**ABSTRACT**

In response to the rise of multicore processors, mainstream languages have begun to offer primitives for concurrent programming. To avoid the cost of inter-core synchronisation, the new C/C++ standard, C11 [2], offers weakly consistent relaxed operations, alongside traditional reads, writes and mutexes. When using relaxed operations, different threads may see different, apparently contradictory orders of events.

C11 permits a particularly surprising kind of relaxed behaviour: cycles in causality. Two conditional guards on different threads can be satisfied by writes down the other branch. Both branches execute, even though each appears to depend on the other.

```c
x = 0; y = 0;
if (x == 42) || if (y == 23)
y = 23; || x = 42;
```

Such behaviour could potentially be produced by hardware speculation or compiler optimisations, but it is unclear whether it occurs in current implementations. Causal cycles are known to be problematic: the Java standard tried to rule them out, but inadvertently forbade several widely-used optimisations [3]. C11 heavily deprecates cycles, but falls short of banning them outright.

A property is compositional if each program sub-component can be analysed separately while assuming its surrounding context is well-behaved. By allowing programs to be decomposed, compositionality aids documentation, testing and verification. In most languages, safety properties (e.g., absence of memory faults) are compositional, because a given fault must originate in the sub-component or its context, but not both. Causal cycles in C11 allow two faults to cause each other, which violates this assumption and breaks compositionality [1].

Fixing compositionality in C11 requires a condition for ruling out cycles which avoids Java’s problems. This remains a difficult open problem.

**BODY**

C/C++ permit seemingly-impossible cycles in causality. This breaks compositionality: two apparently safe programs may fault when composed.

**REFERENCES**