The Future of Weak Memory 2024

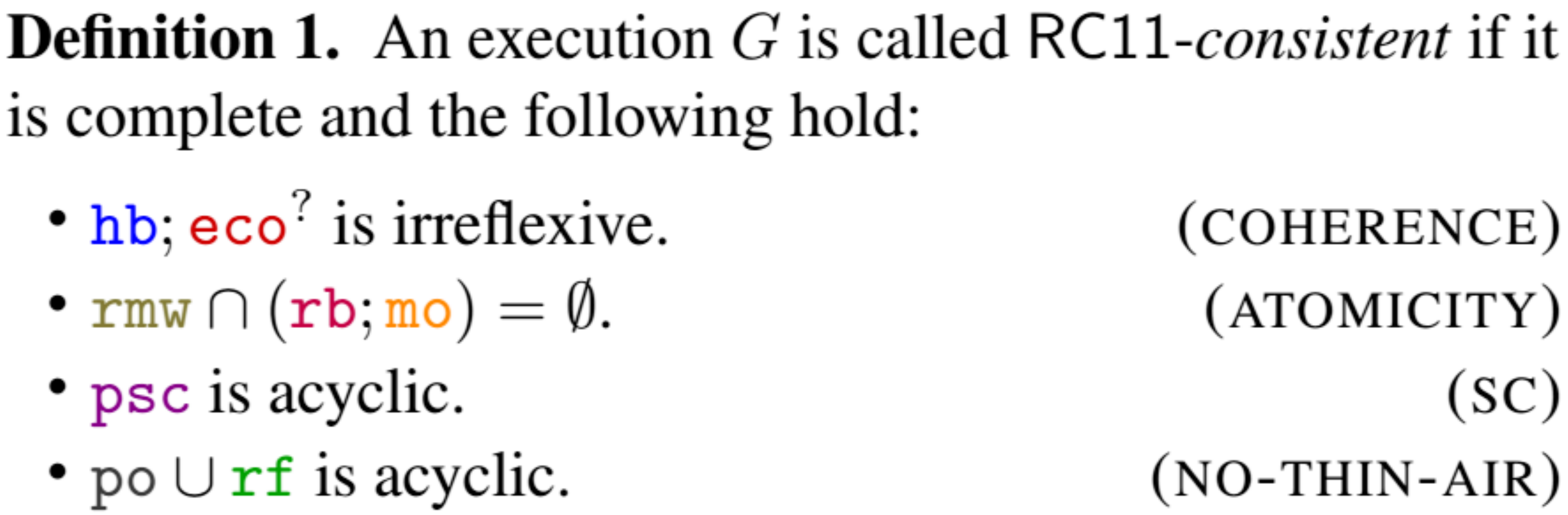
A case against semantic dependencies

***Ori Lahav*, Tel Aviv University**

Based on joint discussions with *Minki Cho, Chung-Kil Hur, Sung-Hwan Lee, Ben Simner*

Imagine a new model for C/C++ that works in three steps:

- Step 1: “calculate” a set S of **candidate program execution graphs**
- Step 2: given S, derive **semantic dependency** (`sdep`) for each graph
- Step 3: apply the **consistency predicate** from the C/C++ standard



Definition 1. An execution G is called RC11-*consistent* if it is complete and the following hold:

- $\mathtt{hb}; \mathtt{eco}^?$ is irreflexive. (COHERENCE)
- $\mathtt{rmw} \cap (\mathtt{rb}; \mathtt{mo}) = \emptyset$. (ATOMICITY)
- $\mathtt{psc}$ is acyclic. (SC)
- $\mathrm{po} \cup \mathtt{rf}$ is acyclic. (NO-THIN-AIR) ⇨ $\mathtt{sdep} \cup \mathtt{rf}$ is acyclic.

```
a = Y (rlx)
X = b (rlx)
```

RW-reorder →

```
X = b (rlx)
a = Y (rlx)
```

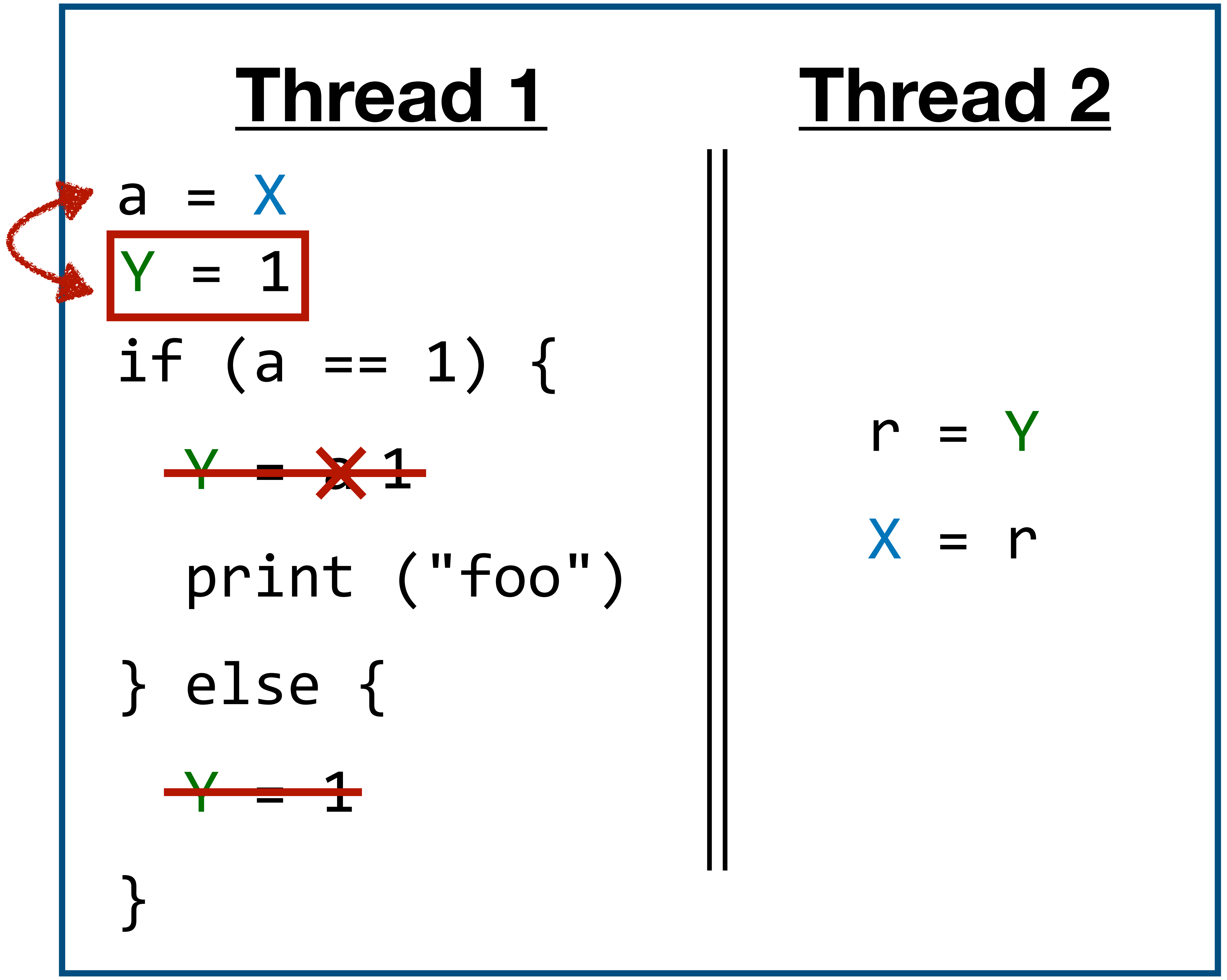
Imagine a new model for C/C++ that works in three steps:

- Step 1: "calculate" a set S of **candidate program execution graphs**
- Step 2: given S, derive **semantic dependency** (`sdep`) for each graph
- Step 3: apply the **consistency predicate** from the C/C++ standard

I believe this approach can't work. I argue via example that:

- *Step 2 cannot be thread-local*
- *Step 2 has to be aware of the consistency predicate in step 3*

Can “foo” be printed?



Thread 1

```
a = X
Y = 1
if (a == 1) {
  ~~Y = a 1~~
  print ("foo")
} else {
  ~~Y = 1~~
}
```

Thread 2

```
r = Y
X = r
```

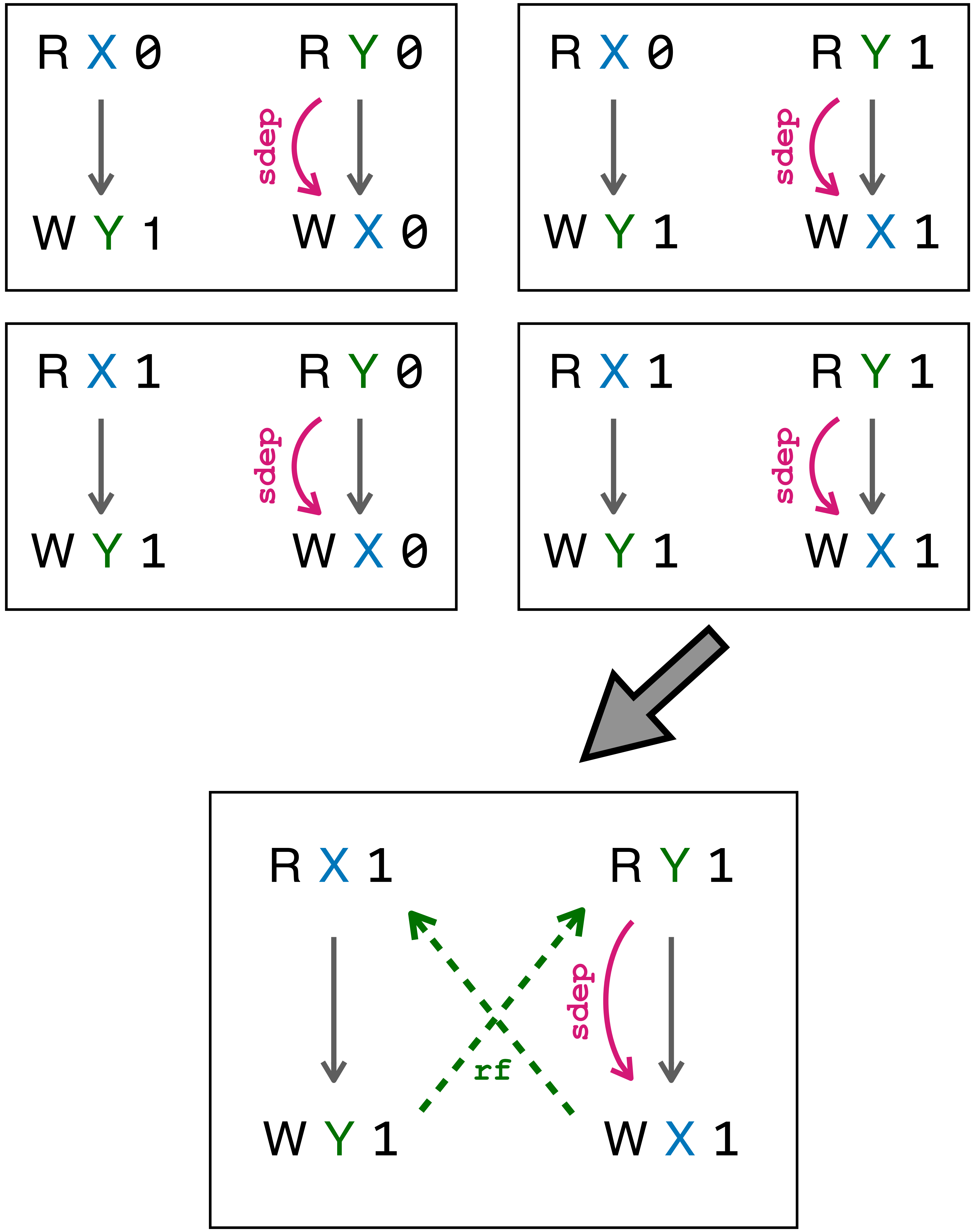
sdep approach

Thread 1

```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  Y = 1
}
```

Thread 2

```
r = Y
X = r
```



Main example

Thread 1

```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

Thread 2

```
r = Y
X = r
```

Thread 3

```
Z = 1
```

Thread 1

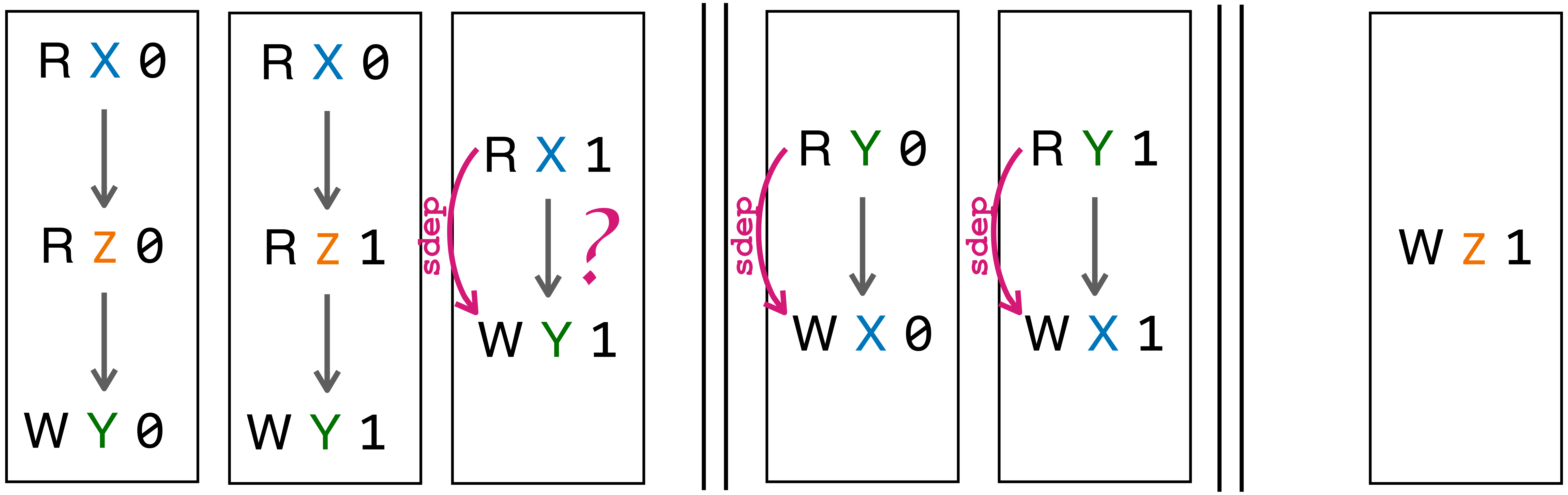
```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

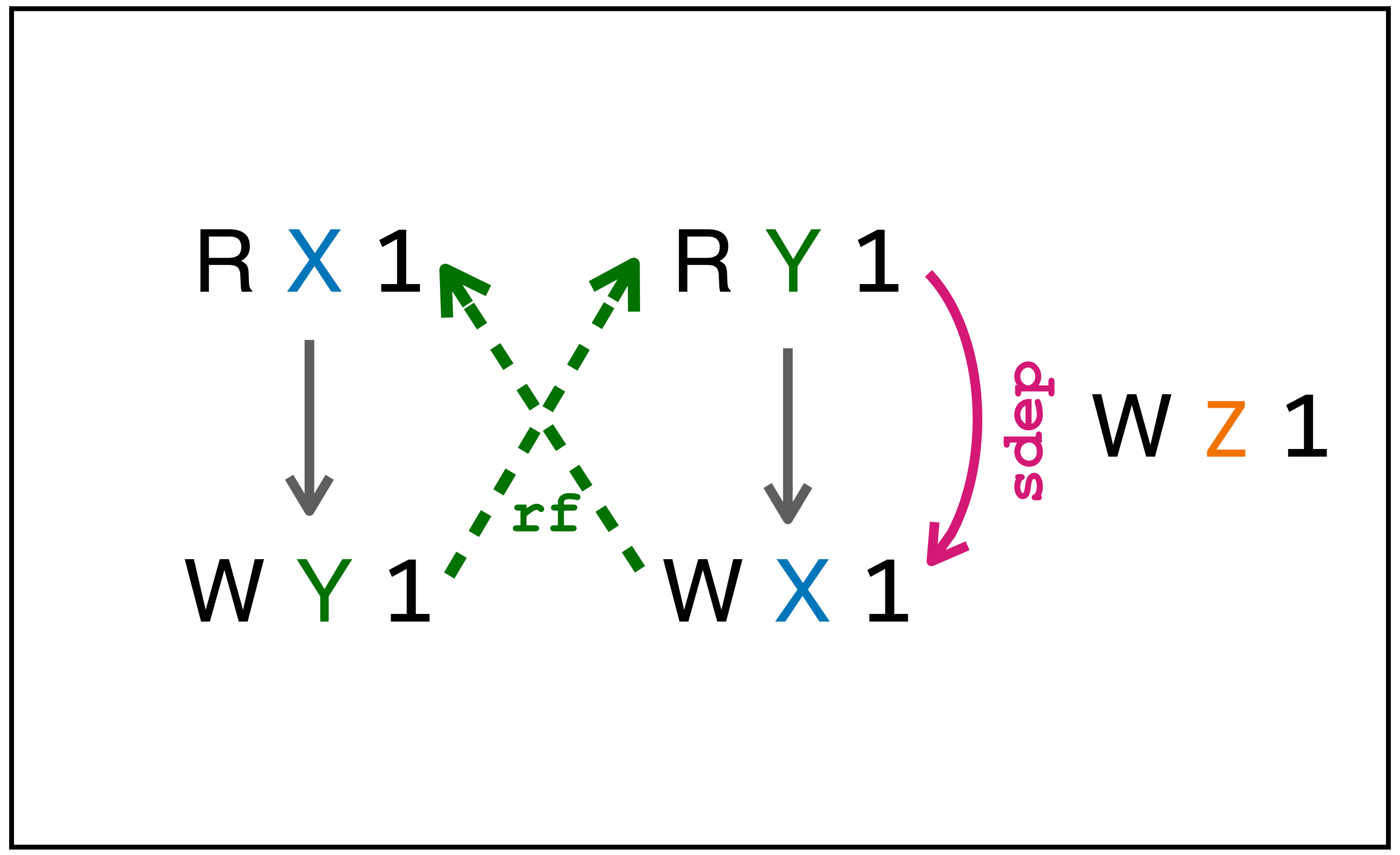
Thread 2

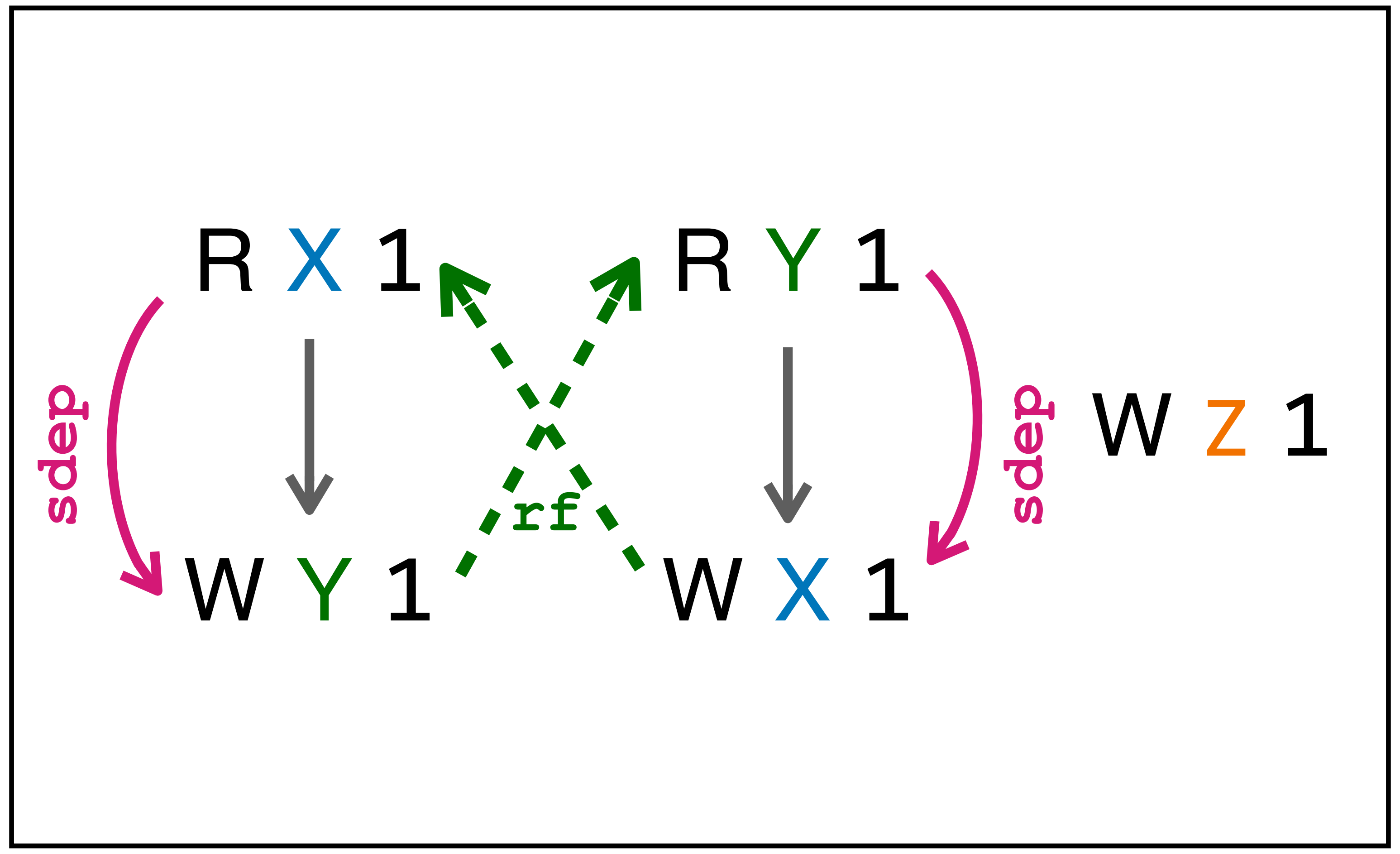
```
r = Y
X = r
```

Thread 3

```
Z = 1
```







Thread 1

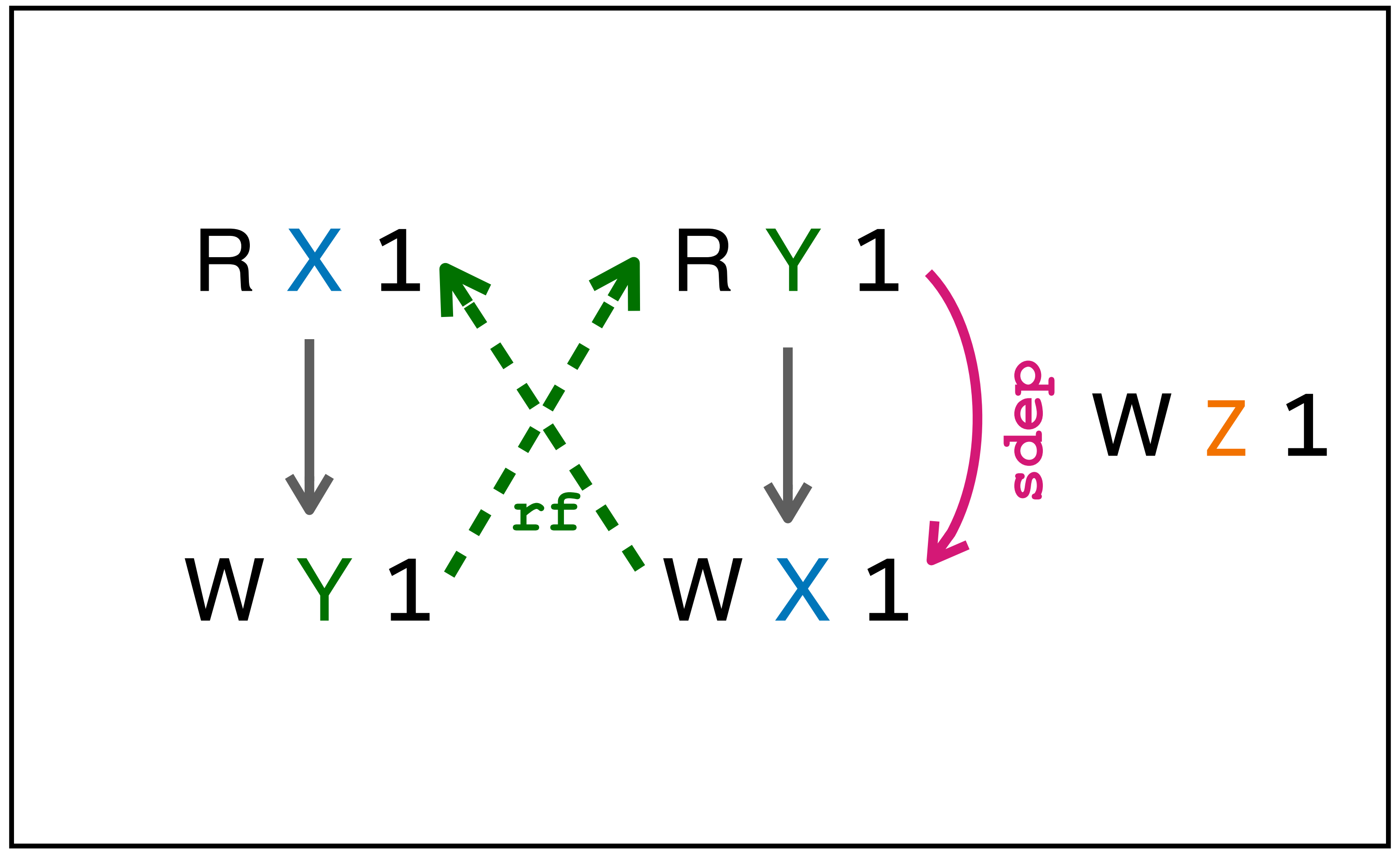
```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

Thread 2

```
r = Y
X = r
```

Thread 3

```
Z = 1
```



- Printing “foo” **has to be allowed**, assuming we allow compilers to:
  - Introduce redundant loads
  - Forward load across atomics:

Both are performed by LLVM/GCC on non-atomics (Z can be easily made non-atomic)

```
c = Z; a = X; b = Z → c = Z; a = X; b = c
```

Thread 1

```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

Thread 1

```
c = Z
if (c == 1) {
  a = X
  if (a == 1) {
    Y = a
    print ("foo")
  } else {
    b = Z
    Y = b
  }
} else {
  ...
}
```

Thread 1

```
c = Z
if (c == 1) {
  a = X
  if (a == 1) {
    Y = a
    print ("foo")
  } else {
    b = c
    Y = b
  }
} else {
  ...
}
```

Thread 1

```
c = Z
if (c == 1) {
  a = X
  if (a == 1) {
    Y = 1
    print ("foo")
  } else {
    b = 1
    Y = 1
  }
} else {
  ...
}
```

• • •

Thread 1

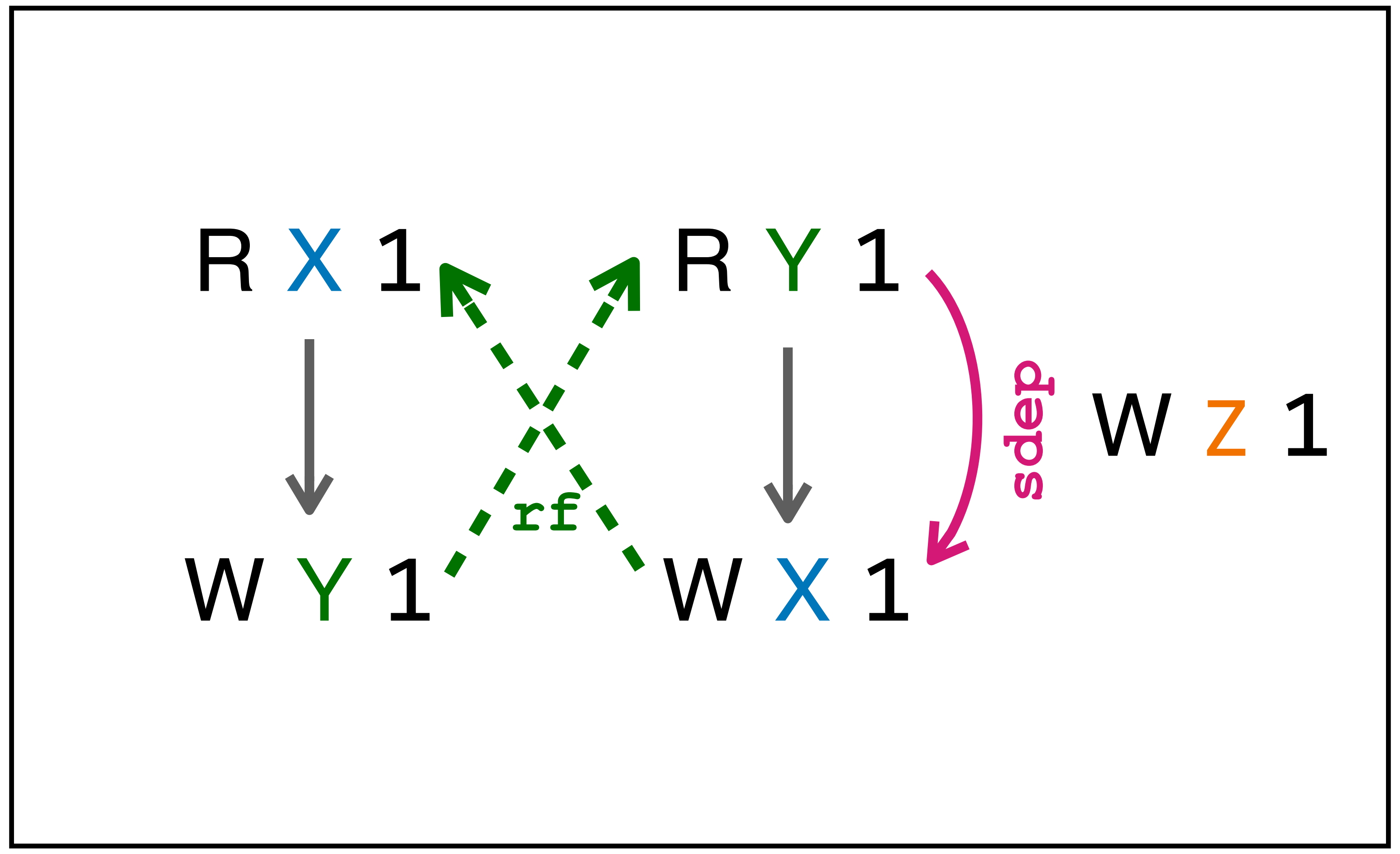
```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

Thread 2

```
r = Y
X = r
```

Thread 3

```
Z = 1
```



- Printing “foo” **has to be allowed**, assuming we allow compilers to:
  - Introduce redundant loads
  - Forward load across atomics:

Both are performed by LLVM/GCC on non-atomics (Z can be easily made non-atomic)

```
c = Z; a = X; b = Z → c = Z; a = X; b = c
```

Thread 1

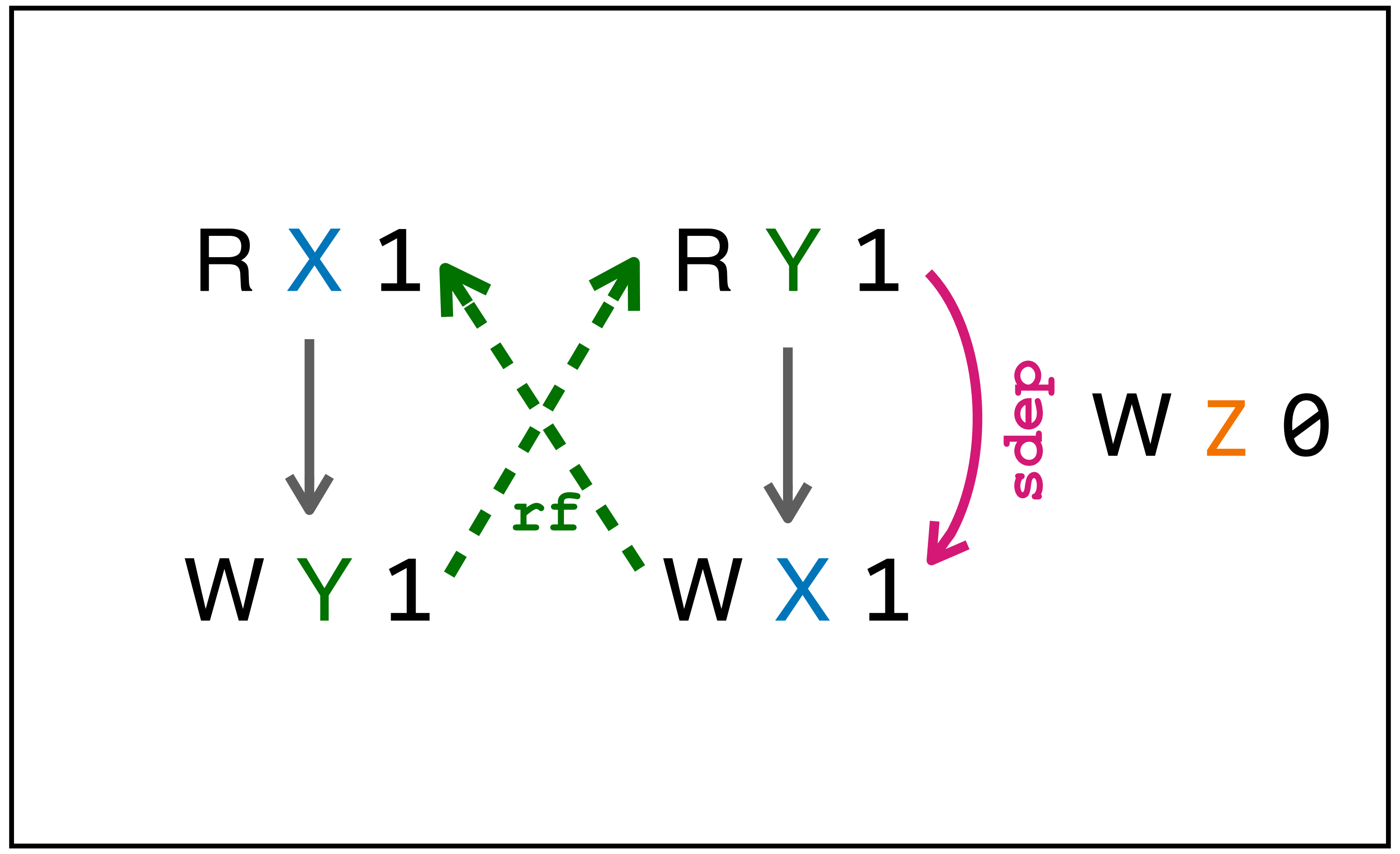
```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

Thread 2

```
r = Y
X = r
```

Thread 3

```
Z = 0
```



- With new Thread 3, printing “foo” **has to be disallowed** (thin-air!)

Thread 1

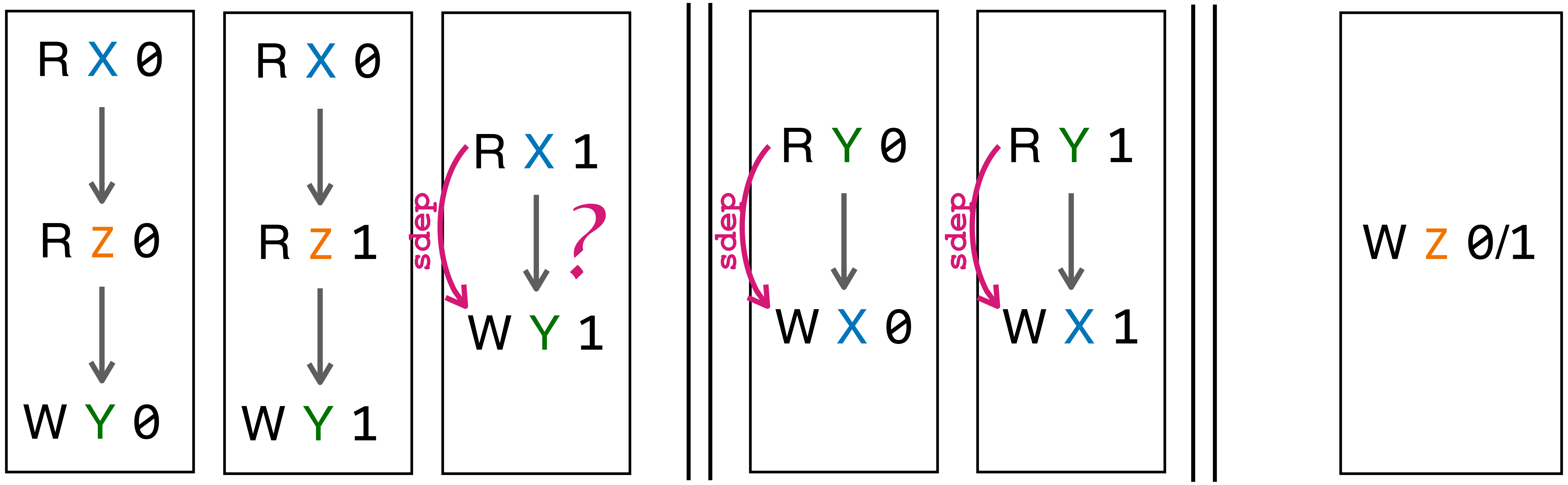
```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

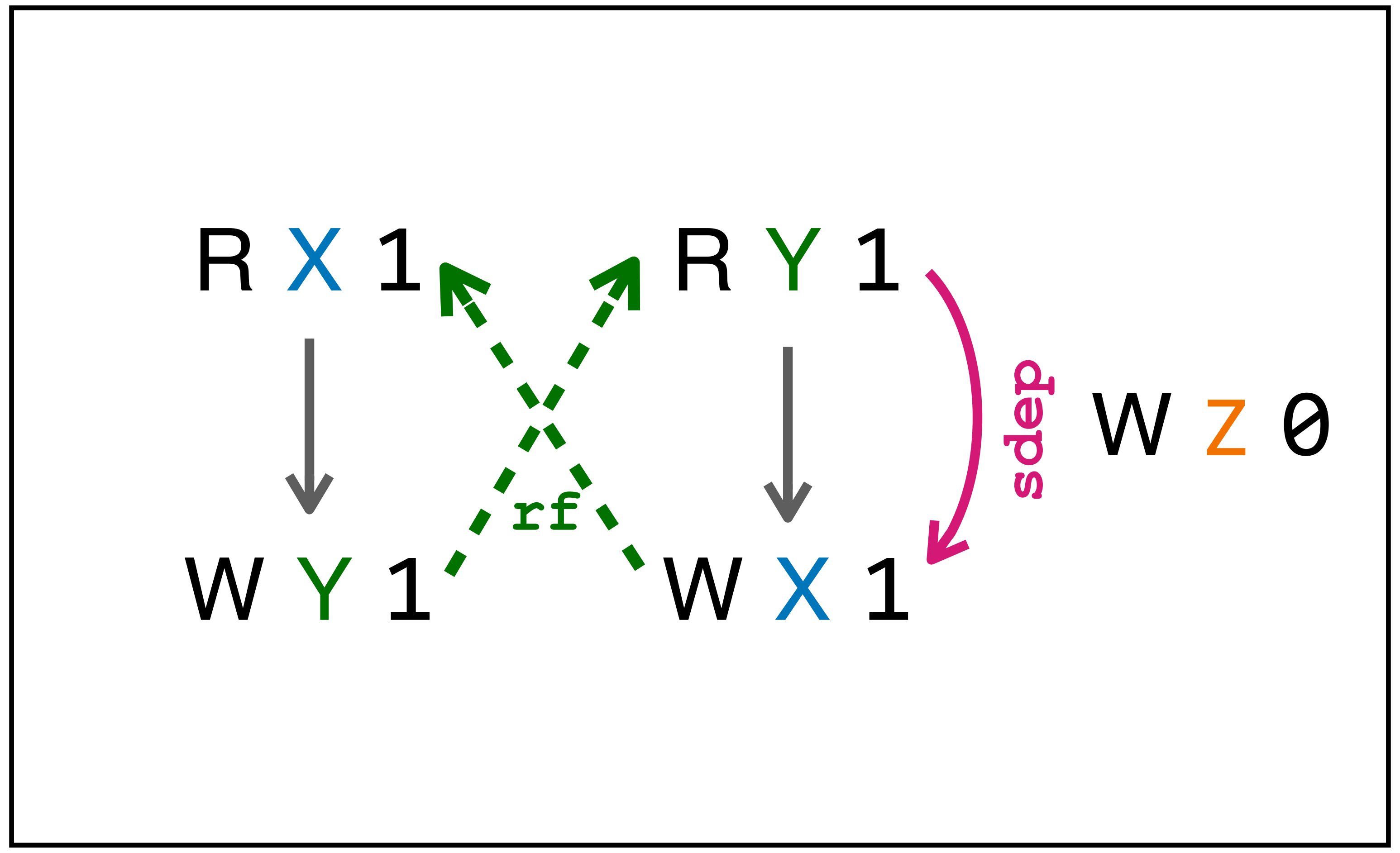
Thread 2

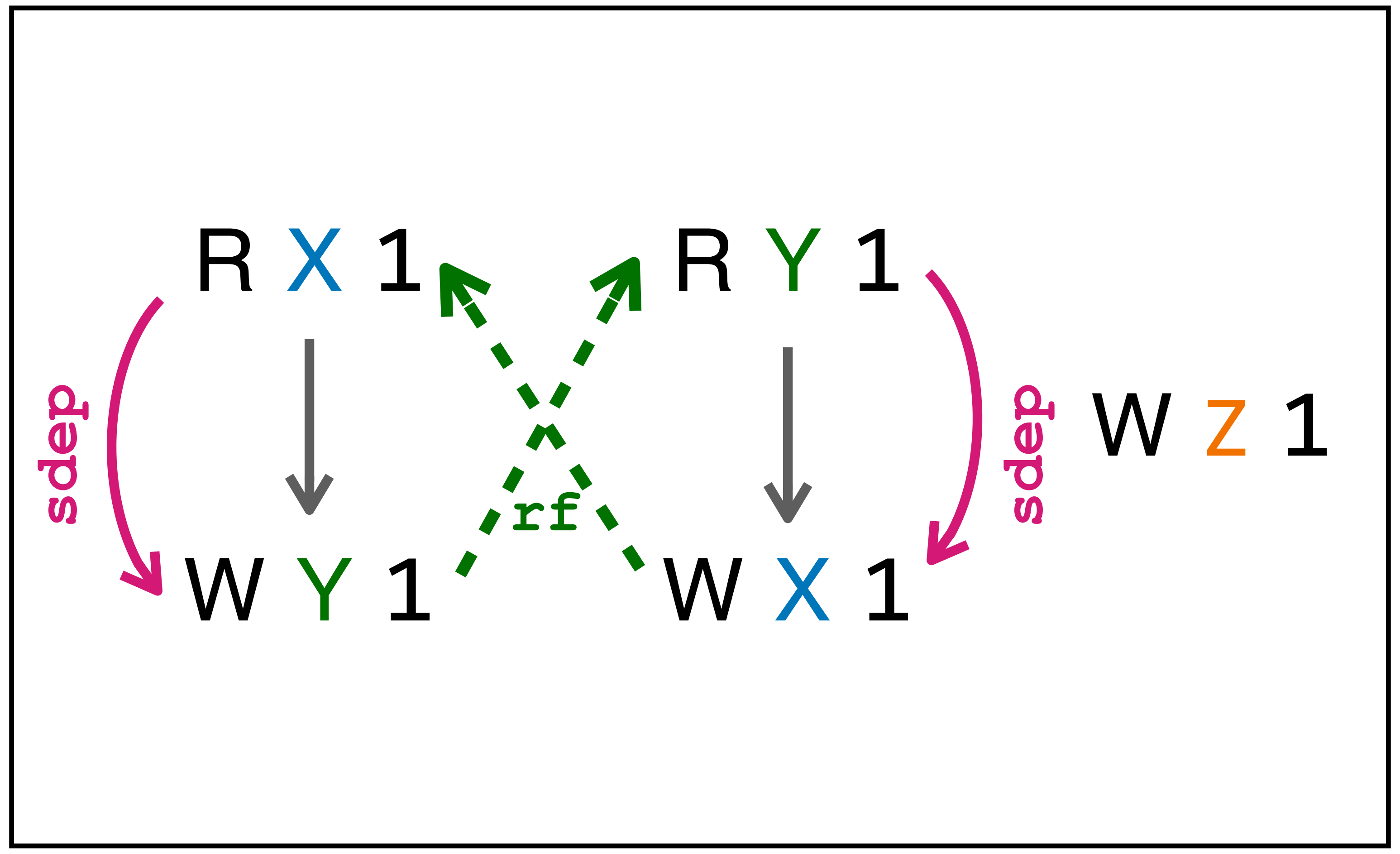
```
r = Y
X = r
```

Thread 3

```
Z = 0/1
```







Step 2 (**`sdep`** calculation) cannot be thread local!

Thread 1

```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

Thread 2

```
r = Y
X = r
```

Thread 3

```
Z = your_favorite_litmus_text()
```

```
U = 1          ||  V = 1
f = V          ||  d = FAA(V,1) acq
V = f rel      ||  e = U
g = V          ||

return(d==1 && e==0 && f==1 && g==2)
```

Step 2 (`sdep` calculation) depends on Step 3 (the consistency predicate)!

Thread 1

```
a = X
if (a == 1) {
  Y = a
  print ("foo")
} else {
  b = Z
  Y = b
}
```

Thread 2

```
r = Y
X = r
```

Thread 3

```
Z = your_favorite_litmus_test()
```

Substitution of Equivalents — “sanity condition” for weak memory models:

- If `f()` always returns `0` in a memory model M, then `f()` and `0` should be a equivalent in M

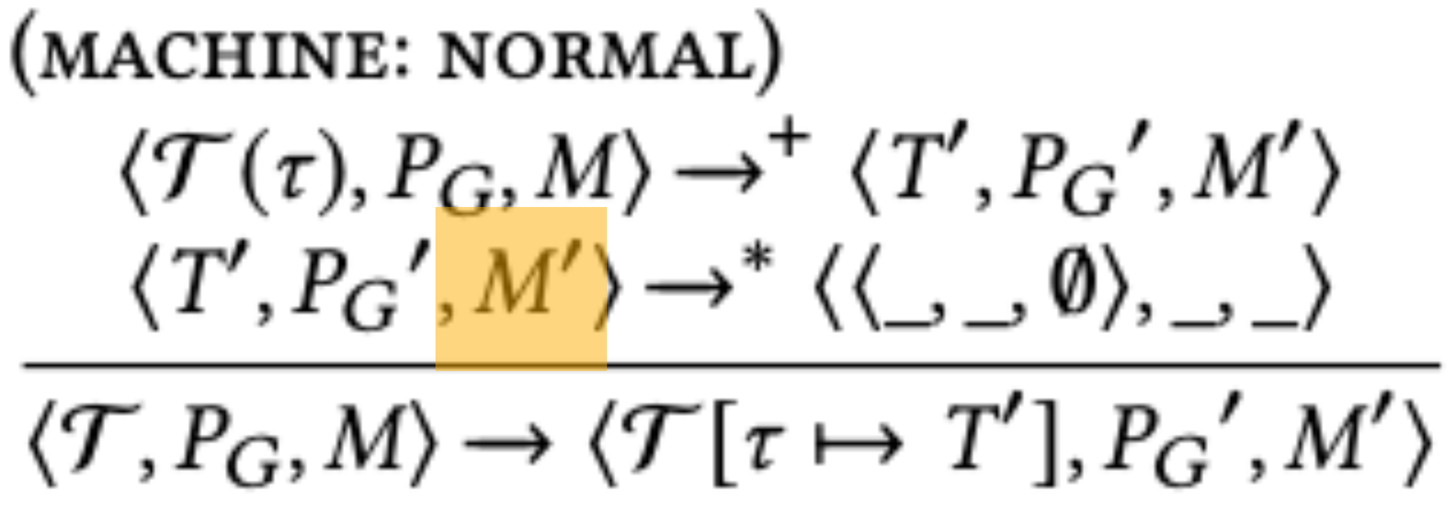
  (assuming `f()` uses a disjoint set of locations wrt rest of the program).

Reasoning-aware `sdep`? :(

- `sdep` calculation has to take intro account our reasoning principles.
  - No thin air values: `f()` never returns `1` in some (possibly inconsistent) execution $\Longrightarrow$ `sdep` must exist
  - A new (sound) program logic can prove that `f()` never returns `1` $\Longrightarrow$ `sdep` must exist
- We have a memory model for reasoning (weaker than the “real” model).
- For reasoning to be potentially precise, `sdep` **needs take into account the full consistency predicate.**

The source of the problem

- Semantic dependencies are "**dynamic**" rather than "**static**":
  - `sdep` $\Longleftrightarrow$ the model allows a thread to read some value at a certain program point.
  - Event-structure-based / pomset models / "Promising Semantics" capture such dynamic dependencies.
  - The approach we discussed fails to do so.



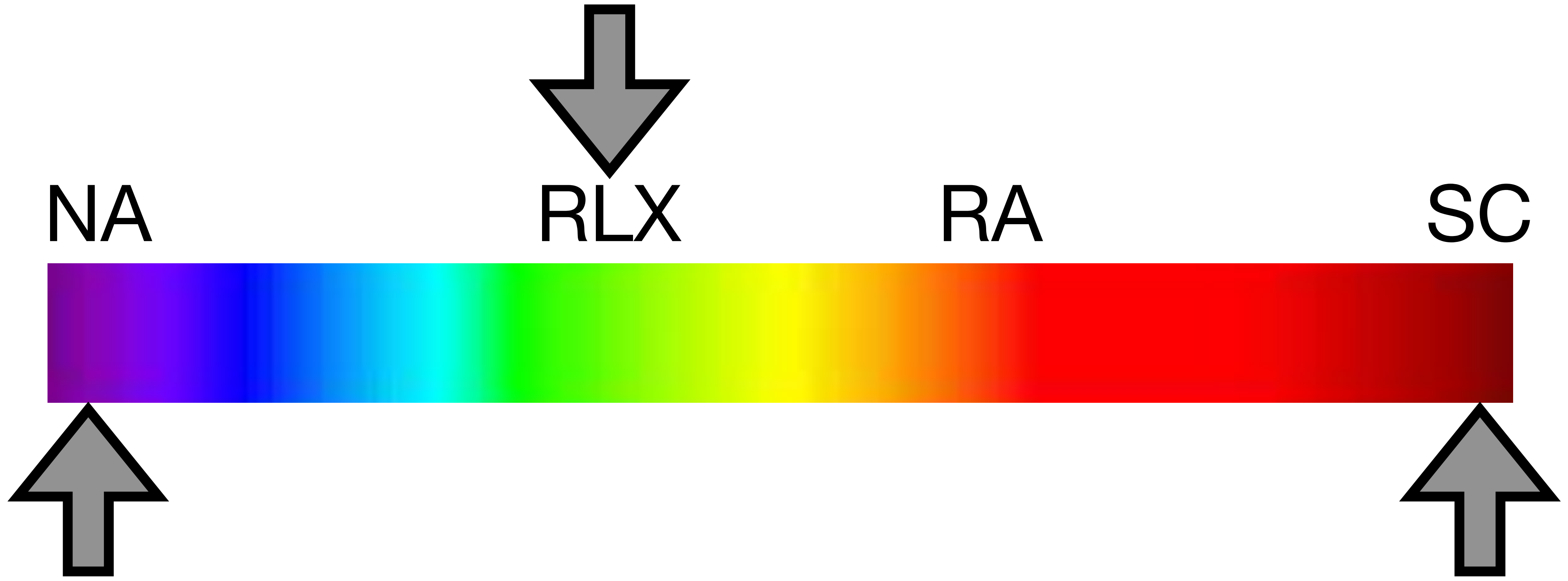
(MACHINE: NORMAL)

$$\frac{\langle \mathcal{T}(\tau), P_G, M\rangle \longrightarrow^{+} \langle T', P_G', M'\rangle \qquad \langle T', P_G', M'\rangle \longrightarrow^{*} \langle\langle \_, \_, \emptyset\rangle, \_, \_\rangle}{\langle \mathcal{T}, P_G, M\rangle \longrightarrow \langle \mathcal{T}[\tau \mapsto T'], P_G', M'\rangle}$$

see our PLDI’23 paper

A fresh look on the out-of-thin-air problem

- The discussion about the OOTA problem in C/C++ revolves around `memory_order_relaxed`
    - *Is it indeed expensive to forbid RW reordering of relaxed accesses?*
    - More provocatively: *do we really need relaxed writes?*



- A (more practical?) challenging problem arises with:
    - Strong accesses (SC) **or mutexes** that allow races
    - Weak accesses (non-atomic) that allow optimizations, **including load introduction**