Fearless concurrency

or how can the compiler save us?

A paper presentation about uniqueness and reference immutability

Thomas Lacroix
KTH – Pardis 18

25 09 2018

Main reference

Colin S. Gordon, Matthew Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy.
Uniqueness and reference immutability for safe parallelism.
Technical report, October 2012.

Wanna race with me?

Problem

We work for the social care. We would like to sort all the ages of Sweden citizens. Can you do that for us?

Ok, so there are about 10M people in Sweden.

$O(\log(N))$	20
$O(N)$	10,000,000
$O(N \cdot \log(N))$	200,000,000
$O(N^2)$	100,000,000,000,000

Wanna race with me?

```
1  class LinearAgeSort implements SortAlgorithm {
2     final int MIN_AGE = 0;
3     final int MAX_AGE = 300;
4     @Override
5     public void sort(int[] list) {
6        int[] age_count = new int[MAX_AGE-MIN_AGE+1];
7        for (int age : list) {
8           age_count[age]++;
9        }
10       int k = 0;
11       for (int i = 0; i < age_count.length; i++) {
12          for (int j = 0; j < age_count[i]; j++) {
13             list[k++] = i;
14          }
15 }  }  }
```

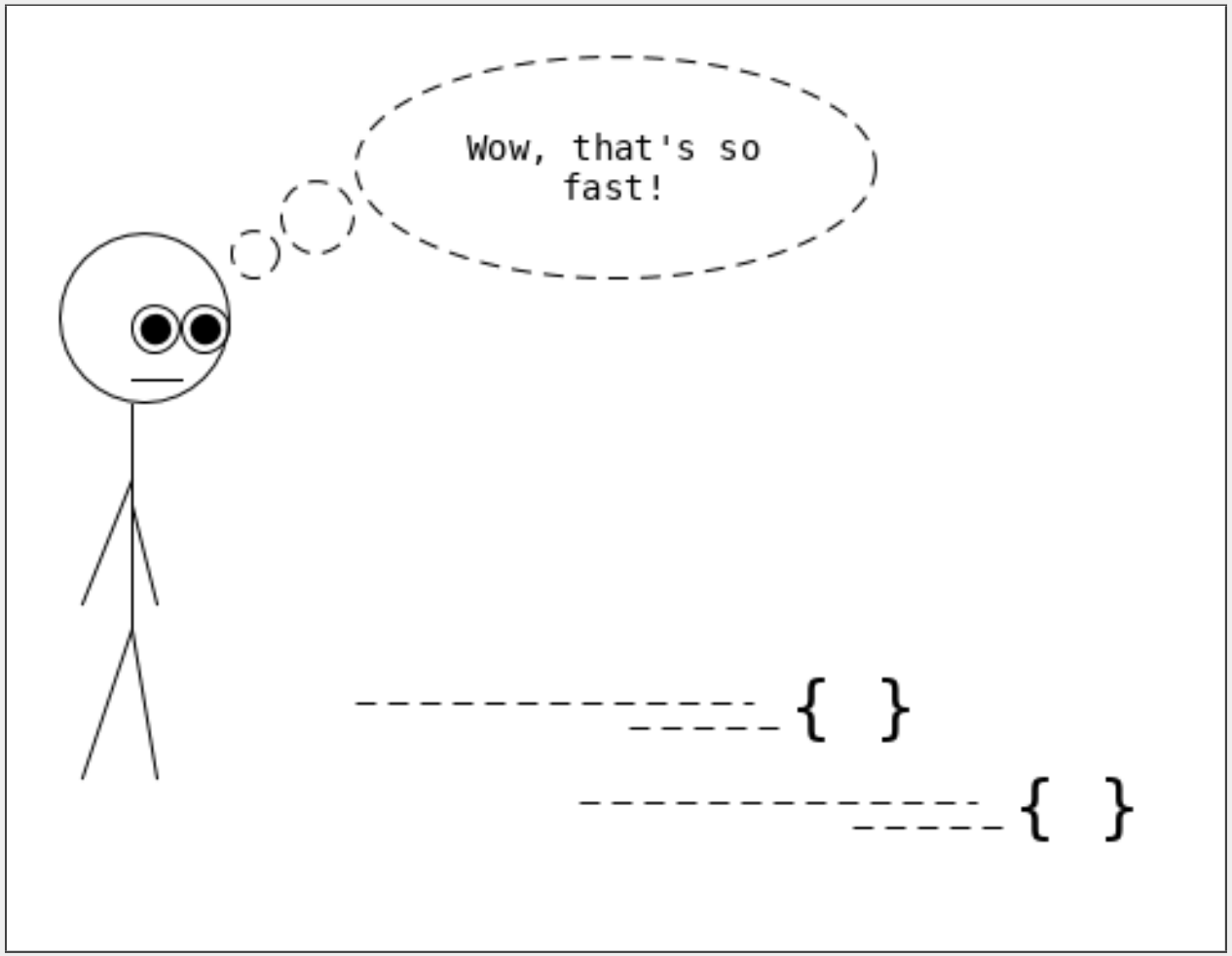
Wanna race with me?

```
{ 4, 10, 5, 2, 10, 4, 6, 7, 8, 7, 8, 4, 10, 11,
  4, 5, 7, 8, 4, 10, 11, 8, 14 }
```

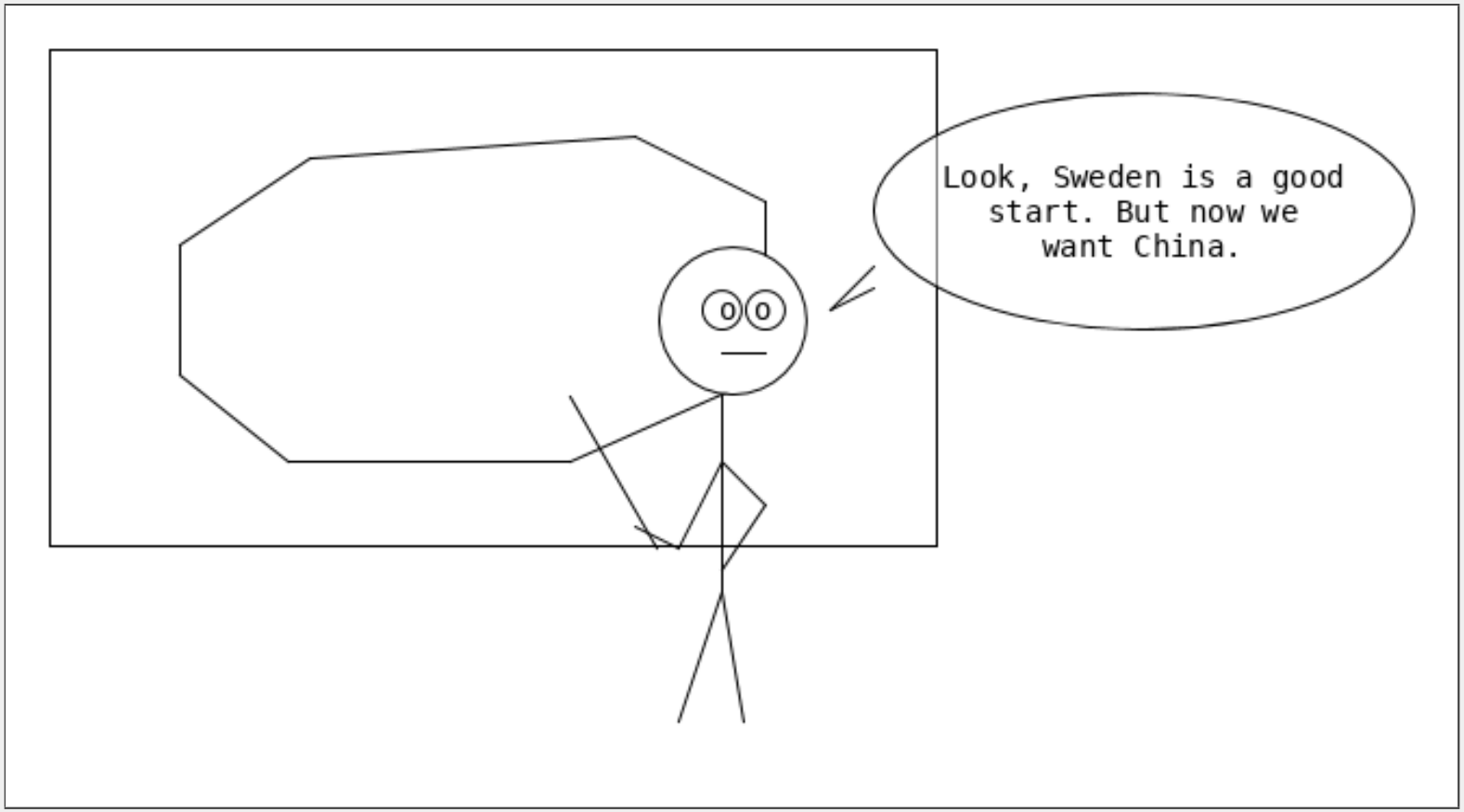
Age	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Count	0	0	1	0	5	2	1	3	4	0	4	2	0	0	1	0

Wanna race with me?

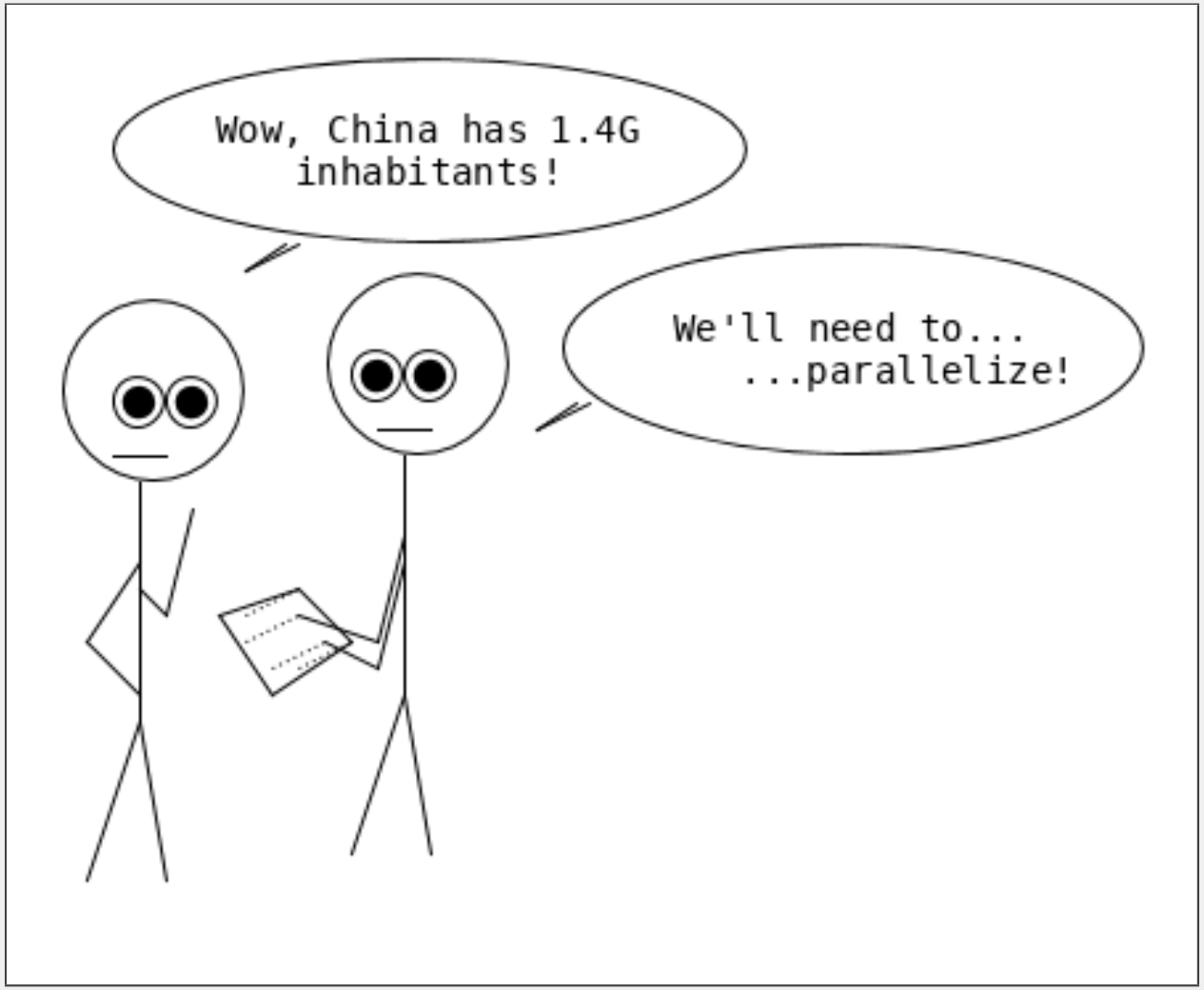
Well done!



Wanna race with me?



Wanna race with me?



Wanna race with me?

No problem, we can just split the list and have multiple threads go though all the data!

```
1  class ParallelLinearAgeSort implements SortAlgorithm {
2    public void sort(int[] list) {
3      int[] age_count = new int[MAX_AGE-MIN_AGE+1];
4      Thread[] ts = new Thread[NUM_THREADS];
5      for (int t = 0; t < ts.length; t++) {
6        int begin_i = t;
7        ts[t] = new Thread(() -> {
8          for (int i = begin_i; i < ts.length; i += NUM_THREADS)
9            age_count[list[i]]++;
10       });
11       ts[t].start();
12     }
13     for (Thread t : ts) t.join();
14     // ...
15   }
16 }
```

Wanna race with me?

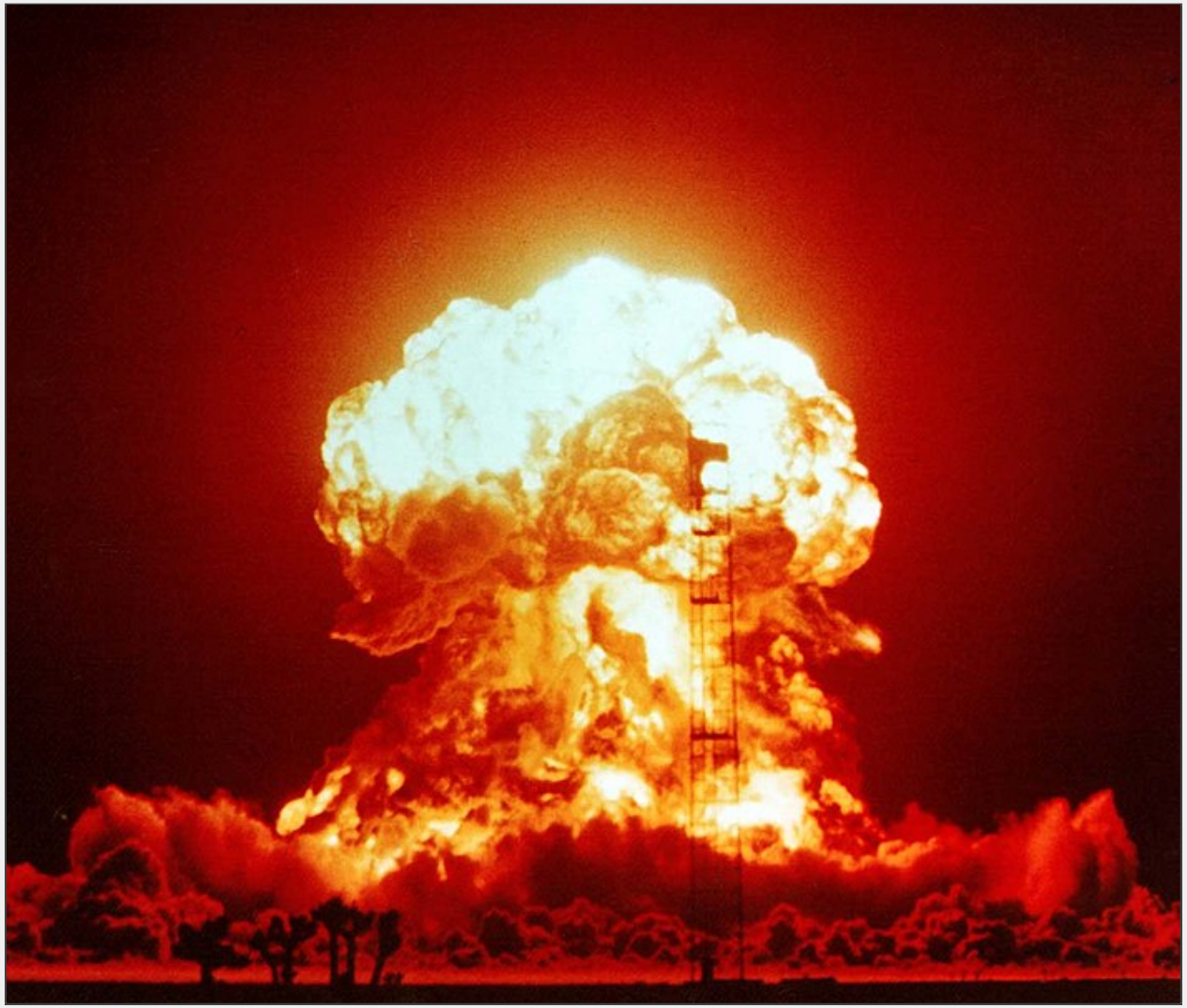


Photo courtesy of National Nuclear Security Administration / Nevada Site Office [Public domain], via Wikimedia Commons

Wanna race with me?

So, what happened?

$\Rightarrow$ There is a data race.

Wanna race with me?

What is a data race?

Definition

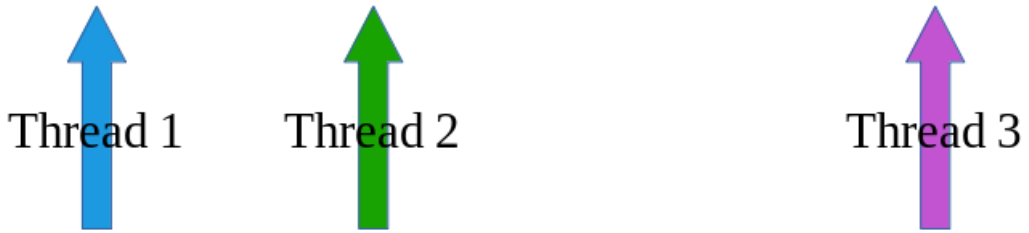
Data races are defined as:

- two or more threads concurrently accessing a location of memory,
- one of them is a write,
- one of them is unsynchronized.

From: Rustonomicon > Races

Wanna race with me?

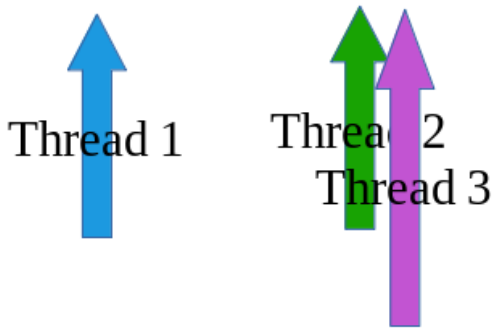
Age	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Count	0	0	1	0	5	2	1	3	4	0	4	2	0	0	1	0



So far so good, the threads operate on distinct cells.

Wanna race with me?

Age	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Count	0	0	1	0	5	2	1	3	4	0	4	2	0	0	1	0



Oups, they work on the same memory location.

Wanna race with me?

Could we have seen it right away?
Yes.

Why?
$\Rightarrow$ Threads work on the same memory location.

But that's basic stuff, right?
$\Rightarrow$ So, why does that even compile?

Wanna race with me?

Is racy code desirable? Yes / No.

Why? Because it's Undefined Behaviour.
$\Rightarrow$ A compiler can **forbid** UB.

Compile-time race-freedom guarantee

Compile-time race-freedom guarantee

Our problem

How can a compiler detect data races?

Definition

Data races are defined as:

- two or more threads concurrently accessing the same memory location,
- one of them is a write,
- one of them is unsynchronized.

From: Rustonomicon > Races

Compile-time race-freedom guarantee

```
1  class ParallelLinearAgeSort implements SortAlgorithm {
2    public void sort(int[] list) {
3      int[] age_count = new int[MAX_AGE-MIN_AGE+1];
4      Thread[] ts = new Thread[NUM_THREADS];
5      for (int t = 0; t < ts.length; t++) {
6        int begin_i = t;
7        ts[t] = new Thread(() -> {
8          for (int i = begin_i; i < ts.length; i += NUM_THREADS)
9            age_count[list[i]]++;
10       });
11       ts[t].start();
12     }
13     for (Thread t : ts) t.join();
14     // ...
15   }
16 }
```

- age_count is accessed by all threads

Compile-time race-freedom guarantee

- `age_count` is accessed by all threads

In other words, two threads share a writeable reference on the same memory location.

Compile-time race-freedom guarantee

The paper presents a **type system** to **restrict updates** to memory to prevent (some) race conditions.

They provide a novel combination of **immutable** and **unique** (isolated) types that ensure **safe parallelism** (race freedom and deterministic execution).

Compile-time race-freedom guarantee

Reference immutability is based on a set of permission-qualified types. The system has four qualifiers:

- **writable:** An "ordinary" object reference, which allows mutation of its referent.
- **readable:** A read-only reference, which allows no mutation of its referent. Furthermore, no heap traversal through a read-only reference produces a `writable` reference (`writable` references to the same objects may exist and be reachable elsewhere, just not through a `readable` reference). A readable reference may also refer to an immutable object.

Compile-time race-freedom guarantee

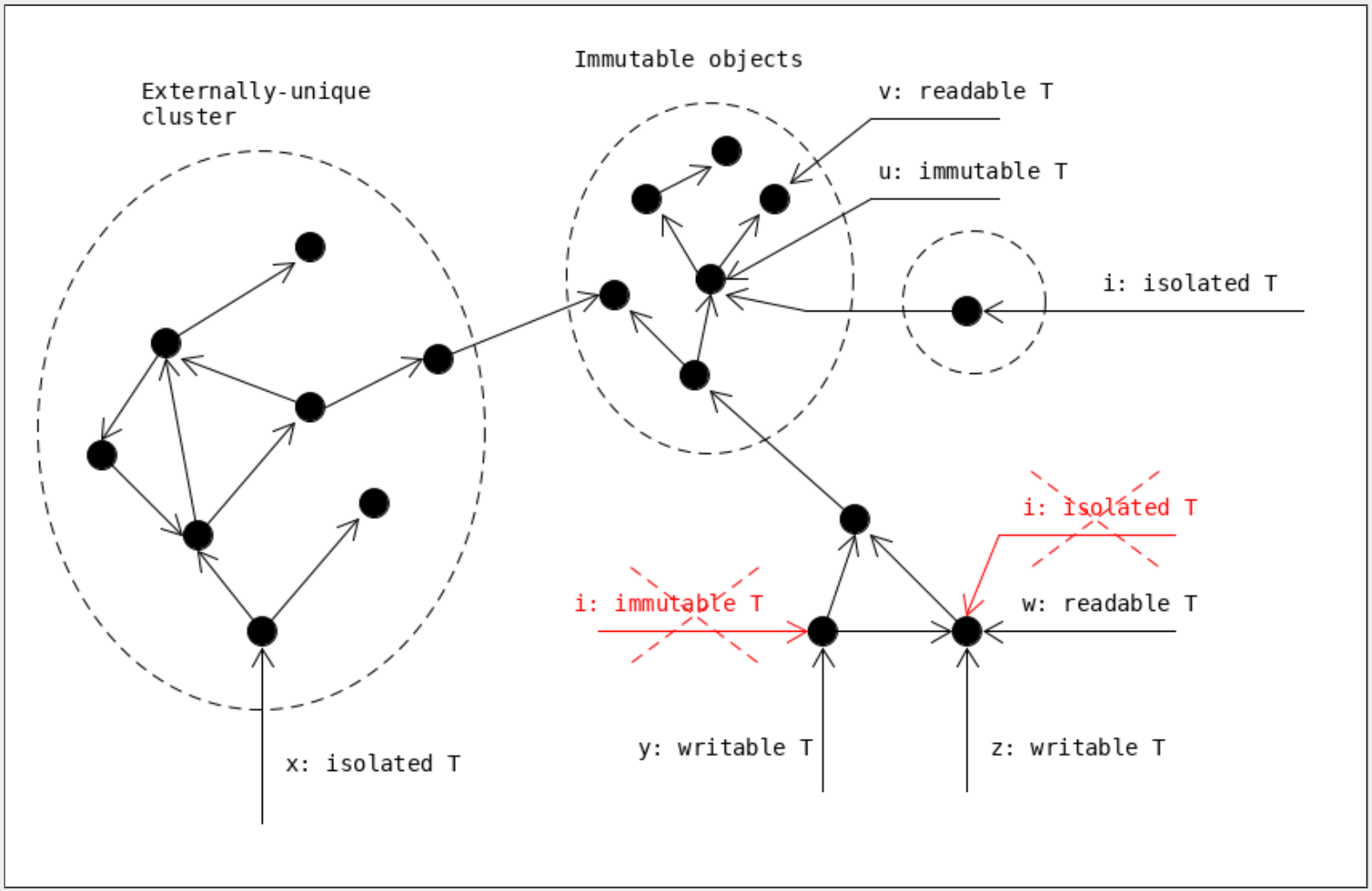
- **immutable:** A read-only reference which additionally notes that its referent can never be mutated through any reference. Immutable references may be aliased by `readable` or `immutable` references, but no other kind of reference. All objects reachable from an `immutable` reference are also `immutable`.

Compile-time race-freedom guarantee

- **isolated:** An external reference to an externally- unique object cluster. External uniqueness naturally captures thread locality of data. An externally-unique aggregate is a cluster of objects that freely reference each other, but for which only one external reference into the aggregate exists. All paths to non-immutable objects reachable from the isolated reference pass through the isolated reference.

`isolated` references must be converted through subtyping to another permission before use.

Compile-time race-freedom guarantee



Compile-time race-freedom guarantee

The objective now is to assemble a set of rules to:

- control where modifications can occur
- guarantee that modifications cannot occur in some situations

Compile-time race-freedom guarantee

For example, a developer can be sure that a library call to a static method with the type signature

```
1     int countElements(readable ElementList lst);
```

will not modify the list or its elements (through the `lst` reference). Accessing any field of the argument `lst` through the `readable` reference passed will produce other `readable` (or `immutable`) references.

Compile-time race-freedom guarantee

For example, a developer could not implement `countElements()` like so:

```
1     int countElements(readable ElementList lst)
2     { lst.head = null; return 0; }
```

because the compiler would issue a **type error**. In fact, **any attempt** within `countElements()` **to modify** the list would result in a **type error**, because `lst` is **deeply (transitively) read-only**, and writes through read-only references are prohibited.

This can remind you of the `const` *modifier in C++.*

Compile-time race-freedom guarantee

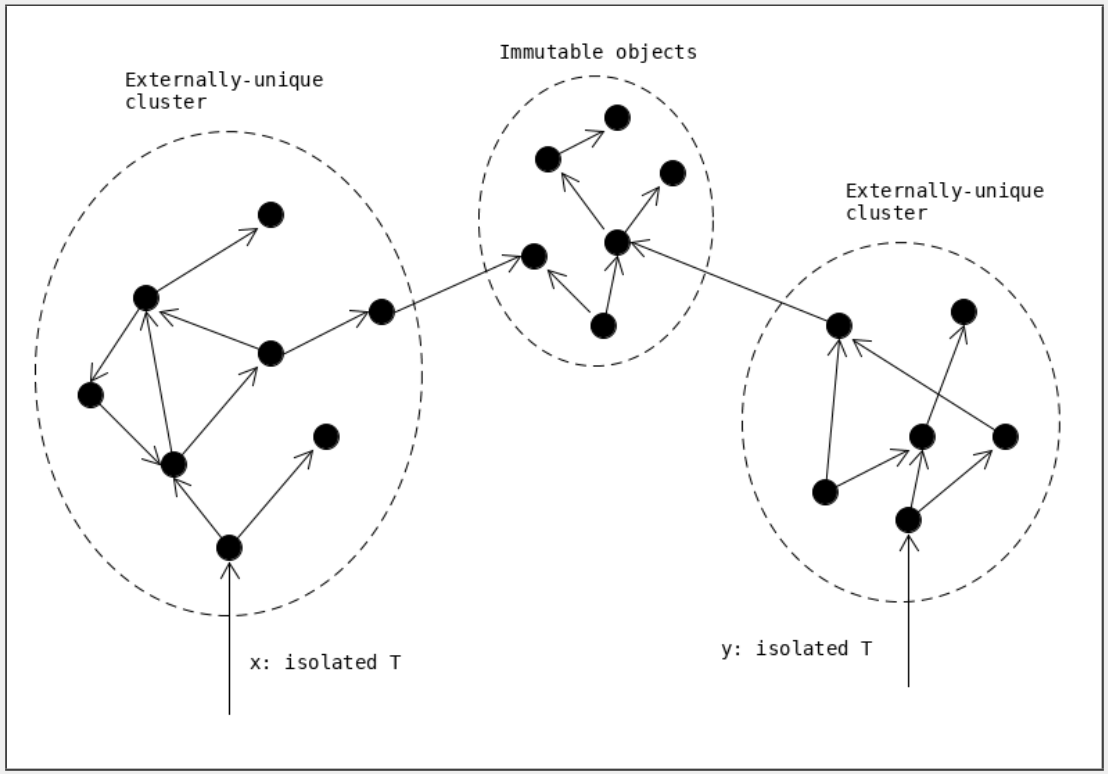
The `isolated` permission is a novelty of this work, and is particularly important for two reasons.

First, they support natural safe parallelism. Here, threads cannot interfere with each other, because they work on distinct memory locations.

```
1     isolated IntList l1 = ...;
2     isolated IntList l2 = ...;
3     { l1.map(new Incrementor()); }
4       || { l2.map(new Incrementor()); }
```

Compile-time race-freedom guarantee

Second, as the whole object graph behind an isolated reference is only accessible though an **unique** reference, it can be converted to other permissions, like `writable`, at once.



Compile-time race-freedom guarantee

The paper provides a set of subtyping rules (remember, it's a type system). Here are a few:

Compile-time race-freedom guarantee

- it is impossible to acquire a `writable` reference to an `immutable` type.

```
1    immutable IntList l = generateRange(0, 100);
2
3    // default permission is writable
4    IntList mutRef = l; // <- this is forbidden, type error!
5
6    mutRef.set(2, -1);
```

Compile-time race-freedom guarantee

- `isolated` references must be converted before any use, and such a conversion is **destructive**.

```
1  isolated IntList l = ...;
2
3  // update l's permission to writable
4  writable IntList l2 = l;
5
6  print(l2.get(3)); // ok
7
8  l.head = ...; // Type Error!
```

Compile-time race-freedom guarantee

- `immutable` and `isolated` references can be recovered under conditions.

```
1    isolated IntBox increment(isolated IntBox b) {
2      // implicitly convert b to writable
3      b.value++;
4
5      // convert b *back* to isolated
6      return b;
7    }
```

Is it safe? **Yes**: there is only one `writable` reference to `b`.

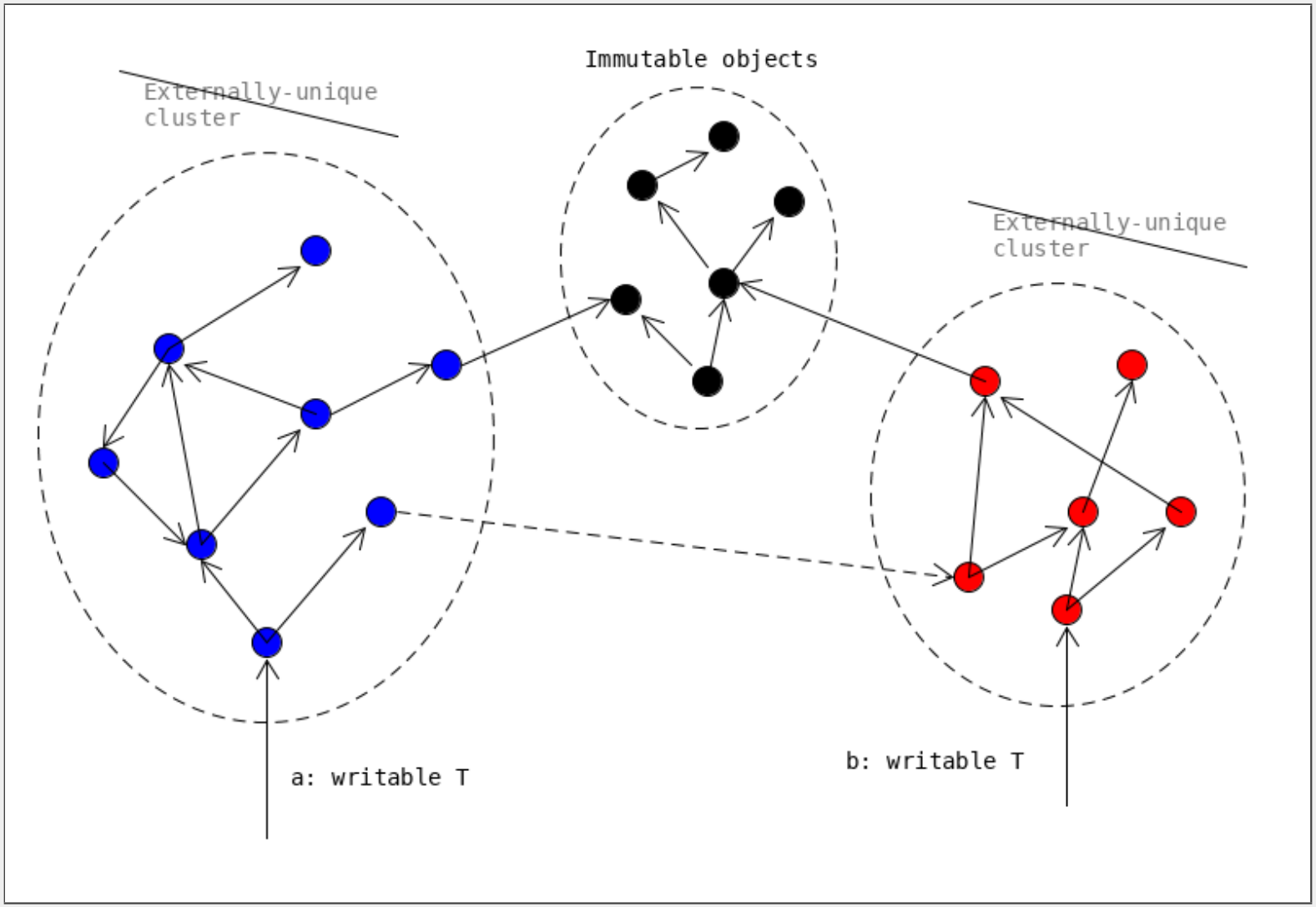
Note: The language doesn't allow mutable global variables.

Compile-time race-freedom guarantee

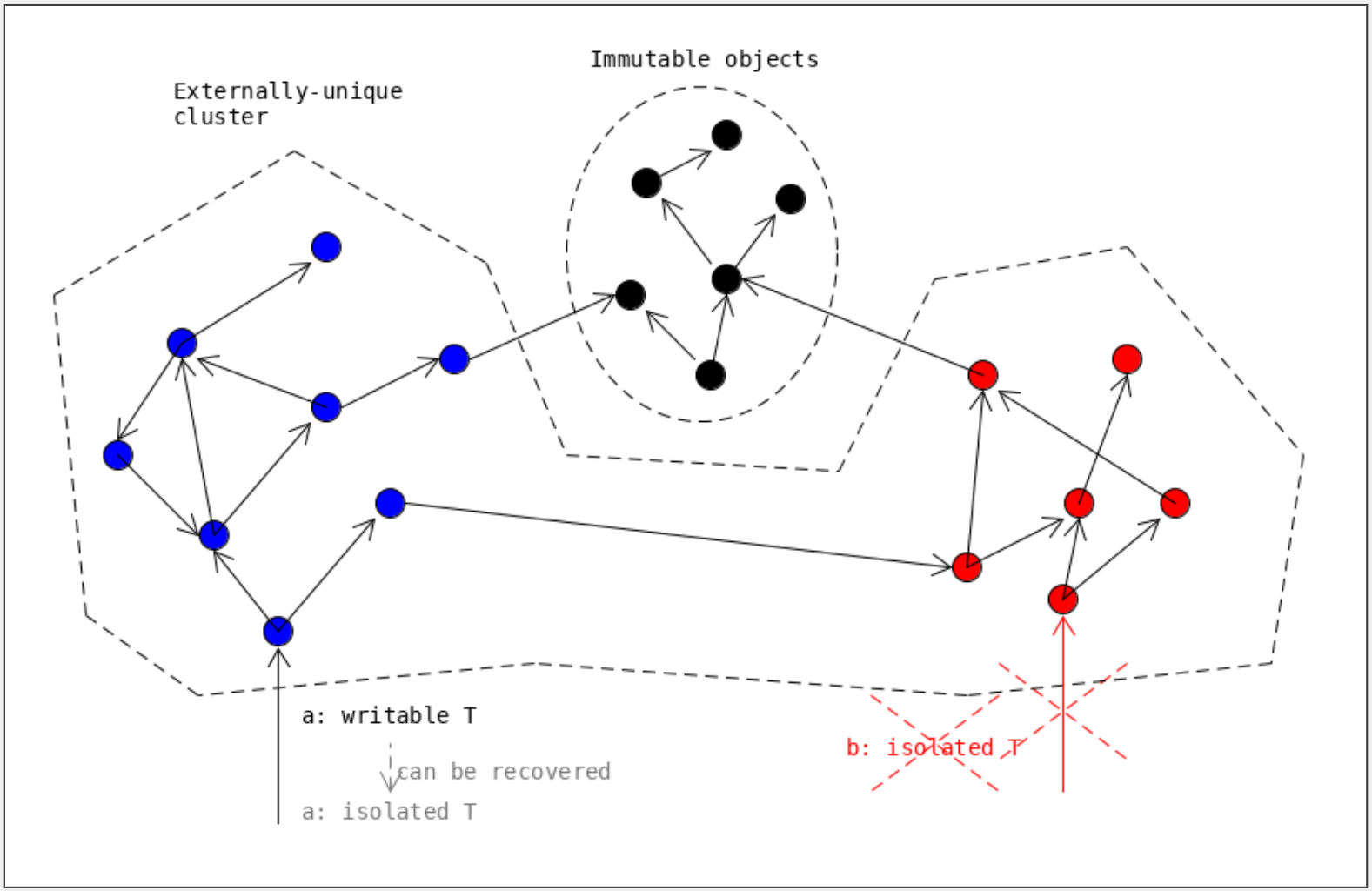
```
1     isolated Bar foo(isolated Bar a, isolated Bar b) {
2       a.baz = b;
3       return b;
4     }
```

What about this?

Compile-time race-freedom guarantee



Compile-time race-freedom guarantee

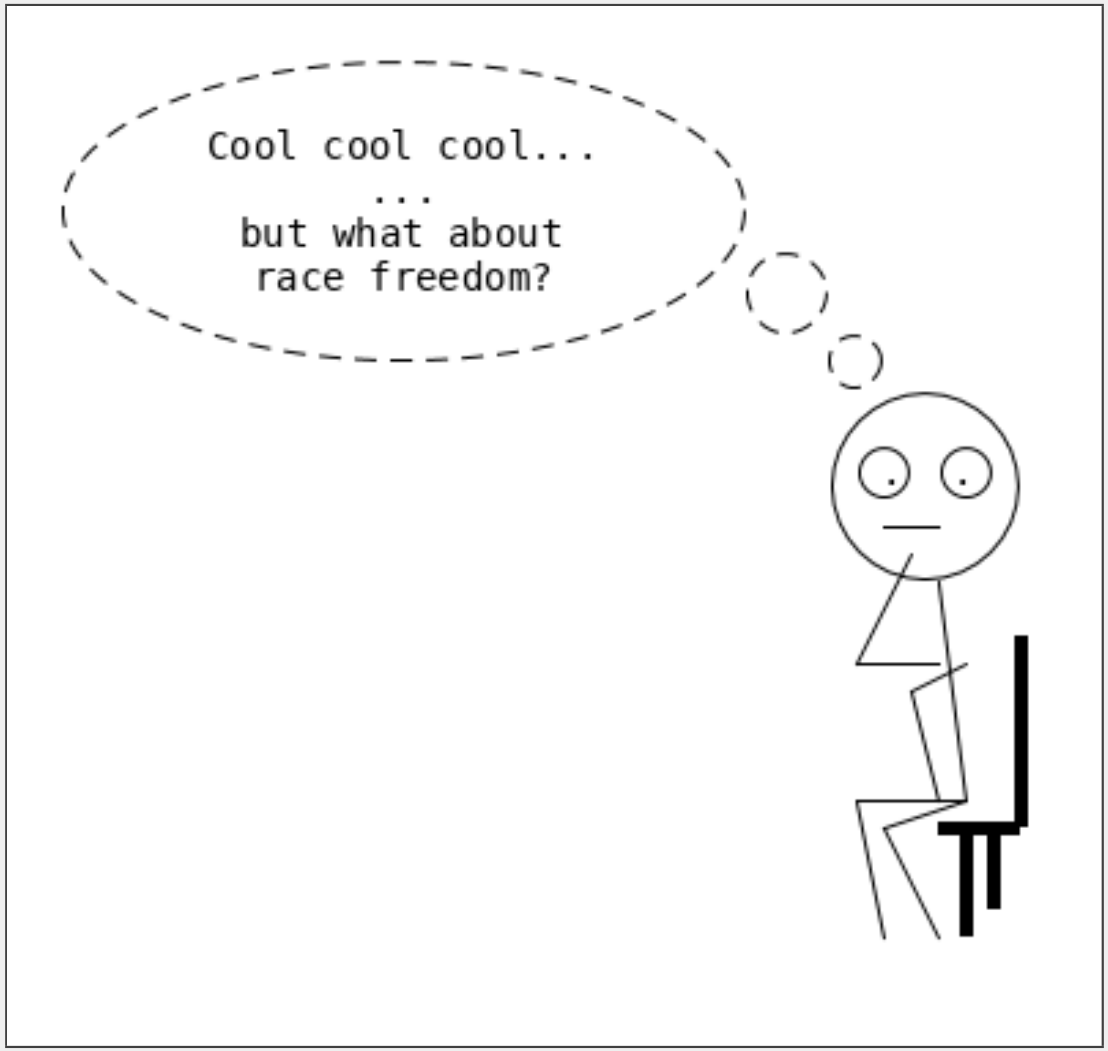


Compile-time race-freedom guarantee

The call to `foo( )`:

```
1       isolated IntBox a = ...;
2       isolated IntBox b = ...;
3
4       // here, a and b are consumed
5       isolated IntBox r = foo(a, b);
6
7       a.x; // Type Error, a has been consumed
8
9       // only r is valid now
```

Compile-time race-freedom guarantee

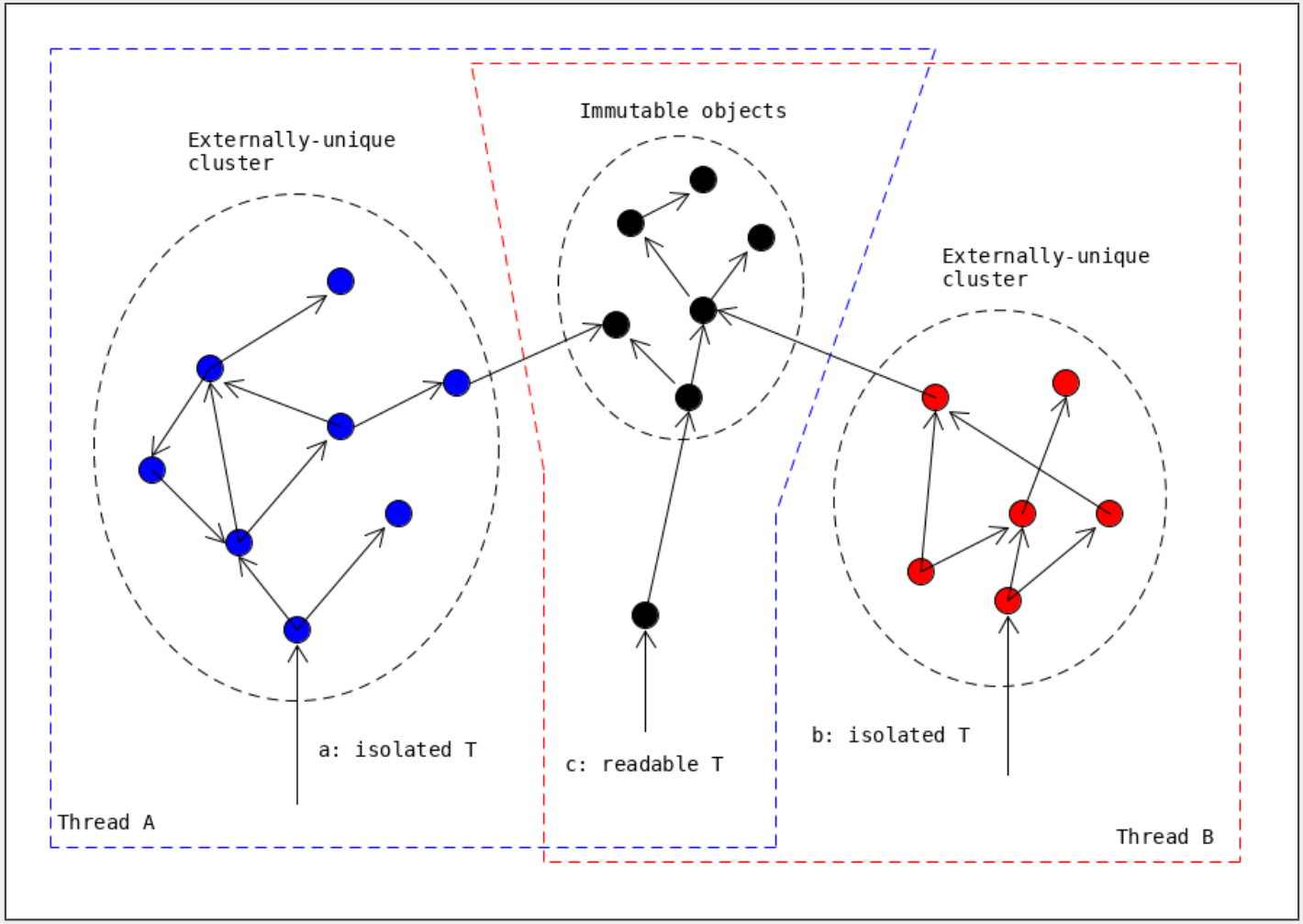


Compile-time race-freedom guarantee

The paper presents two forms of parallelism:

- **Symmetric:** Assuming that at most one thread may hold `writable` references to an object at a given point in time, then while all `writable` references in a context are temporarily forgotten, it becomes safe to share all read-only or immutable references among multiple threads, in addition to partitioning externally-unique clusters between threads.
- **Asymmetric:** If all data accessible to a new thread is immutable or from externally-unique clusters which are made inaccessible to the spawning thread, then the new and old threads may run in parallel without interference.

Compile-time race-freedom guarantee

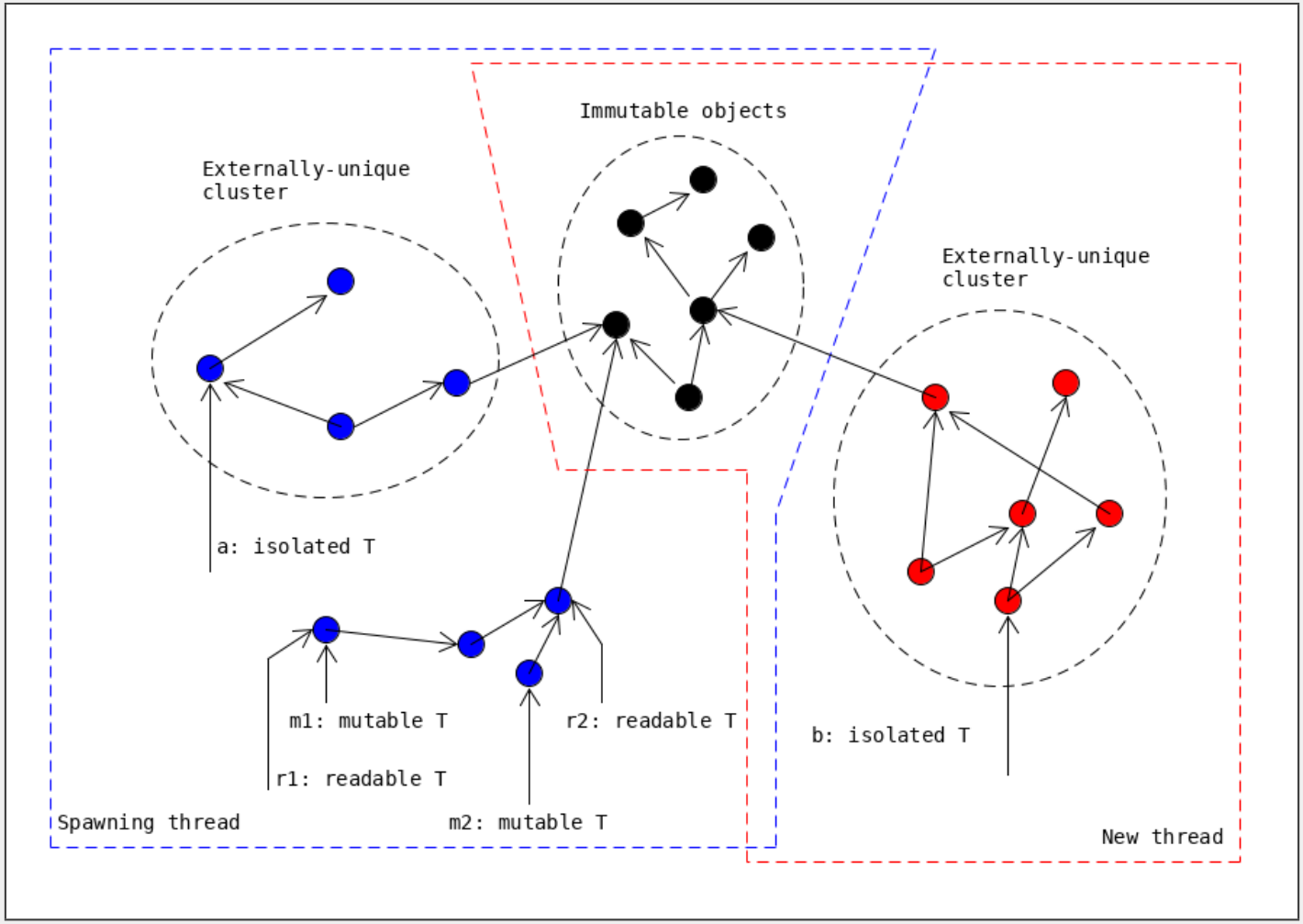


Symmetric parallelism

Compile-time race-freedom guarantee

```
1     x = new Integer(); x.val = 3;
2
3     y = x; z = x;
4     // y and z are readable aliases of x
5
6     a = new Integer(); b = new Integer();
7     // a and b are isolated
8
9     // frame away writable references (x)
10    { a.val = y.val; } || { b.val = z.val; }
11    // get back writable references (x)
12
13    x.val = 4;
```

Compile-time race-freedom guarantee



Asymmetric parallelism

Compile-time race-freedom guarantee

```
1     writable Integer x = ...;
2
3     // construct isolated list of isolated integers
4     isolated y = new IsolatedIntegerList();
5
6     // populate list ...
7
8     // Sort in parallel with other work
9     {
10       f = new SortFunc();
11       y.map(f);
12    } // 1
13    || { x.val = 3; } // 2
```

1 is the red thread
2 is the blue thread

Compile-time race-freedom guarantee

```
1     writable Integer x = ...;
2     isolated y = new IsolatedIntegerList();
3     // populate list ...
4
5     {
6        f = new SortFunc();
7        y.map(f);
8        y.append(x.val); // Type Error
9     }
10    || { x.val = 3; }
```

Compile-time race-freedom guarantee

Then the paper formalizes all the typing rules. For example:

$$\boxed{\Gamma_1 \vdash C \dashv \Gamma_2} \qquad \frac{t \neq \texttt{isolated}\ \_}{x : \_, y : t \vdash x = y \dashv y : t, x : t}\ \text{T-AssignVar} \qquad \frac{}{\vdash x = \texttt{new}\ T() \dashv x : \texttt{isolated}\ T}\ \text{T-New}$$

$$\frac{t'\ f \in T \quad p \neq \texttt{isolated} \vee t' = \texttt{immutable}\ \_ \quad t' \neq \texttt{isolated}\ \_ \vee p = \texttt{immutable}}{x : \_, y : p\ T \vdash x = y.f \dashv y : p\ T, x : p \rhd t'}\ \text{T-FieldRead}$$

$$\frac{t\ f \in T}{y : \texttt{writable}\ T, x : t \vdash y.f = x \dashv y : \texttt{writable}\ T, \mathsf{RemIso}(x : t)}\ \text{T-FieldWrite}$$

$$\frac{\texttt{isolated}\ T_f\ f \in T}{y : \texttt{writable}\ T \vdash x = \texttt{consume}(y.f) \dashv y : \texttt{writable}\ T, x : \texttt{isolated}\ T_f}\ \text{T-FieldConsume}$$

$$\frac{}{x : \_ \vdash x = n \dashv x : \texttt{int}}\ \text{T-Int} \qquad \frac{}{x : \_ \vdash x = b \dashv x : \texttt{bool}}\ \text{T-Bool} \qquad \frac{}{x : \_ \vdash x = \mathsf{null} \dashv x : p\ T}\ \text{T-Null}$$

$$\frac{\begin{array}{c} t'\ m(\overline{u'\ z'})\ p' \in T \qquad \vdash p \prec p' \qquad \overline{\vdash u \prec u'} \\ p = \texttt{isolated} \implies t \neq \texttt{readable}\ \_ \wedge t \neq \texttt{writable}\ \_ \wedge \mathsf{IsoOrImm}(\overline{z : t}) \wedge p' \neq \texttt{immutable} \end{array}}{y : p\ T, \overline{z : u} \vdash x = y.m(\overline{z}) \dashv y : p\ T, \mathsf{RemIso}(\overline{z : t}), x : t'}\ \text{T-Call}$$

$$\frac{\Gamma_1 \prec \Gamma_1' \quad \Gamma_1' \vdash C \dashv \Gamma_2' \quad \Gamma_2' \prec \Gamma_2}{\Gamma_1 \vdash C \dashv \Gamma_2}\ \text{T-SubEnv} \qquad \frac{\Gamma_1 \vdash C \dashv \Gamma_2}{\Gamma, \Gamma_1 \vdash C \dashv \Gamma, \Gamma_2}\ \text{T-Frame} \qquad \frac{\Gamma \vdash C \dashv \Gamma}{\Gamma \vdash C* \dashv \Gamma}\ \text{T-Loop}$$

$$\frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \quad \Gamma_2 \vdash C_2 \dashv \Gamma_3}{\Gamma_1 \vdash C_1; C_2 \dashv \Gamma_3}\ \text{T-Seq} \qquad \frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \quad \Gamma_1 \vdash C_2 \dashv \Gamma_2}{\Gamma_1 \vdash C_1 + C_2 \dashv \Gamma_2}\ \text{T-Branch} \qquad \frac{\Gamma, y : t', x : t, \Gamma' \vdash C \dashv \Gamma''}{\Gamma, x : t, y : t', \Gamma' \vdash C \dashv \Gamma''}\ \text{T-Shuffle}$$

Figure 5. Core typing rules.

$$\frac{\begin{array}{c} TD = \texttt{class}\ cn\ [<: T2]\ \{\overline{fld}\ \overline{meth}\} \qquad t'\ m(\overline{t}\ \overline{x})\ p' \in T2 \qquad P \vdash t \prec t' \qquad P \vdash p' \prec p \qquad P \vdash \overline{t'} \prec \overline{t} \\ p \neq \texttt{isolated} \qquad \forall i \in [1 \ldots n].\ P \vdash t_i \qquad P \vdash t \qquad \texttt{this} : p\ cn, \overline{t}\ \overline{x} \vdash C; \texttt{return}\ x \dashv \texttt{result} : t \end{array}}{P; TD \vdash t\ m(\overline{t}\ \overline{x})\ p\ \{\ C; \texttt{return}\ x;\ \}}\ \text{T-Method2}$$

Figure 6. Method override definition typing

Compile-time race-freedom guarantee

But there are only two rules for parallelism:

$$\frac{\mathsf{NoWrit}(\Gamma_1) \quad \mathsf{NoWrit}(\Gamma_2) \quad \Gamma_1 \vdash C_1 \dashv \Gamma_1' \quad \Gamma_2 \vdash C_2 \dashv \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma_1', \Gamma_2'} \text{ T-Par}$$

$$\frac{\mathsf{IsoOrImm}(\Gamma_1) \quad \Gamma_1 \vdash C_1 \dashv \Gamma_1' \quad \Gamma_2 \vdash C_2 \dashv \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma_1', \Gamma_2'} \text{ T-Async}$$

where $\mathsf{NoWrit}(\Gamma) \stackrel{\text{def}}{=} \forall (x : p\,T) \in \Gamma.\, p \neq \texttt{writable}$

Figure 8. Type rules for safe parallelism. IsoOrImm is defined in Figure 7

Compile-time race-freedom guarantee

So, what is the problem here?

```
1       immutable IntList l = getAgeData();
2       writable IntList counters = new IntList();
3
4       {
5          for (int i = 0; i < l.length / 2; i++)
6             counters.inc(l.get(i));
7       } || {
8          for (int i = l.length / 2; i < l.length; i++)
9             counters.inc(l.get(i));
10      }
11
12      writable Int k = 0;
13      writable IntList sorted = new IntList();
14      for (int i = 0; i < MAX_AGE; i++)
15         for (int j = 0; j < counters.get(i); j++)
16            sorted.set(k++, i);
```

Compile-time race-freedom guarantee

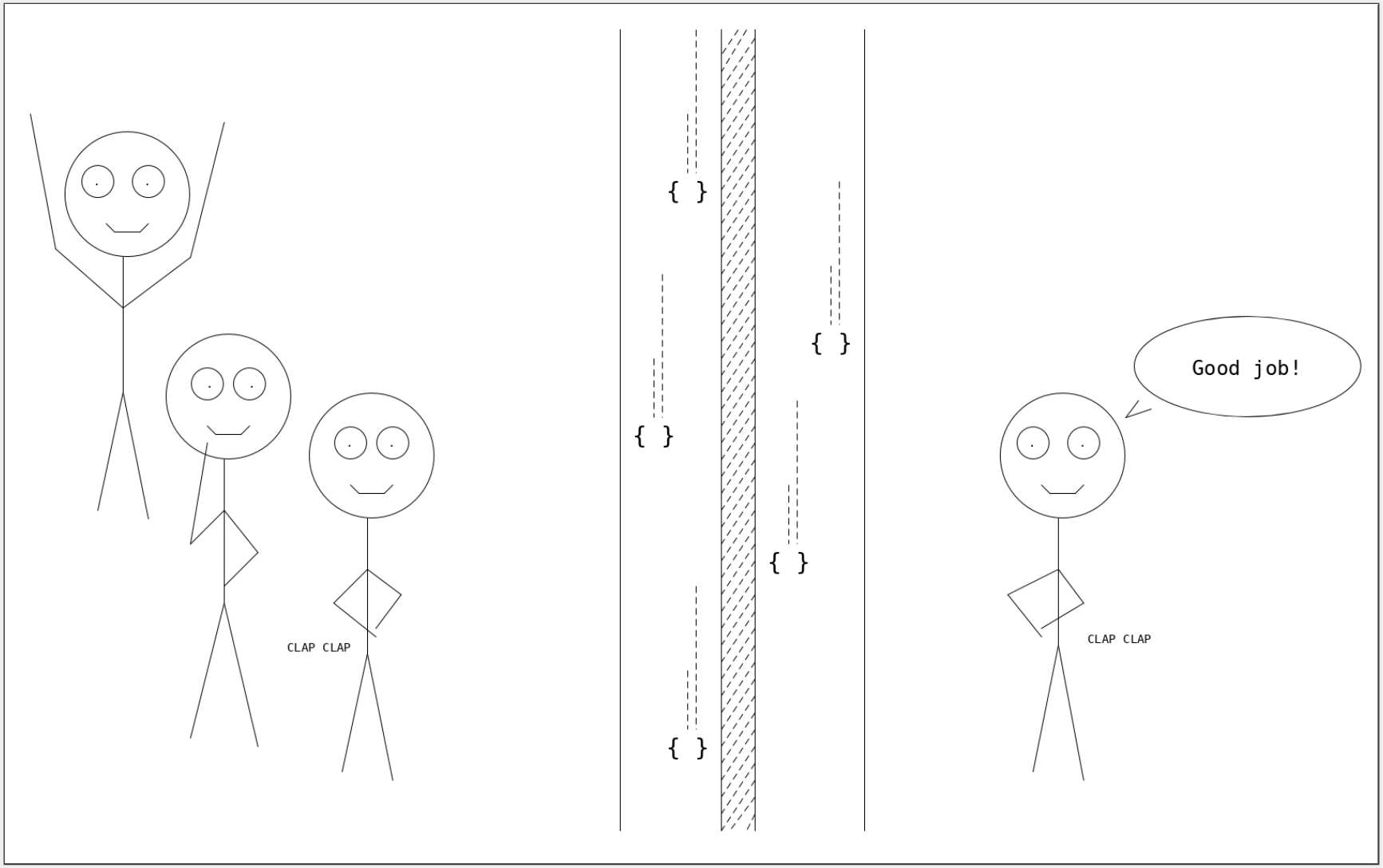
So, what is the problem here?

```
1       immutable IntList l = getAgeData();
2       writable IntList counters = new IntList();
3
4       {
5          for (int i = 0; i < l.length / 2; i++)
6             counters.inc(l.get(i)); // Type Error
7       } || {
8          for (int i = l.length / 2; i < l.length; i++)
9             counters.inc(l.get(i)); // Type Error
10      }
11
12      writable Int k = 0;
13      writable IntList sorted = new IntList();
14      for (int i = 0; i < MAX_AGE; i++)
15         for (int j = 0; j < counters.get(i); j++)
16            sorted.set(k++, i);
```

Compile-time race-freedom guarantee

```
1   immutable IntList l = getAgeData();
2   isolated IntList c1 = new IntList(), c2 = new IntList();
3
4   {
5      for (int i = 0; i < l.length / 2; i++)
6         c1.inc(l.get(i));
7   } || {
8      for (int i = l.length / 2; i < l.length; i++)
9         c2.inc(l.get(i));
10  }
11
12  writable Int k = 0;
13  writable IntList sorted = new IntList();
14  for (int i = 0; i < MAX_AGE; i++)
15     for (int j = 0; j < c1.get(i) + c2.get(i); j++)
16        sorted.set(k++, i);
```

Compile-time race-freedom guarantee



Questions?