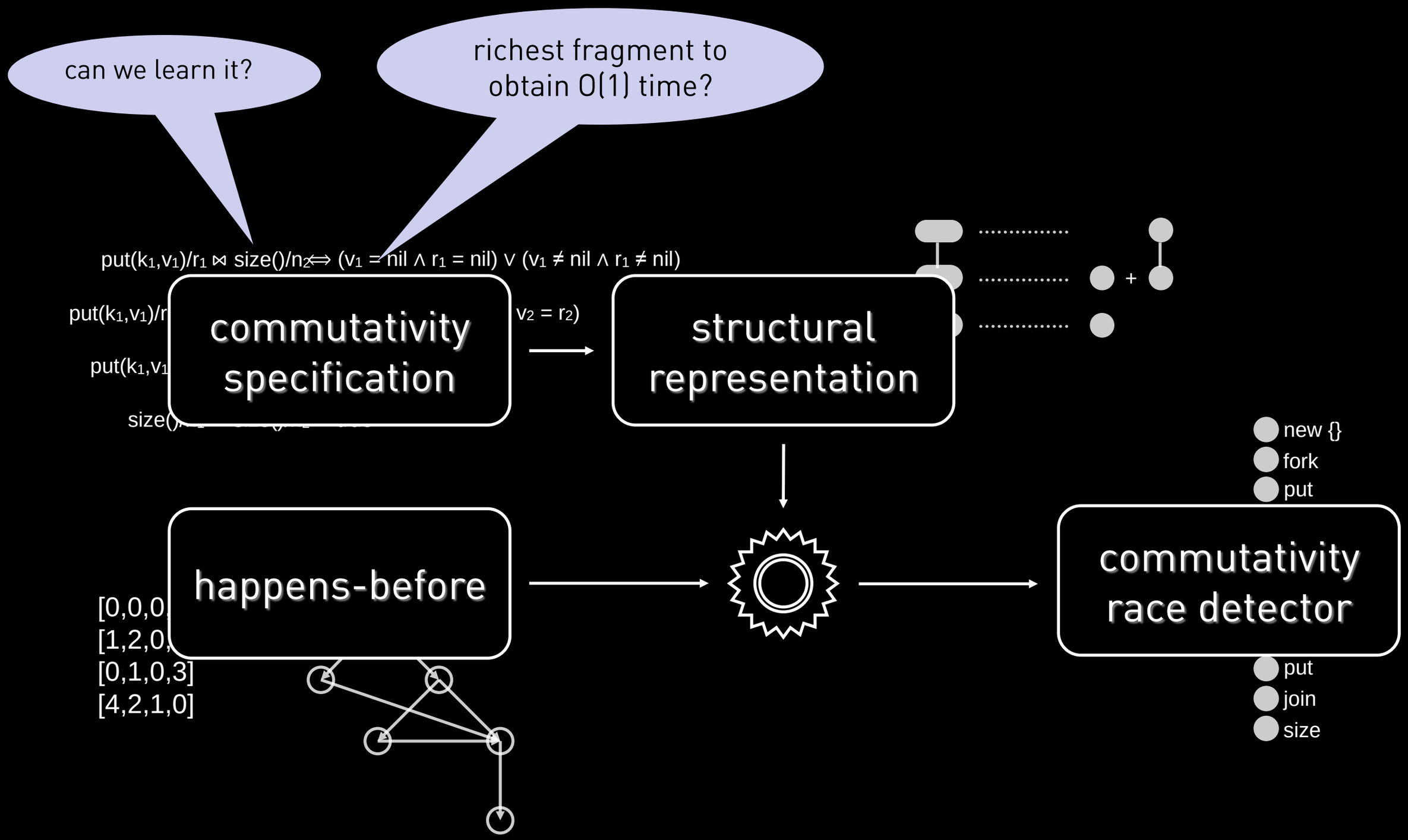
$$B ::= P_{V1} \mid P_{V2} \mid \neg B \mid B \wedge B \mid B \vee B \mid \text{true} \mid \text{false}$$

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

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

What about O(logN)?

Open Problems



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

can we learn it?

richest fragment to obtain O(1) time?

compiler optimizations exploiting fragment

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

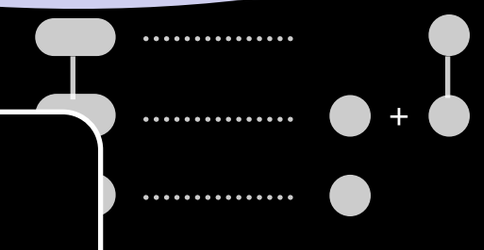
put(k_1,v_1)/r … v_2 = r_2)

put(k_1,v_1

size()/…

commutativity specification

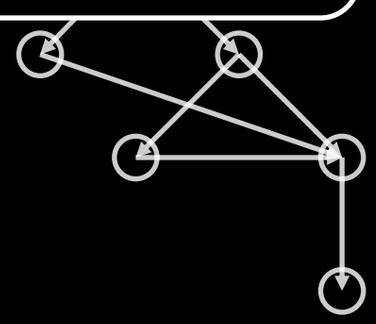
structural representation



new {}
fork
put

happens-before

[0,0,0,…
[1,2,0,…
[0,1,0,3]
[4,2,1,0]



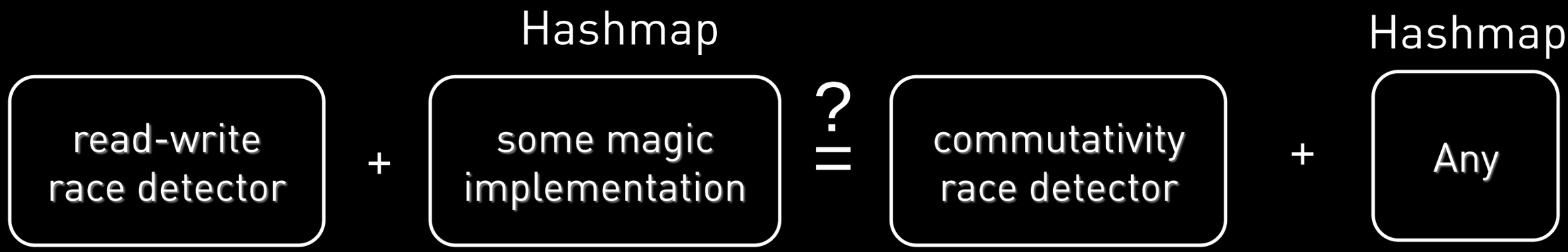
commutativity race detector

put
join
size

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)

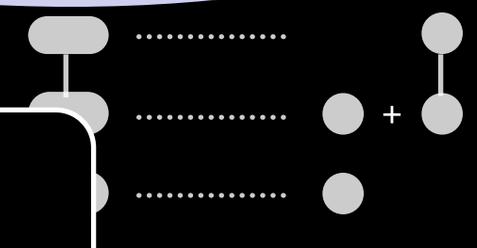
put(k_1,v_1)/r … v_2 = r_2)

put(k_1,v_1…

size()/…

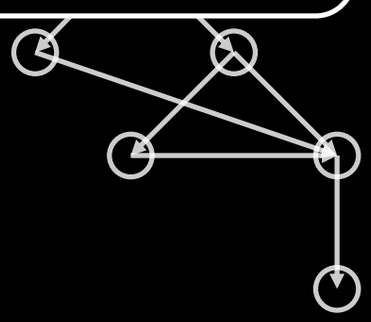
commutativity specification

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