Program Analysis – Lecture 16

Analysis of Dynamic Languages and Web Applications

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Warm-up Quiz

```
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz

```
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

Result: a

https://ilias3.uni-stuttgart.de/vote/KN2I

Warm-up Quiz

```
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

a is a function that returns the second argument, i.e., the function itself

Result: a

https://ilias3.uni-stuttgart.de/vote/KN2I

Dynamic Languages

- **Not a formally defined term**
- **Here means**
  - Dynamically typed languages
  - Languages where usually static properties of a program may change at runtime
  - Languages that facilitate dynamic code generation and code loading

Dynamic Languages

- **Not a formally defined term**
- **Here means**
  - Dynamically typed languages
  - Languages where usually static properties of a program may change at runtime
  - Languages that facilitate dynamic code generation and code loading

Typical examples: Lisp, Python, JavaScript

Challenges

Static analysis is extremely hard

- Absence of types
- Code may change at runtime
- New code many appear at runtime

Challenges

Static analysis is extremely hard

- Absence of types
- Code may change at runtime
- New code many appear at runtime

Makes most static analysis results highly imprecise

- Call graph
- Points-to information
- Data flow

Example

```
if (someCond)
  foo = function() { /*...*/ };
else
  foo = function() { /*...*/ };

// ...

eval("foo = function() { /*...*/ }");

function bar() {
    foo();
}
```

Example

```
if (someCond)
  foo = function() { /*...*/ };
else
  foo = function() { /*...*/ };

// ...

eval("foo = function() { /*...*/ }");

function bar() {
    foo();
}
```

Multiple `foo` functions: Not clear which one gets called

Example

```
if (someCond)
  foo = function() { /*...*/ };
else
  foo = function() { /*...*/ };

// ...

eval("foo = function() { /*...*/ }");

function bar() {
    foo();
}
```

Dynamically loaded code that redefines the function

Example

```
if (someCond)
  foo = function() { /*...*/ };
else
  foo = function() { /*...*/ };

// ...

eval("foo = function() { /*...*/ }");

function bar() {
    foo();
}
```

This kind of code really exists, e.g., because libraries monkey-patch missing built-in libraries.

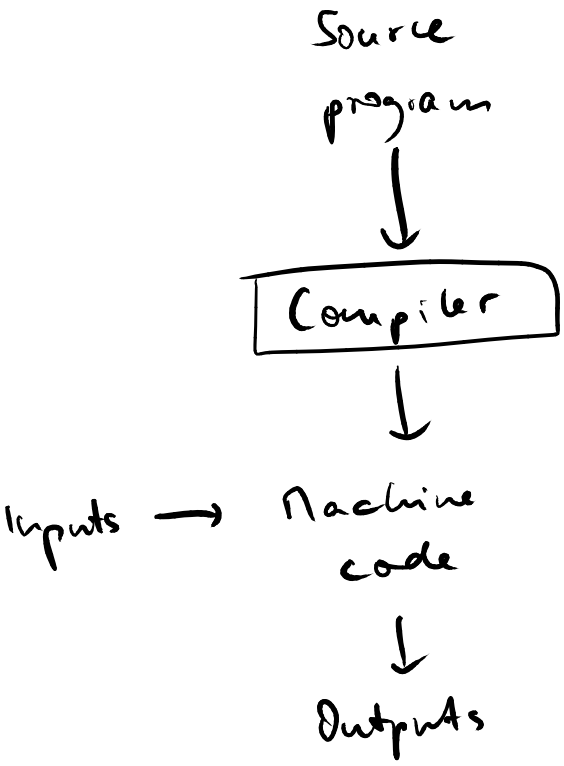
Overview

1. **Type Inference for JIT Compilation**
2. **Race Detection in Web Applications**

Mostly based on these papers:

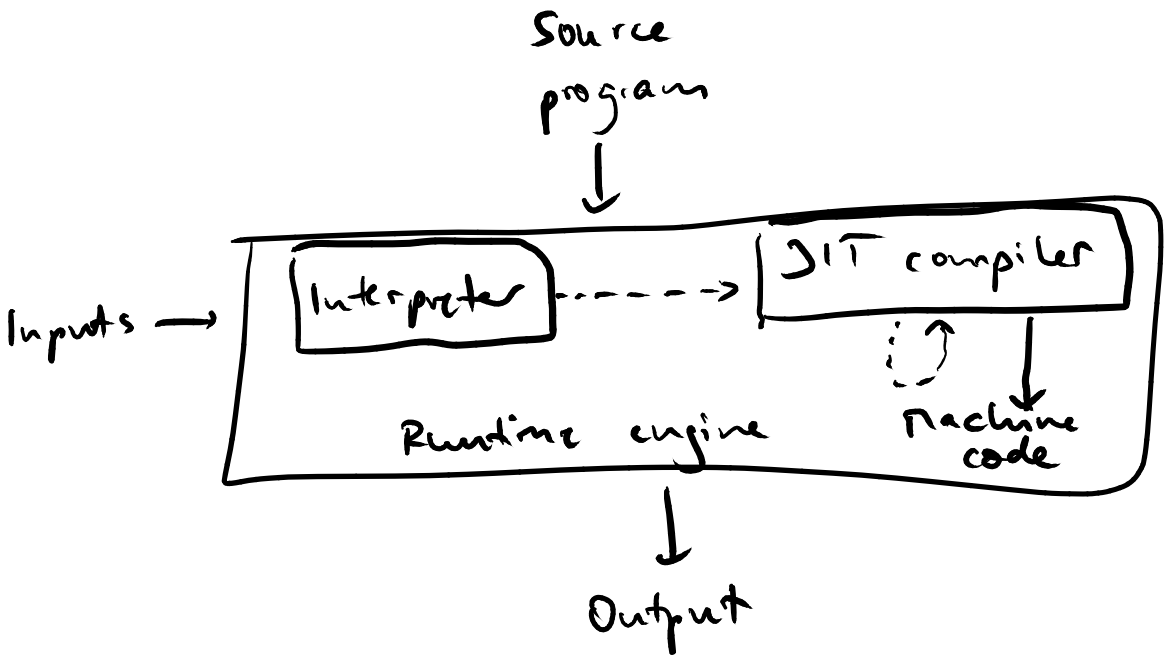
- *Fast and Precise Hybrid Type Inference for JavaScript*, Hackett and Guo, PLDI 2012
- *Race Detection for Web Applications*, Petrov et al., PLDI 2012
- *Effective Race Detection for Event-Driven Programs*, Raychev et al., OOPSLA 2013

Traditional compilation



JIT compilation

"just in time"



Why Compile Just-in-Time?

- **Static compilation: Need to cover all possible behaviors of a piece of code**
  - E.g., + may mean integer addition, double addition, string concatenation, etc.
- **At runtime: Can specialize machine code to behavior that code actually has**
  - Gives much faster code

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

In general, plus operator has many meanings

Example

[Excerpt of JavaScript spec]

11.6.1 The Addition operator (+)

The addition operator either performs string concatenation or numeric addition.

The production *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating MultiplicativeExpression.
4. Let *rval* be GetValue(*rref*).
5. Let *lprim* be ToPrimitive(*lval*).
6. Let *rprim* be ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   a. Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*)
8. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*). See the Note below 11.6.3.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 5 and 6. All native ECMAScript objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Host objects may handle the absence of a hint in some other manner.

NOTE 2 Step 7 differs from step 3 of the comparison algorithm for the relational operators (11.8.5), by using the logical-or operation instead of the logical-and operation.

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

In general, plus operator has many meanings

Here, plus operator always means integer addition

Inferred Types for Faster Code

- **Idea of type inference-based JIT compilation**
  - Lightweight static analysis to infer likely types
  - Optimistically compile code assuming these types
  - Insert runtime checks to handle unexpected types

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

`Box` objects can have integer property `p`

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

`a` is an array and it can have elements of type `Box`

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

`a` within `use` can be an array `Box` objects

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

`v` can be an `integer`

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

+ can mean integer addition

Type Inference

- **Analyze one function at a time**
- **Compute set of possible types for each expression**
- **Model flow of types via constraints**
  - T .. set of types
  - $T_{e1} \supseteq T_{e2}$ .. type of expression $e1$ now includes types of expression $e2$

Type Inference (2)

- **Type propagations are triggered:**
  - At static analyses time
  - When unexpected types occur at runtime
- **Examples**
  - `x = e;` adds constraint $T_x \supseteq T_e$
  - `x.p = e;` adds constraint $prop(o, p) \supseteq T_e$ for all $o \in T_x$

Formal description of analysis rules: See paper by Hackett et al.

Unsound Static Analysis

- **Most static analyses aim at soundness**
  - E.g., report all possible types a variable may have
- **Here: Unsoundness is accepted**
  - Anyway, practically impossible to soundly infer types in JavaScript
  - If inferred types hold true in most executions, generated code is fast in most executions

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Why could this be anything else but integer addition?

Example

```
function Box(v) {
  if (someCond)
    this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Example

```
function Box(v) {
  if (someCond)
    this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Only some `Box` objects have a `p` property.

Array may have holes. Hence, + must handle `undefined`.

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Addition may overflow. If possible, code remains integer addition, but overflow will produce a double.

Handling Unsoundness

- **Guard specialized code with runtime checks**
- **If an unexpected type occurs, de-optimize code and re-compile with support for this type**
  - “Unexpected”: Types neither predicted by type inference nor observed so far at runtime

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
```

Specialized code for integer addition, guarded by check that `v` is integer

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
a.append(new Box("hello")); use(a);
```

Example

```
function Box(v) {
  this.p = v;
}
function use(a) {
  var res = 0;
  for (var i = 0; i < 1000; i++) {
    var v = a[i].p;
    res = res + v;
  }
  return res;
}
var a = [];
for (var i = 0; i < 1000; i++) {
  a[i] = new Box(10);
}
use(a);
a.append(new Box("hello")); use(a);
```

Re-compile code to support integer addition and string concatenation

Implementation and Deployment

- **Implemented in SpiderMonkey JavaScript engine of Firefox**
- **Faster than interpretation and purely profiling-based type specialization**
  - Time saved through specialized code pays for
    - Time spent analyzing and JIT-compiling
    - Time spent de-compiling and re-compiling on unexpected types

Overview

1. **Type Inference for JIT Compilation**
2. **Race Detection in Web Applications** ←

Mostly based on these papers:

- *Fast and Precise Hybrid Type Inference for JavaScript*, Hackett and Guo, PLDI 2012
- *Race Detection for Web Applications*, Petrov et al., PLDI 2012
- *Effective Race Detection for Event-Driven Programs*, Raychev et al., OOPSLA 2013

Motivation

Event-driven programs

- Complex, asynchronous control flow
- Non-determinism

Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Parse input type=”button”

Show button

Parse script define f()

Click button

Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Parse input type=”button”

Show button

Click button

Parse script define f()

Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Parse input type=”button”

Show button

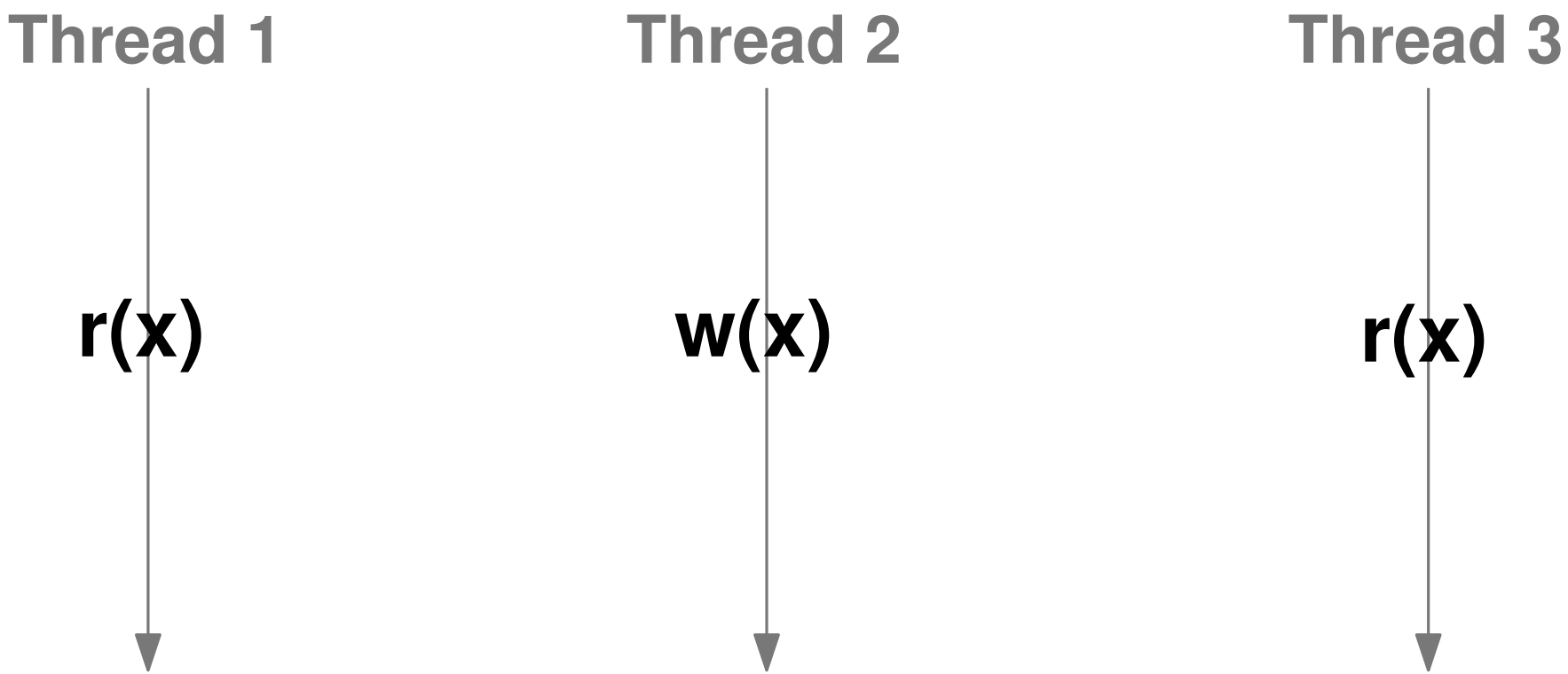
Click button

Crash!

Parse script define f()

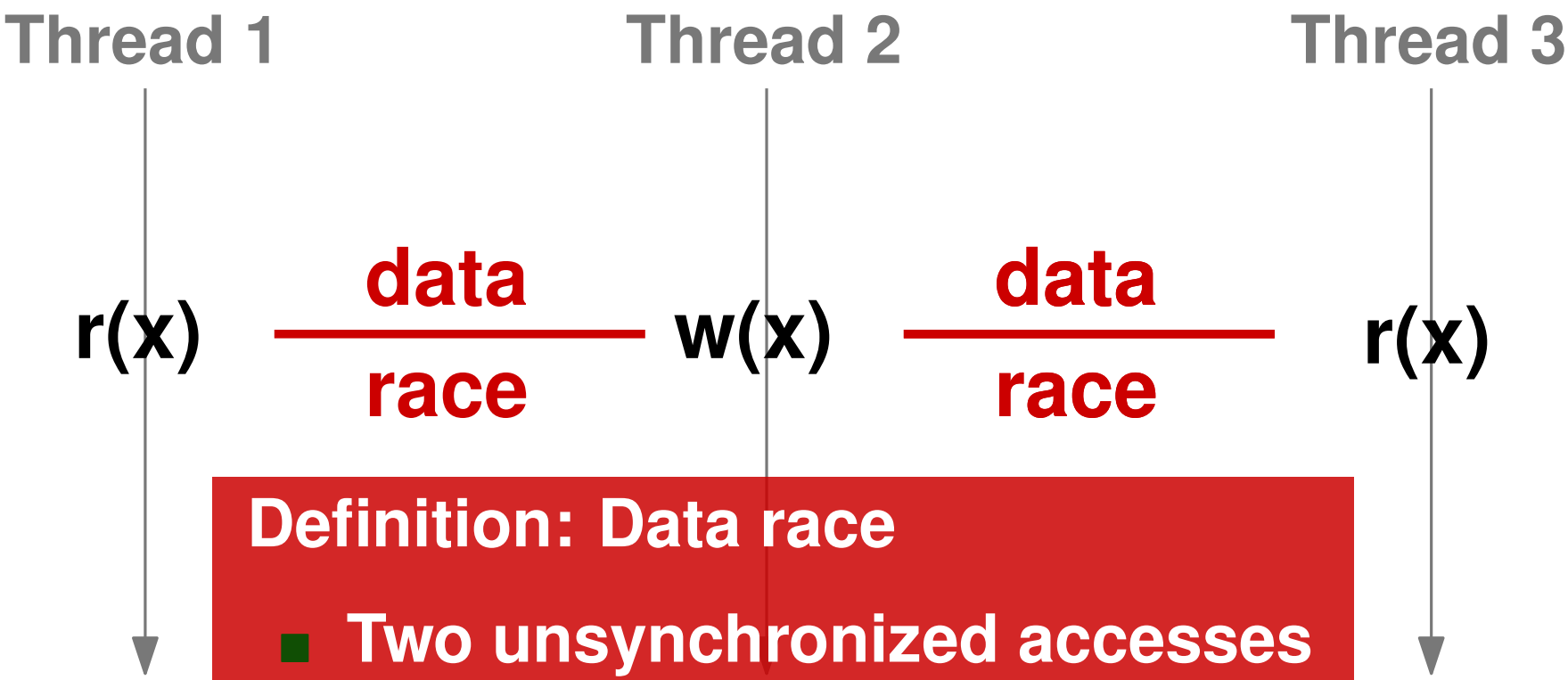
Data Races

Traditionally: Multi-threaded program



Data Races

Traditionally: Multi-threaded program

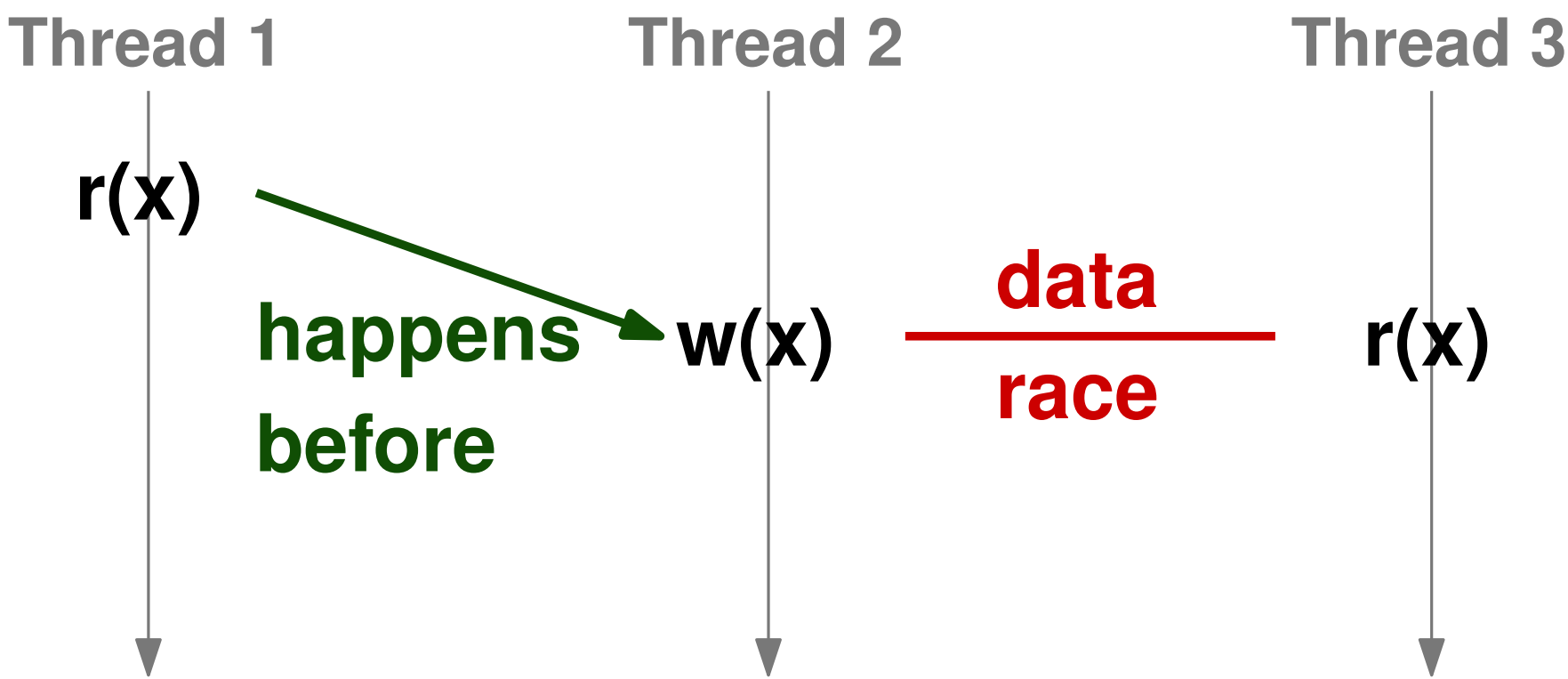


Definition: Data race

- Two unsynchronized accesses to a shared variable
- At least one access is a write

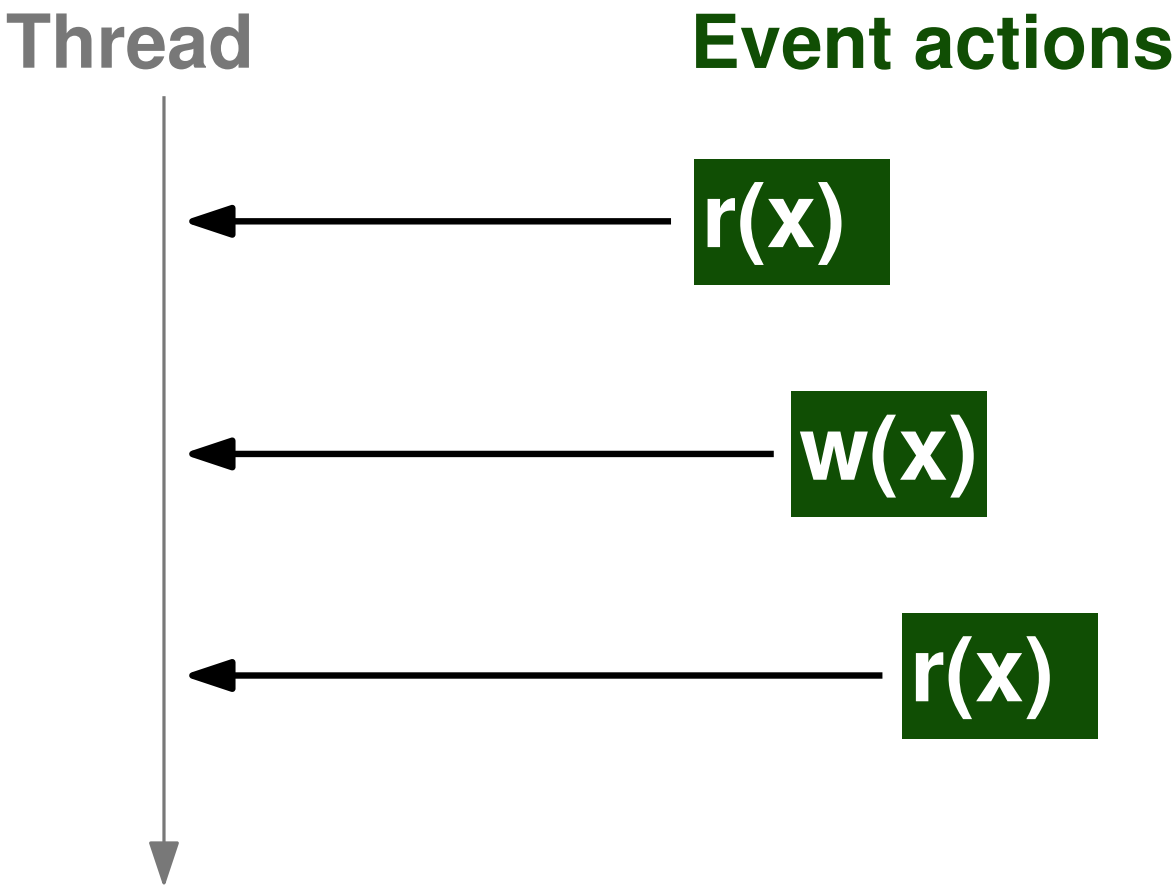
Data Races

Traditionally: Multi-threaded program



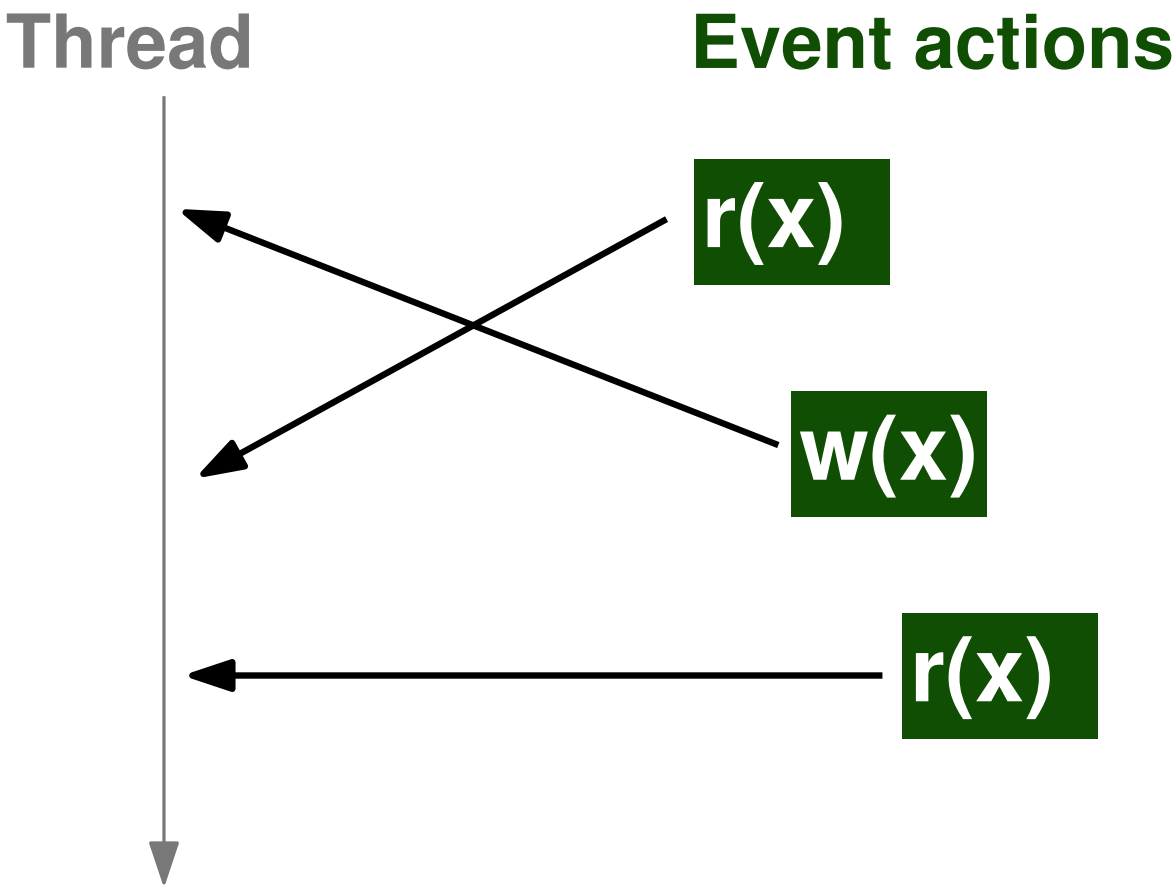
Data Races: Event-Driven

Here: Single-threaded but event-driven



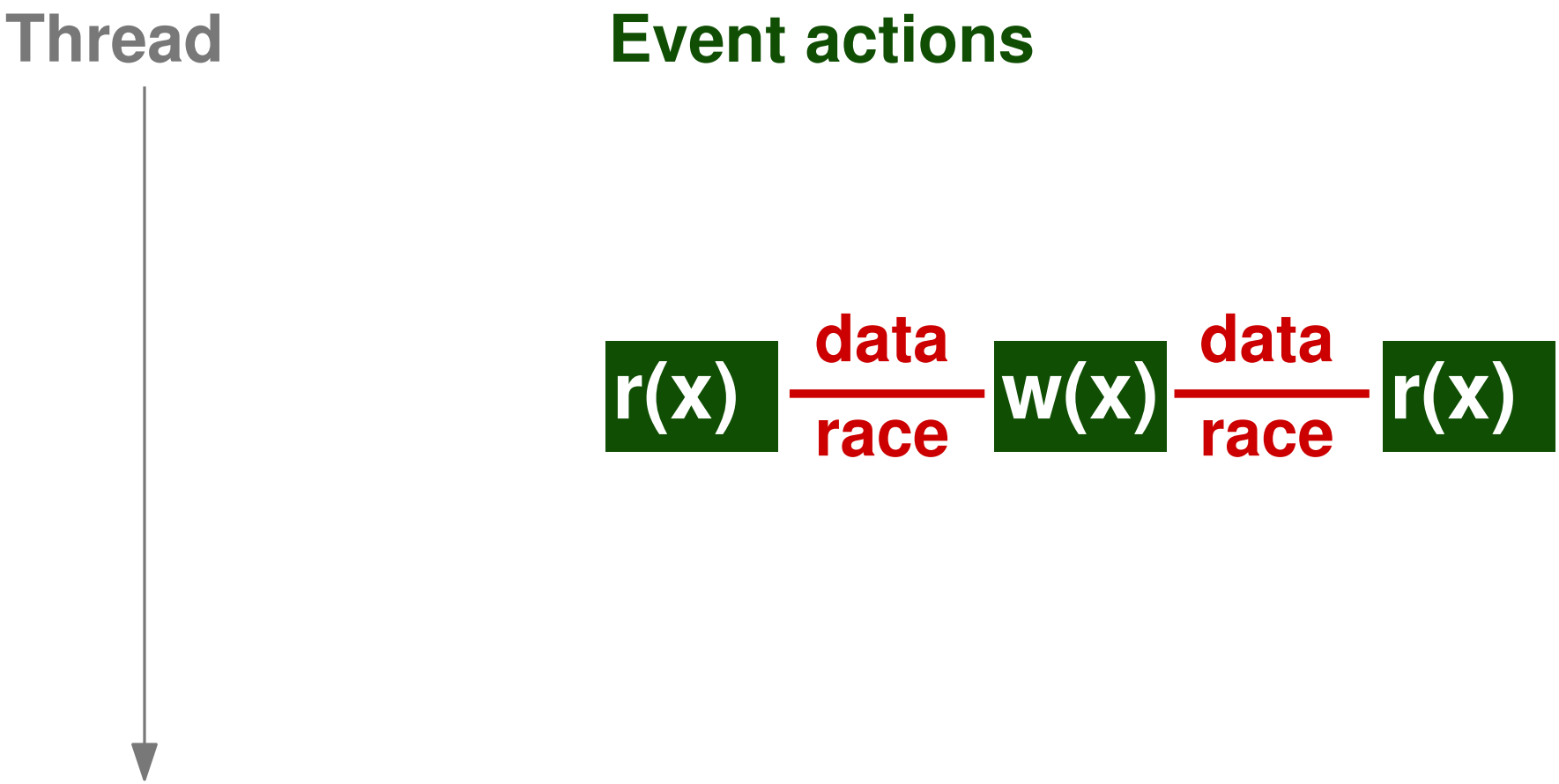
Data Races: Event-Driven

Here: Single-threaded but event-driven



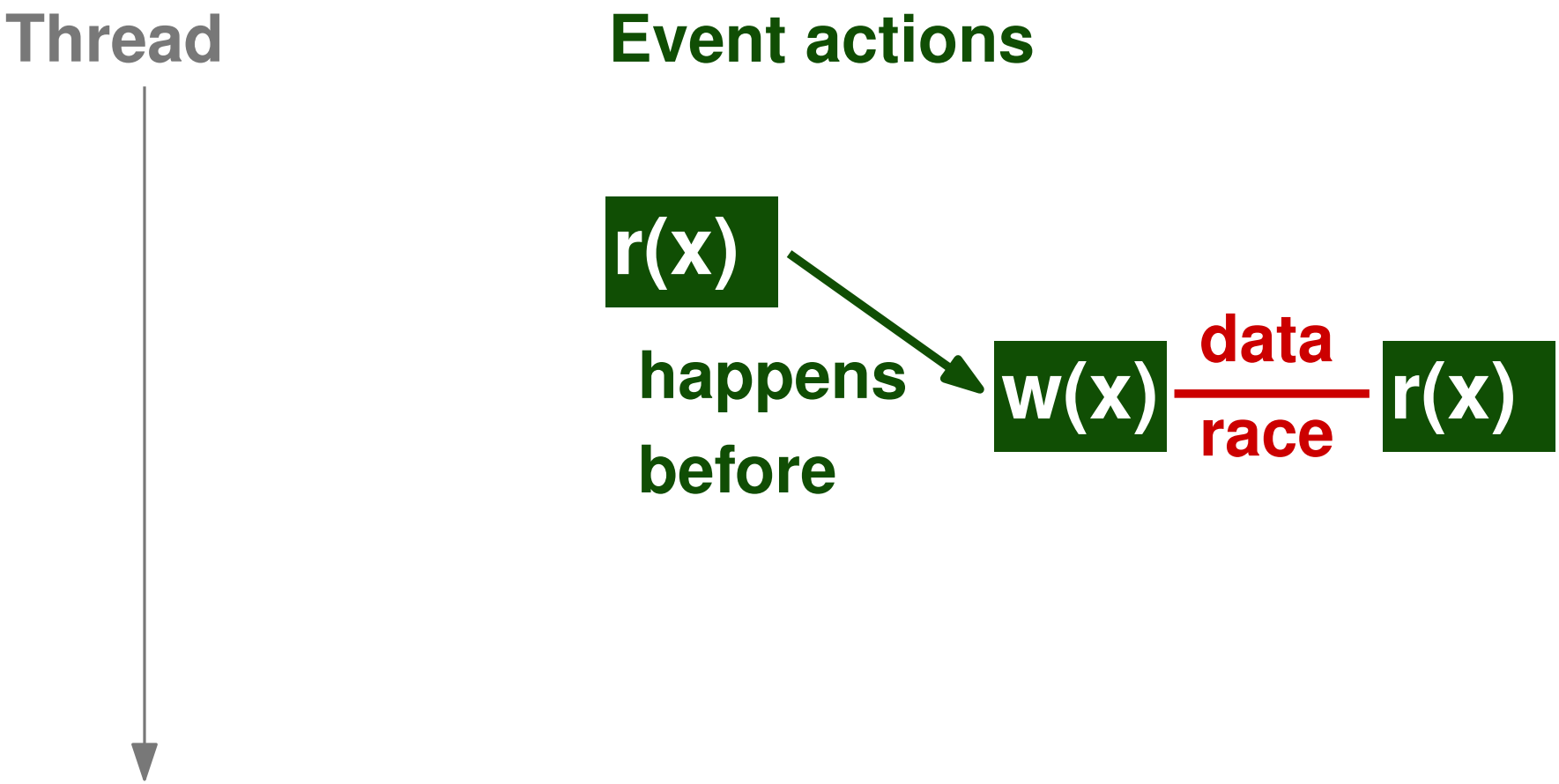
Data Races: Event-Driven

Here: Single-threaded but event-driven

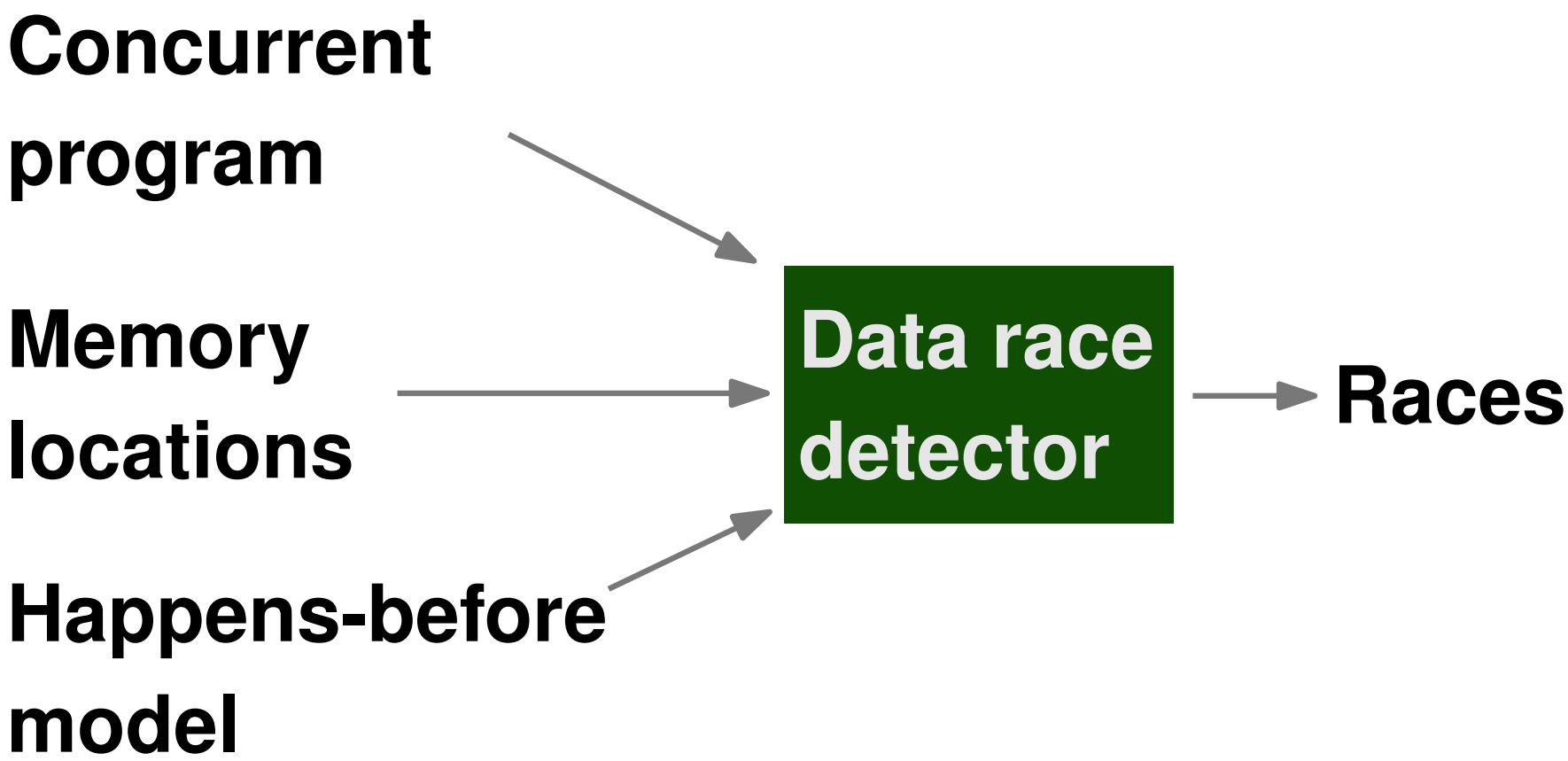


Data Races: Event-Driven

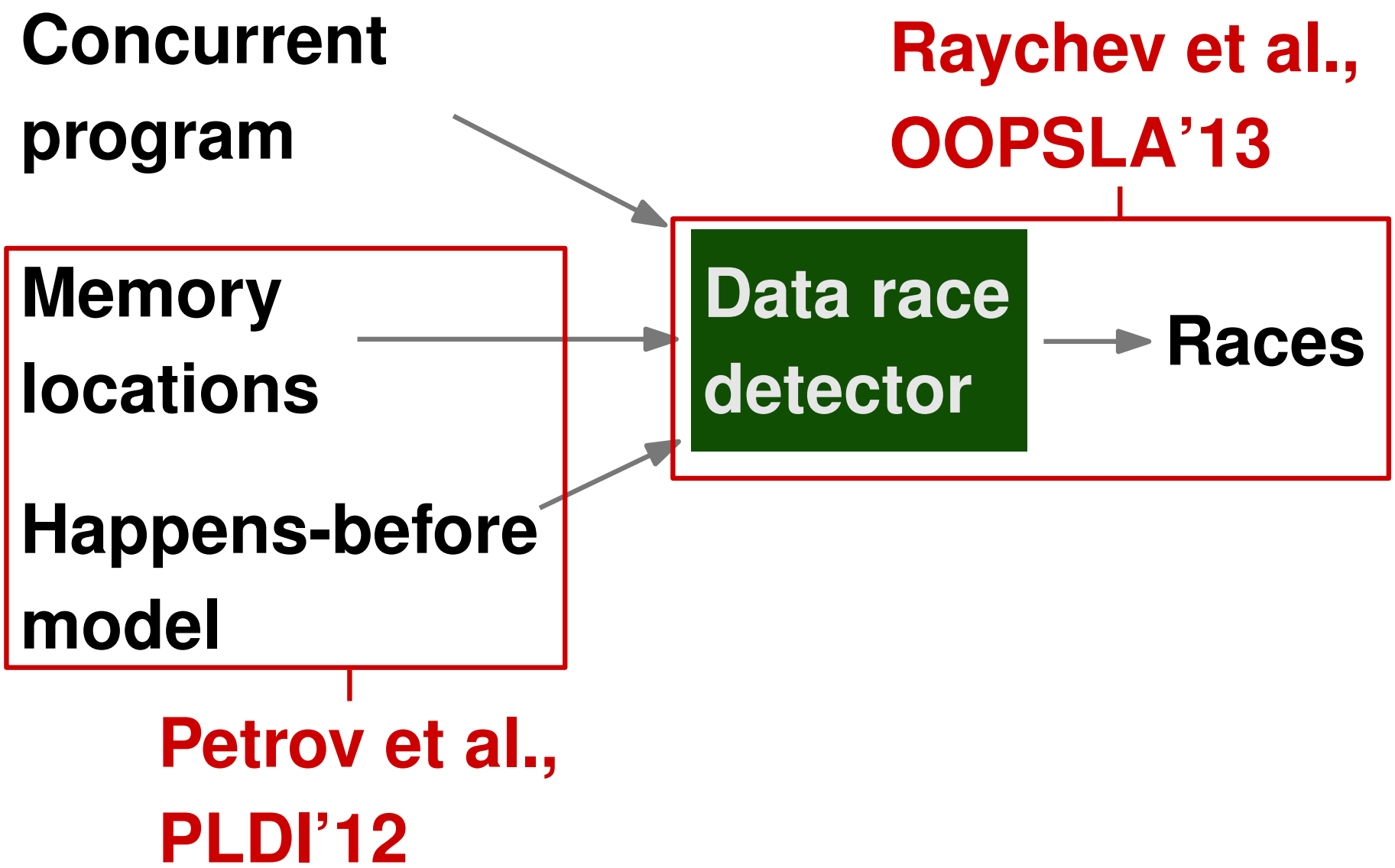
Here: Single-threaded but event-driven



Goal: Data Race Detection



Goal: Data Race Detection



Memory Locations

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Memory Locations

```
<html><body>
<input type="button" onclick="f()"> // w(button), r(f)
<script>
var init = false, y = null; // w(init), w(y)
function f() {              // w(f)
  if (init) alert (y.g);    // r(init), r(y), r(y.g)
}
</script>
<script>
y = { g: 42 };              // w(y), w(y.g)
init = true;                // w(init)
</script>
</body></html>
```

Happens-Before Model

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Happens-Before Model

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Race Detection: Challenges

Precision: Avoid false positives

- Race coverage
- Heuristic classification

Scalability: Avoid vector clock blow-up

- Chain decomposition

Race Detection: Challenges

Precision: Avoid false positives

- Race coverage
- Heuristic classification

Scalability: Avoid vector clock blow-up

- Chain decomposition

Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

w(button)

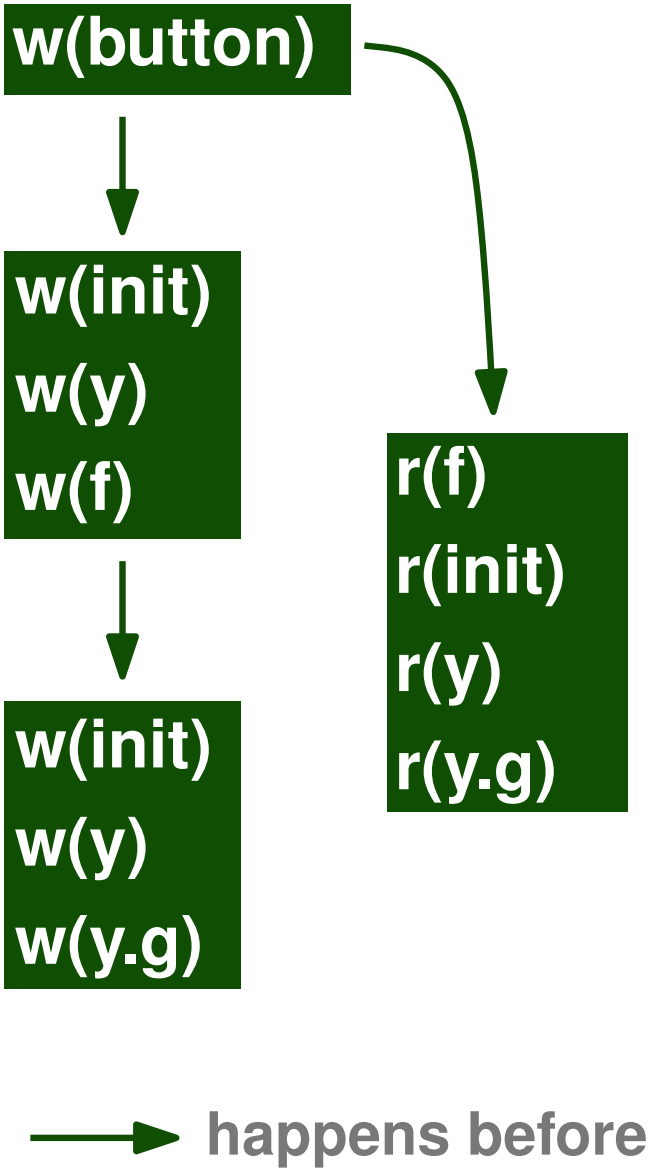
w(init)
w(y)
w(f)

w(init)
w(y)
w(y.g)

r(f)
r(init)
r(y)
r(y.g)

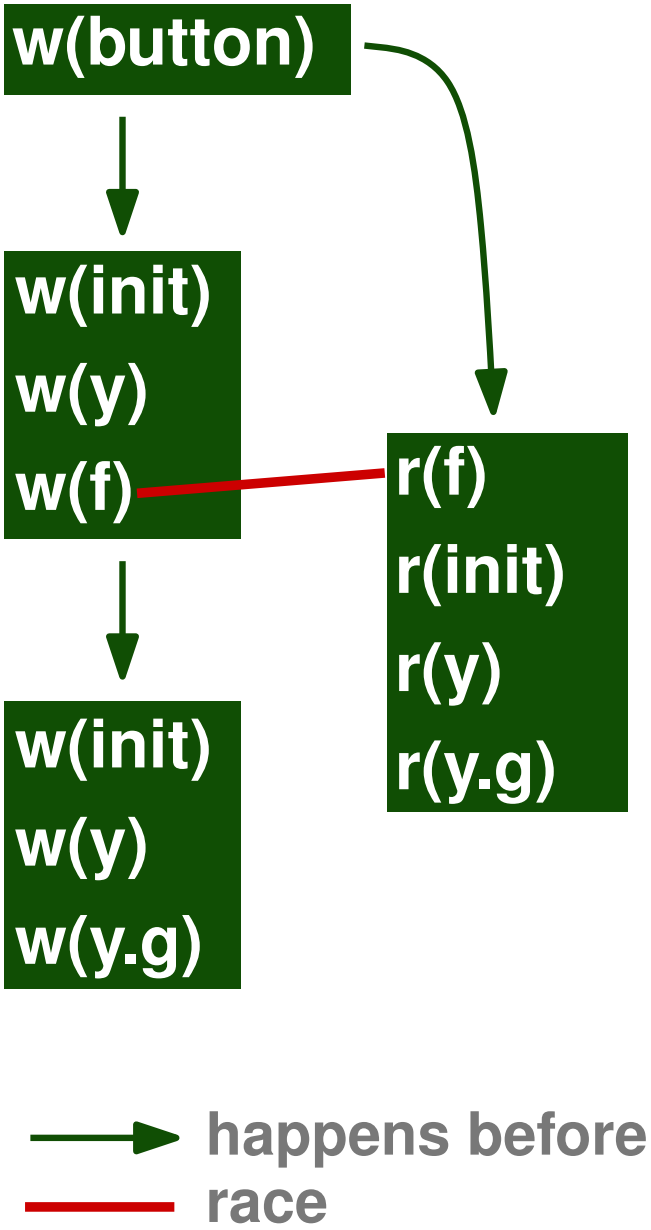
Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



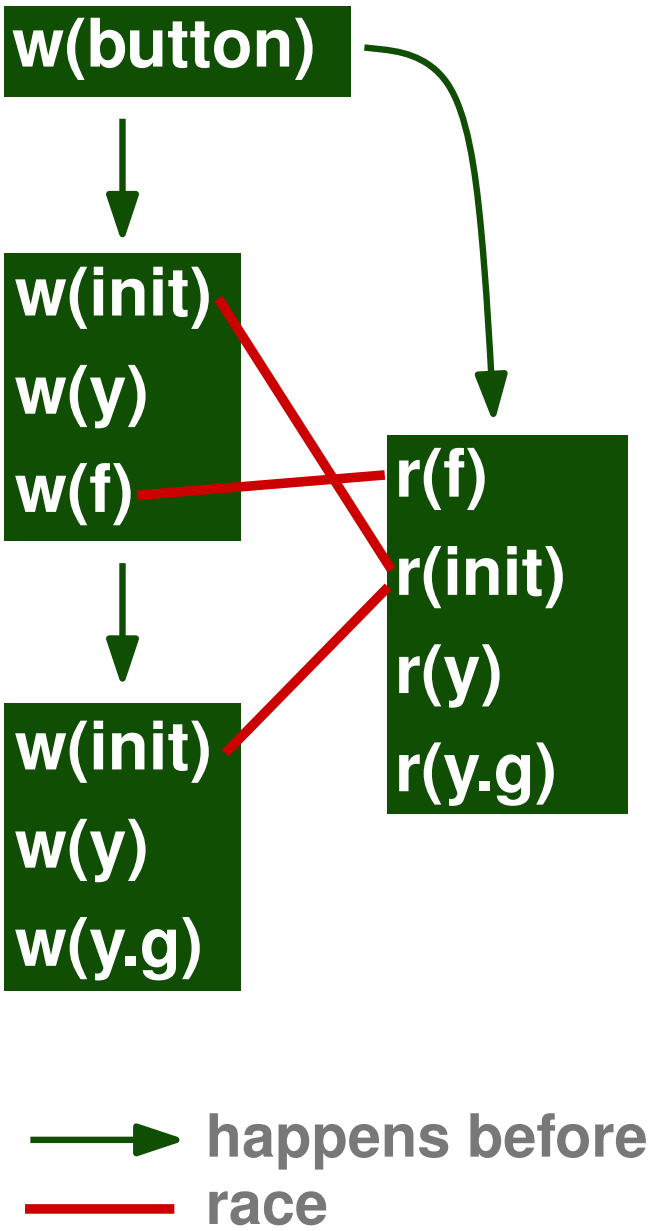
Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



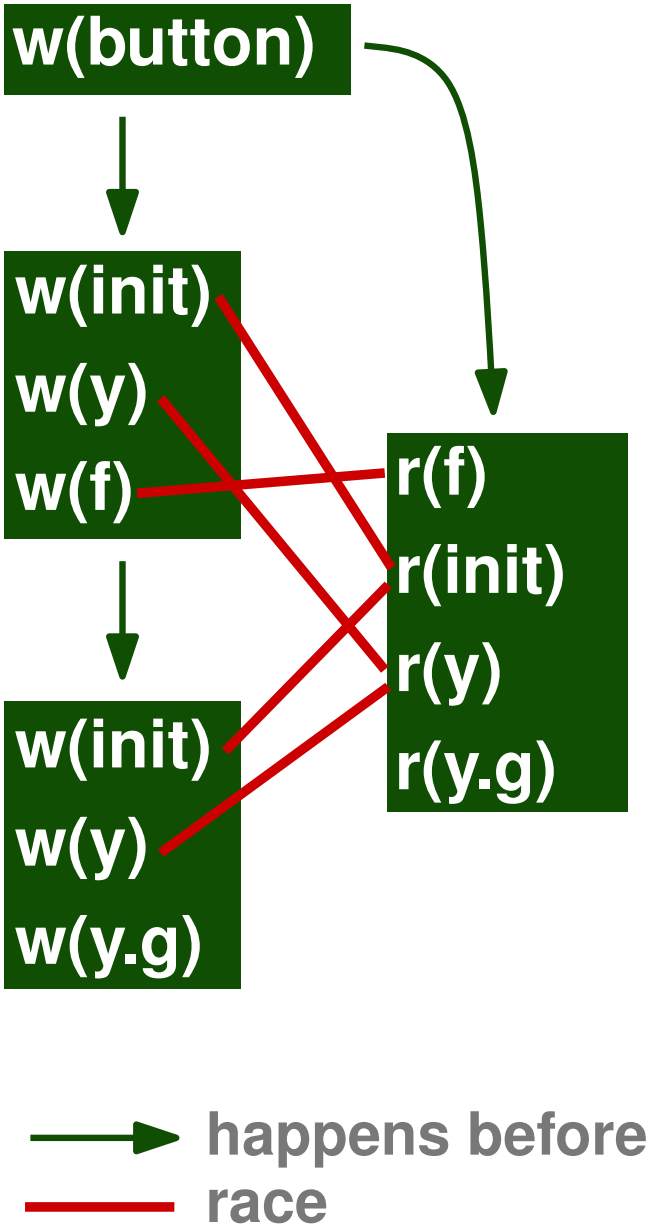
Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



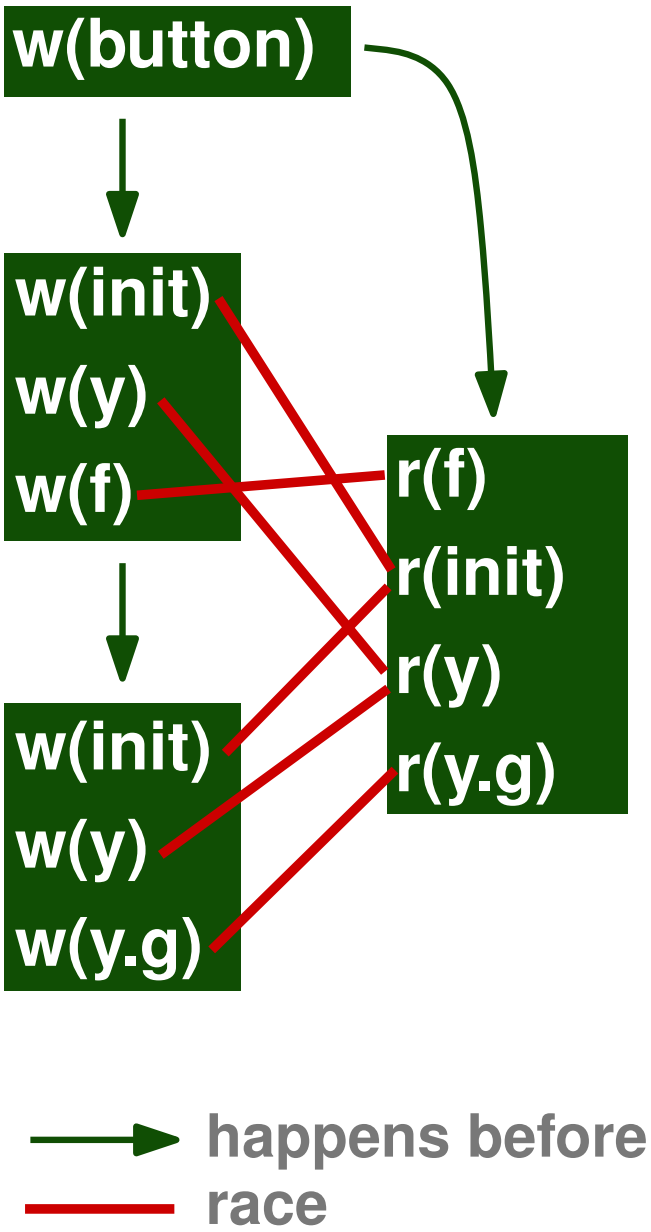
Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



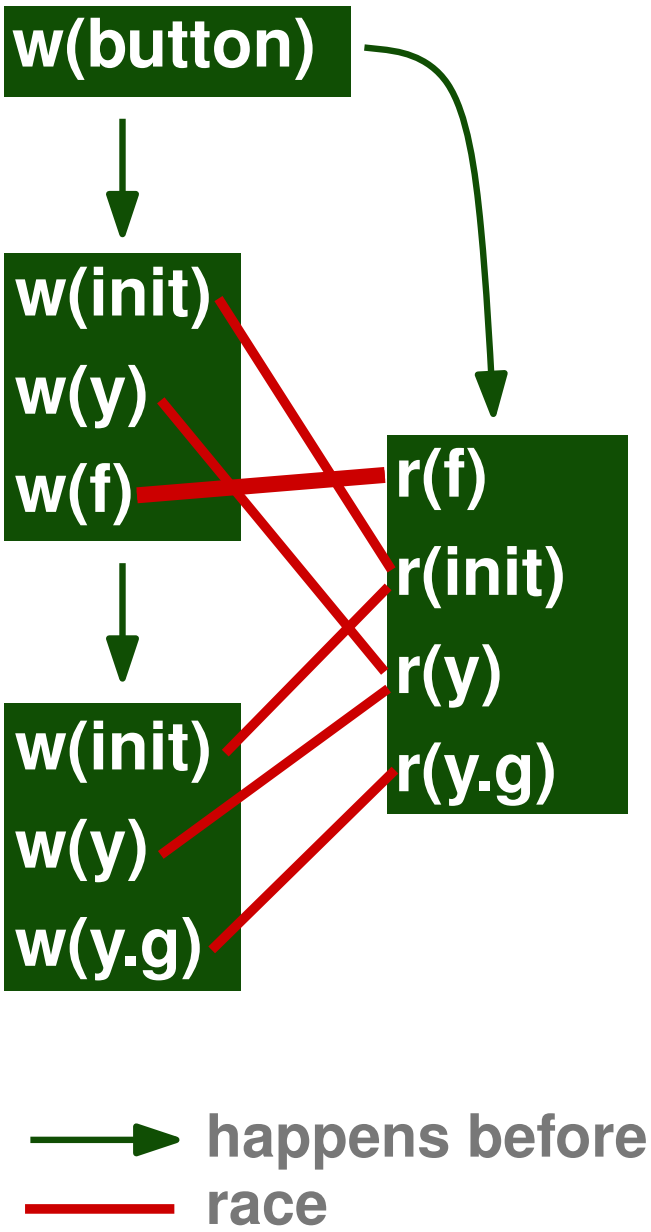
Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



Harmless Races

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```

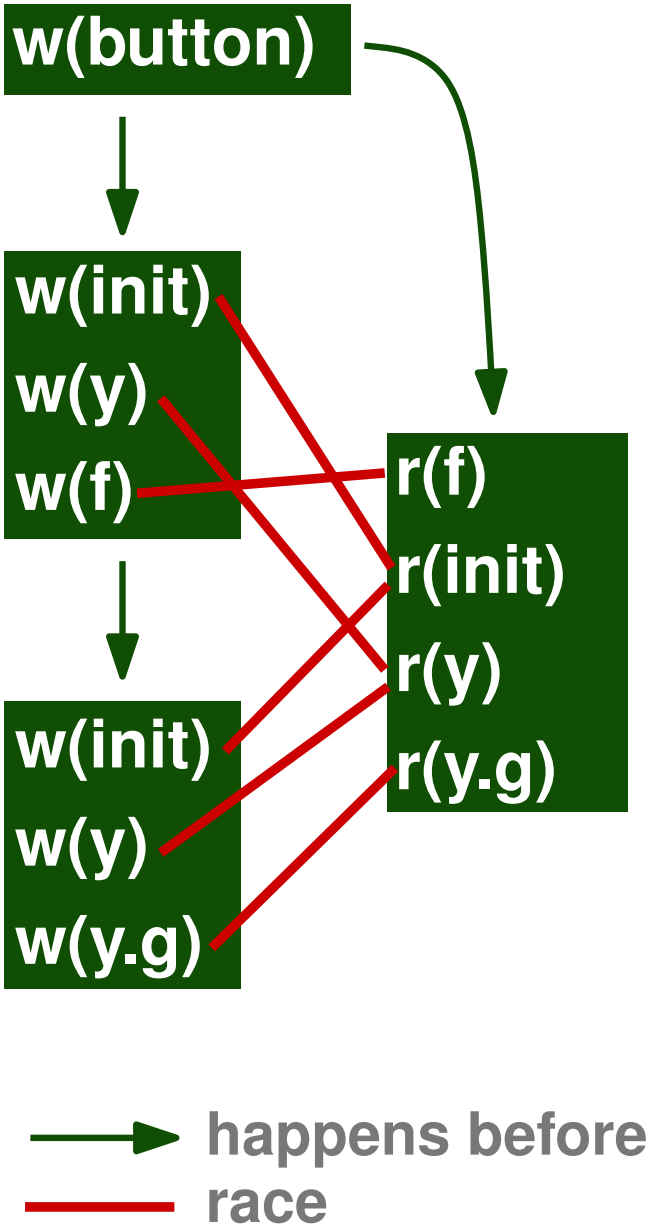


Race Coverage: Idea

- **Ignore a race that disappears when fixing another race**
- **Intuition: Other race is intended synchronization**

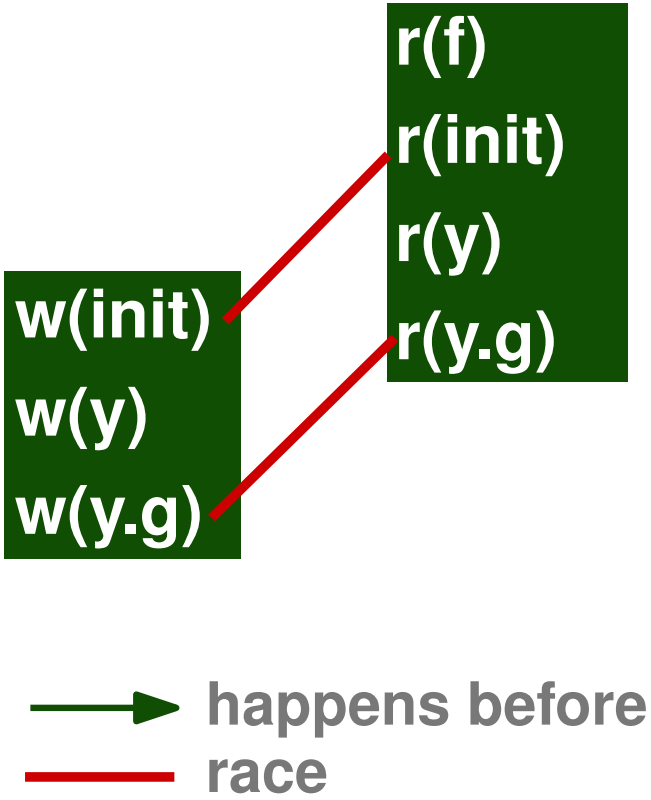
Race Coverage: Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



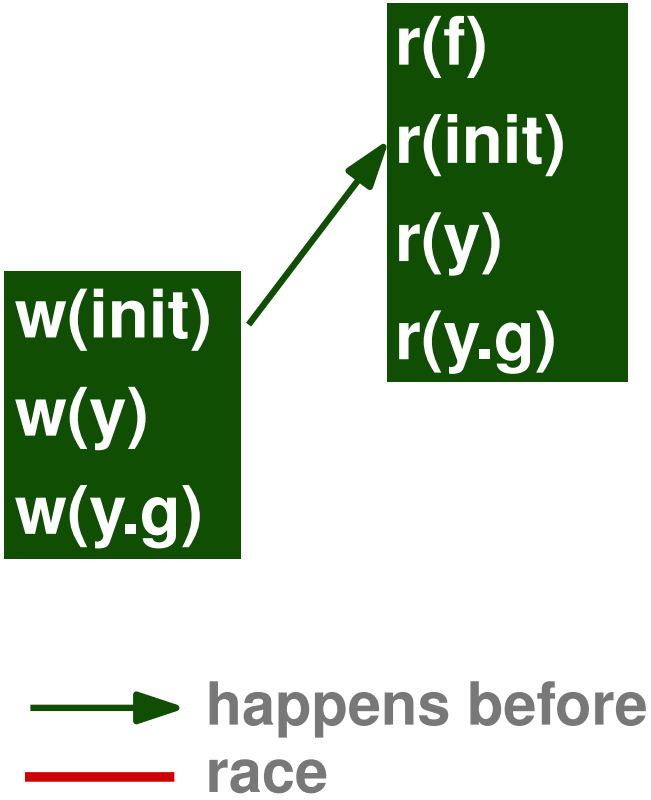
Race Coverage: Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



Race Coverage: Example

```
<html><body>
<input type="button" onclick="f()">
<script>
var init = false, y = null;
function f() {
  if (init) alert (y.g);
}
</script>
<script>
y = { g: 42 };
init = true;
</script>
</body></html>
```



Race Coverage: Definition

Race (a,b) is covered by race (c,d) iff

- **event(a) $\preceq$ event(c) and**
- **d $\preceq$ b**

$\prec$ **.. happens before**

Race Coverage: Definition

Race (a,b) is covered by race (c,d) iff

- **event(a) $\preceq$ event(c) and**
- **d $\preceq$ b**

$\prec$ **.. happens before**

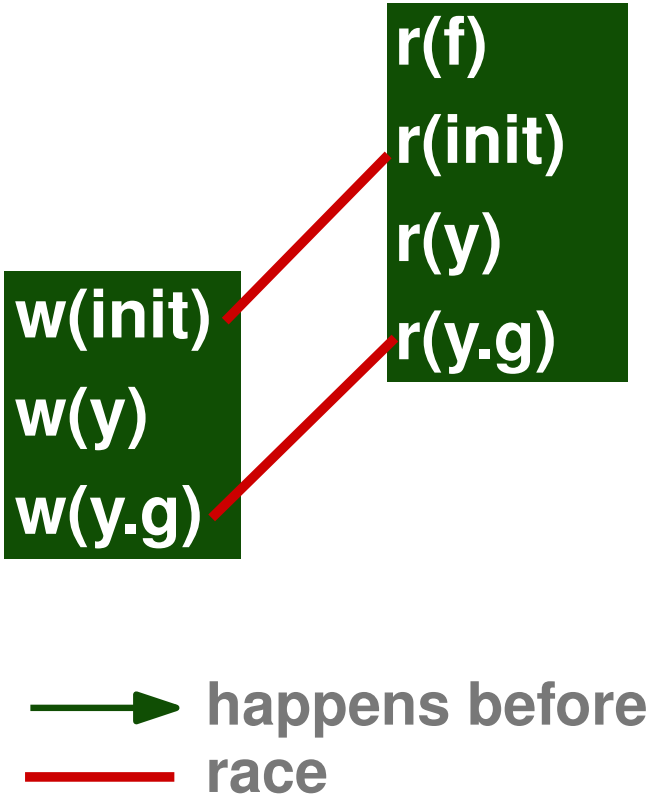
Report only races not covered by any other races

Race Coverage: Example

Race (a,b) is covered by race (c,d) iff

- **event(a) $\preceq$ event(c) and**
- **d $\preceq$ b**

$\prec$ **.. happens before**

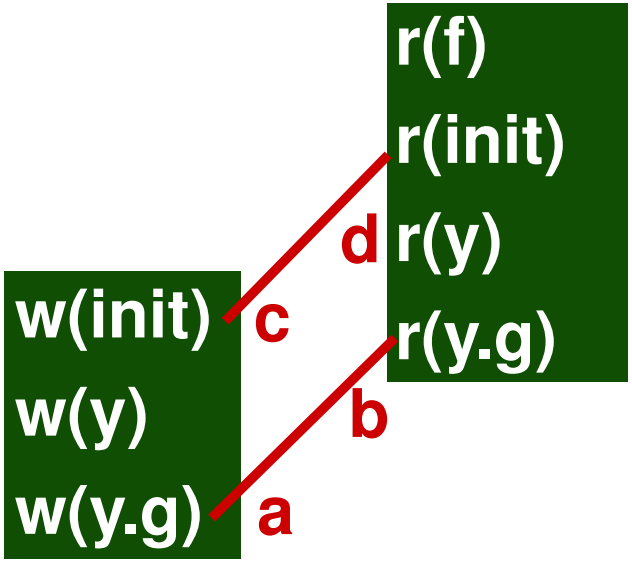


Race Coverage: Example

Race (a,b) is covered by race (c,d) iff

- **event(a) $\preceq$ event(c) and**
- **d $\preceq$ b**

$\prec$ **.. happens before**



Do not report race on y.g

happens before
race

Race Coverage: Limitations

May miss harmful races:

```
y = {g: 42};
init = true;
```

vs.

```
alert(init);
alert(y.g);
```

May still have false positives:

```
ctr++;
```

vs.

```
ctr++;
```

Race Detection: Challenges

Precision: Avoid false positives

- Race coverage ✓
- Heuristic classification

Scalability: Avoid vector clock blow-up

- Chain decomposition

Computing Uncovered Races

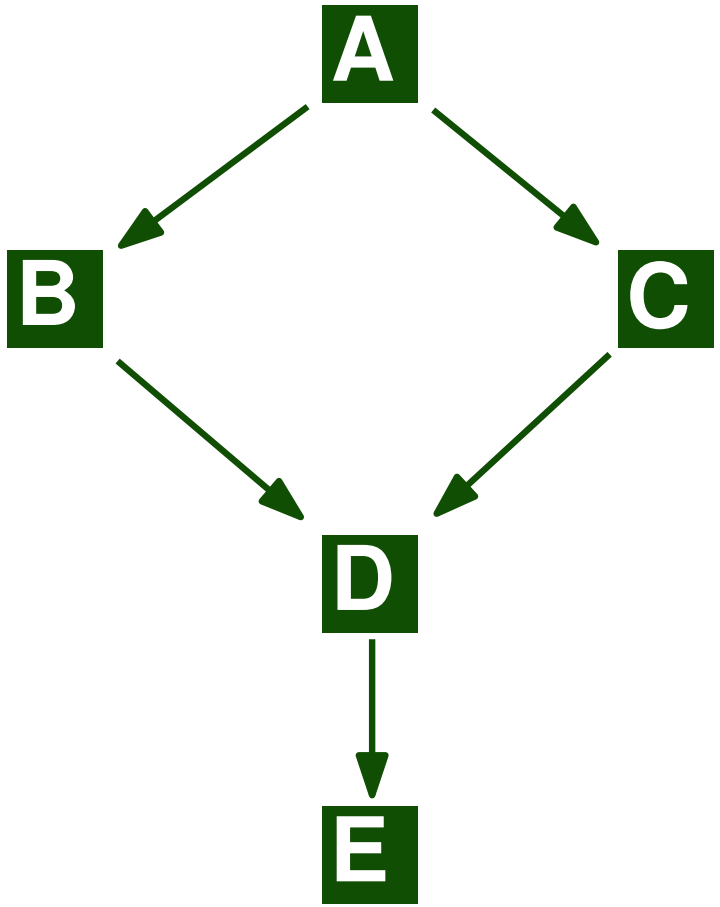
Input:

Sequence of event actions, partially ordered by happens-before relation

Challenge:

Make happens-before queries scalable

Happens Before Graph



$A \preceq B$

$A \preceq C$

$B \preceq D$

$C \preceq D$

$D \preceq E$

$A \preceq D$

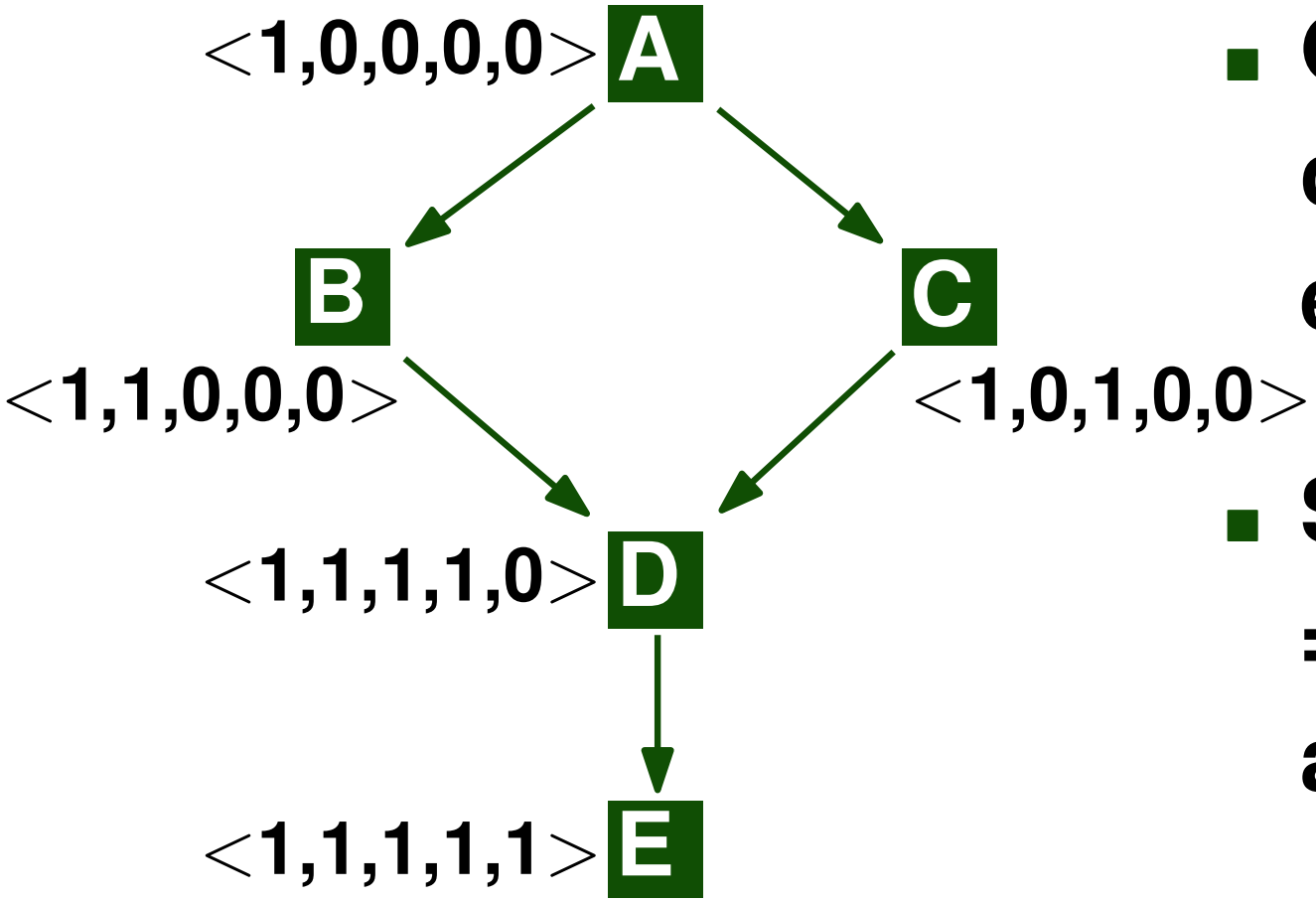
$A \preceq E$

$B \preceq E$

$C \preceq E$

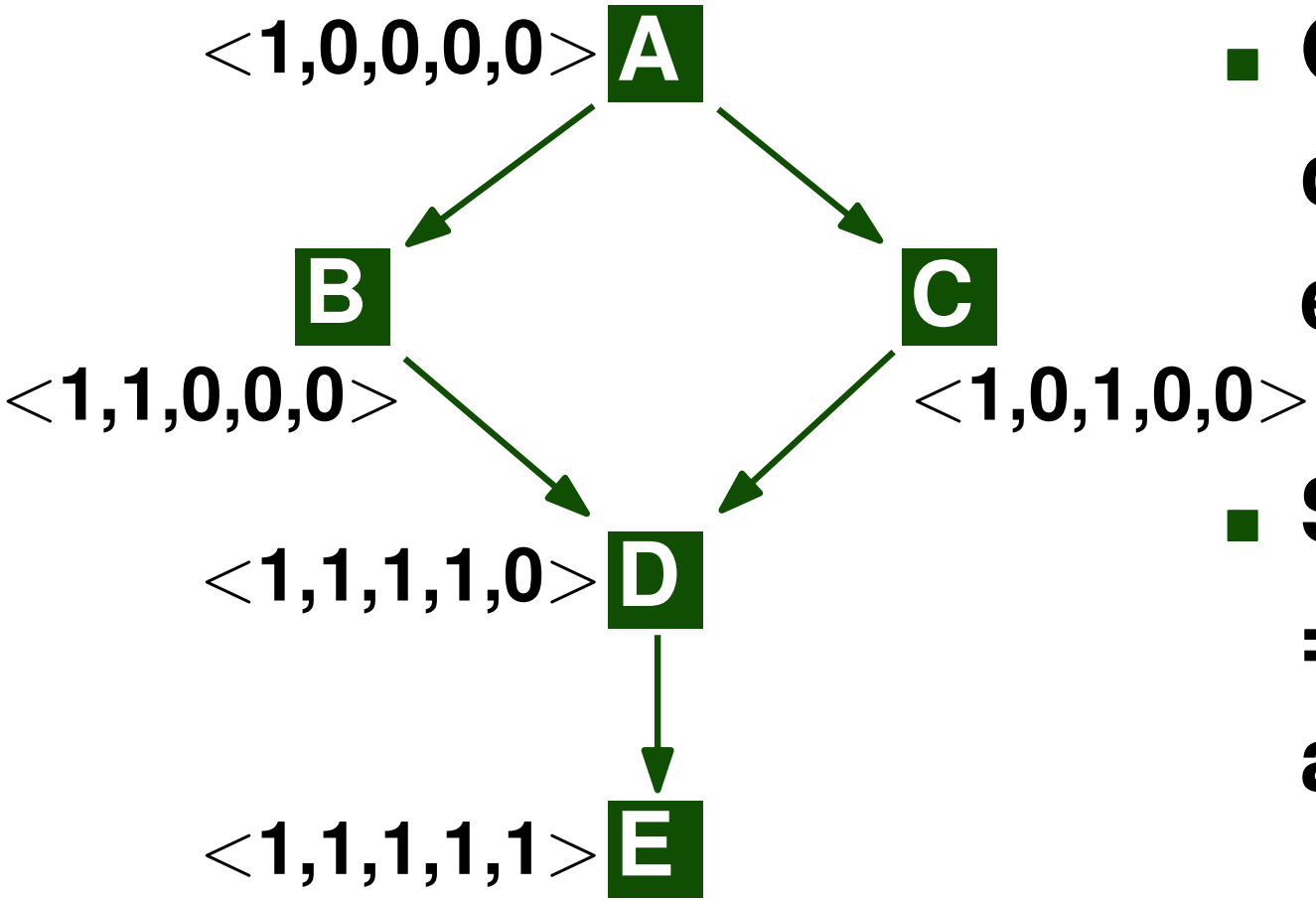
$B \not\preceq C$

Traditional: Vector Clocks



- One vector clock per event action
- Size of vector = # event actions

Traditional: Vector Clocks

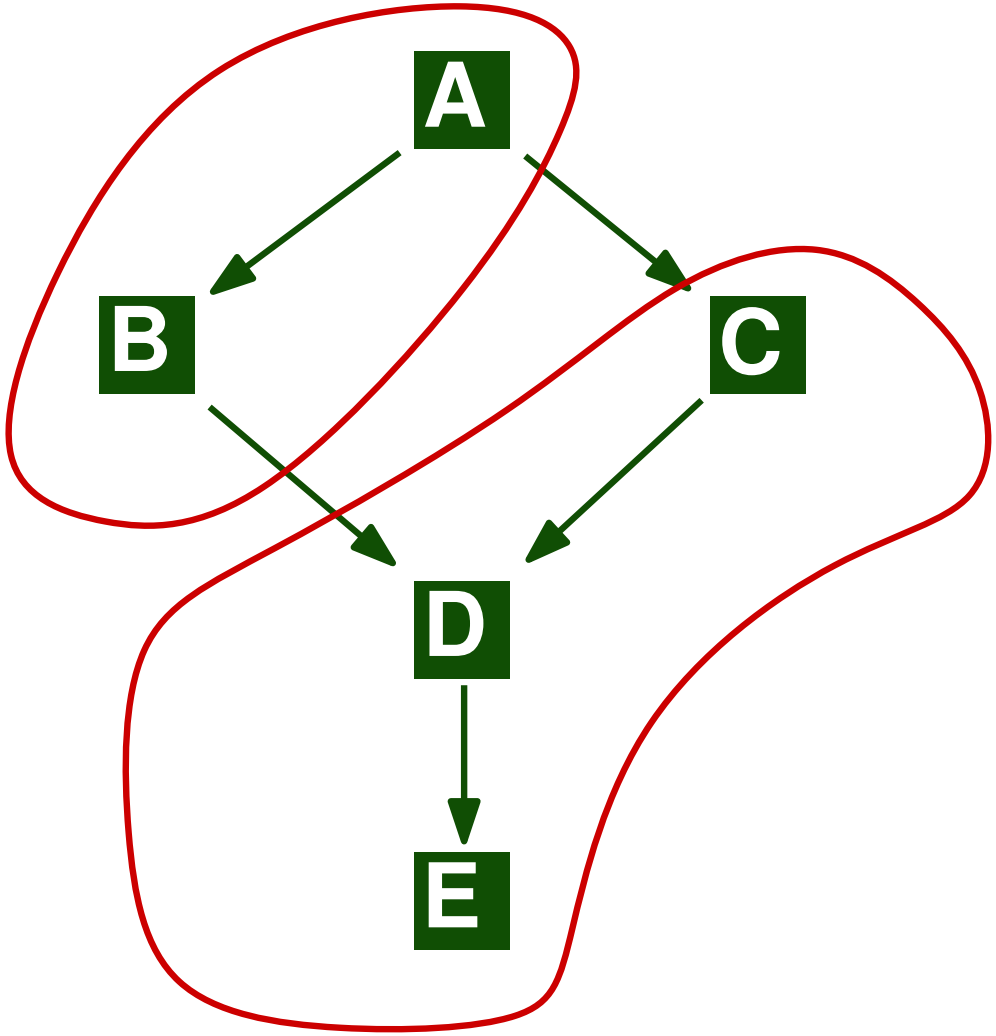


- One vector clock per event action
- Size of vector = # event actions

Needs O(A^2) space

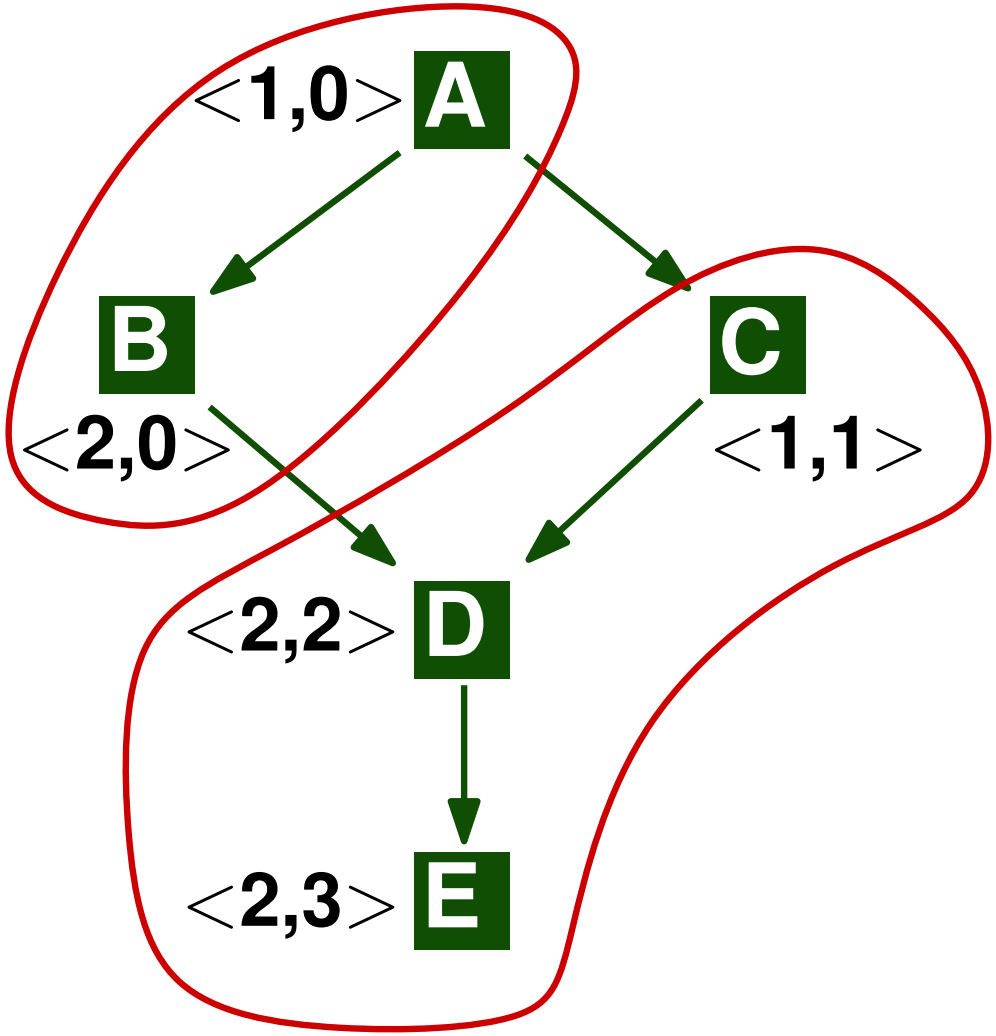
A .. # event actions

Chain Decomposition



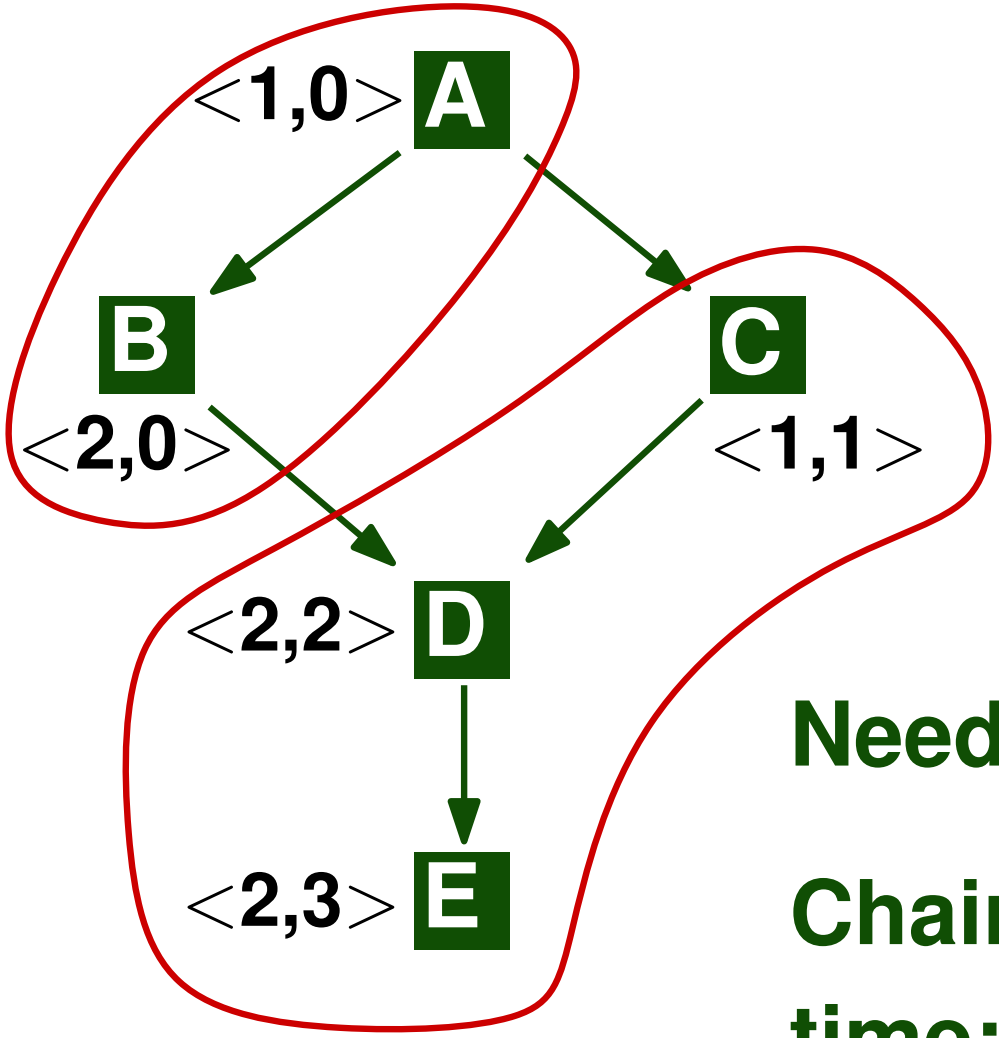
- Cover nodes with chains
- Size of vector = # chains

Chain Decomposition



- Cover nodes with chains
- Size of vector = # chains

Chain Decomposition



- Cover nodes with chains
- Size of vector = # chains

Needs O($C \cdot A$) space

Chain computation time: O($C \cdot H$)

A .. # event actions, C .. # chains, H .. # hb-edges

Race Detection: Challenges

Precision: Avoid false positives

- Race coverage ✓
- Heuristic classification

Scalability: Avoid vector clock blow-up

- Chain decomposition ✓

Evaluation

Implementation

- Modified WebKit browser engine
- Capture trace and analyse offline

Experiments

- Fortune 100 web sites
- Load + random events: 15 seconds

Results: Precision

Mean nb. of races per site

- **All races: 635**
- **Uncovered races: 45**
- **Heuristic filters: 18**

14x improvement

Results: Precision (2)

Manual classification of 305 races:

- **178 synchr. variables**
- **52 harmless**
- **75 harmful**

Results: Precision (2)

Manual classification of 305 races:

- **178 synchr. variables**
- **52 harmless** → **Claim: 17% false positives**
- **75 harmful**

Results: Precision (2)

Manual classification of 305 races:

- **178 synchr. variables** → **Could also say: 75% false positives**
- **52 harmless** →
- **75 harmful**

Overview

1. **Type Inference for JIT Compilation**
2. **Race Detection in Web Applications** ✔

Mostly based on these papers:

- *Fast and Precise Hybrid Type Inference for JavaScript*, Hackett and Guo, PLDI 2012
- *Race Detection for Web Applications*, Petrov et al., PLDI 2012
- *Effective Race Detection for Event-Driven Programs*, Raychev et al., OOPSLA 2013