Coping with shared mutable state in a typestate-oriented concurrent language

João Mota - *jd.mota@campus.fct.unl.pt*

A Ph.D. Thesis Plan presented at PLDI@SRC 2024

(1) CONTEXT: SAFETY IS KEY

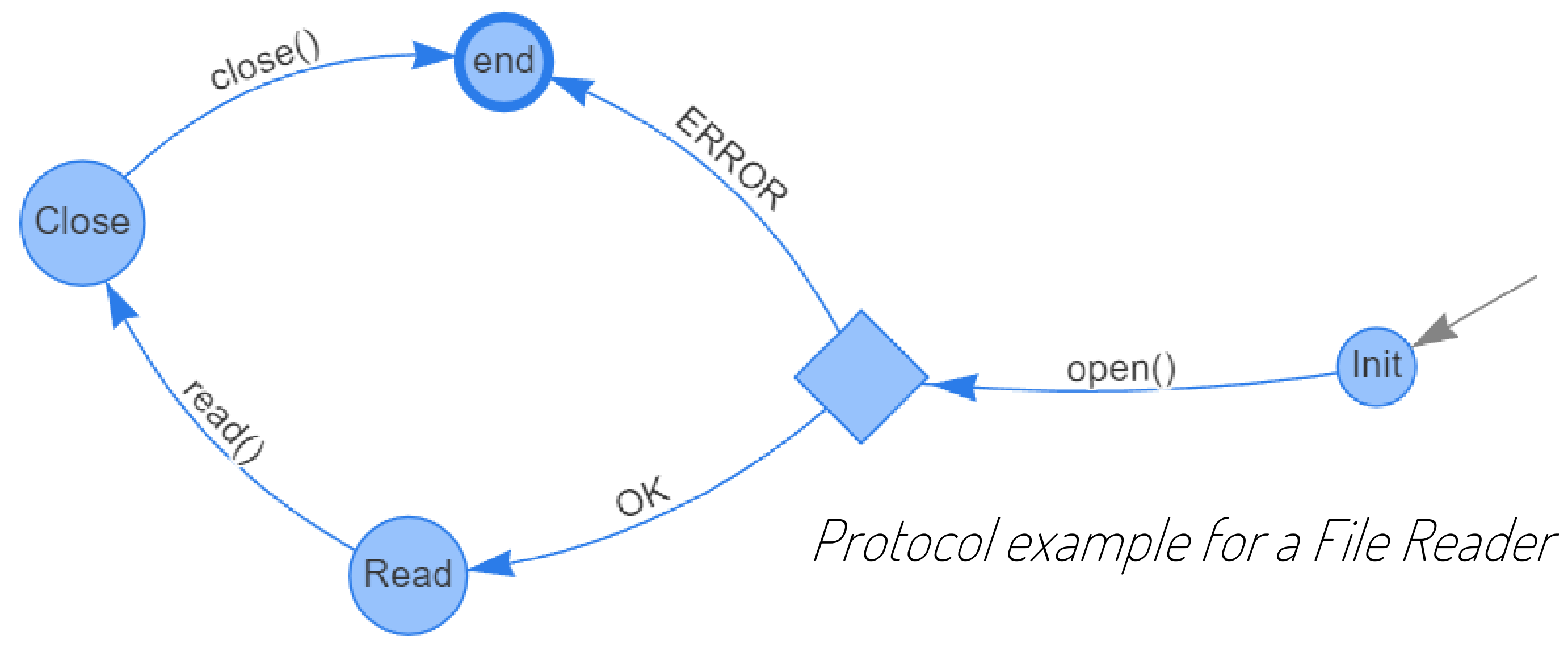
> *Studies … found that about 70 percent of all security vulnerabilities are … memory safety issues* - InfoWorld, 2024

> *Future Software Should Be Memory Safe* - White House, 2024

JaTyC (Java Typestate Checker) analyses Java code with respect to **typestates** (i.e., object protocols). Statically detects violations of:

- **protocol compliance:** method calls are in order
- **protocol completion:** protocols reach the "end" state
- **memory-safety:** no null deref., no data-races, no leaks

But forces linear use of objects…



Protocol example for a File Reader

(2) PROBLEM: HOW TO DEAL WITH SHARED MUTABLE STATE?

Type systems to deal with **protocols** and **shared mutable state** in languages with modern features are lacking.

Issue 1: **Sharing patterns are limited** to fixed sets of capabilities or certain ownership disciplines.

Issue 2: **No distinction between** thread-local and thread-shared data, forcing locks usage even in sequential code.

Issue 3: **No protocol completion** when objects are shared (because affinity has been preferred).

(3) MOTIVATING EXAMPLE:
ASYNC. COOPERATION PROTOCOL

JavaScript code example features:

1. **Complex cooperation**: producer adds items, consumer receives them and **completes** the protocols.
2. **Single-threaded asynchronous code** (i.e., concurrency without parallelism from event loops).

```
const queue = new Queue();

async function producer() {
  const data = await otherTask1();
  // Mistake: consumer may have finished at this point
  // queue.use();
  queue.add(data);
  queue.unuse();
}

async function consumer() {
  while (queue.inUse()) {
    const data = await queue.take();
    await otherTask2(data);
  }
}

queue.use(); // Correct: claim use before starting
await Promise.all([ producer(), consumer() ]);
```

Mainstream languages do not give the desired guarantees.
Static typestated approaches do not accept this code:

1. Limited sharing prevents using the **queue**
2. Asynchronous code is not supported

and do not guarantee protocol completion.

(4) GOAL: SAFE SHARED STATE IN A TYPESTATE-ORIENTED LANGUAGE

Design a typed core OO language to reason about **shared mutable state** and **objects protocols**, guaranteeing **memory-safety** and **thread-safety**, protocol **compliance** and **completion**.

Solution components:

- Overcome sharing limitations to reason about cooperation protocols.
- Support single-threaded asynchronous code, which does not require locks.
- Guarantee protocol completion.

RQ: What if we also wanted to support multi-threading? What if we also wanted *droppable states*?

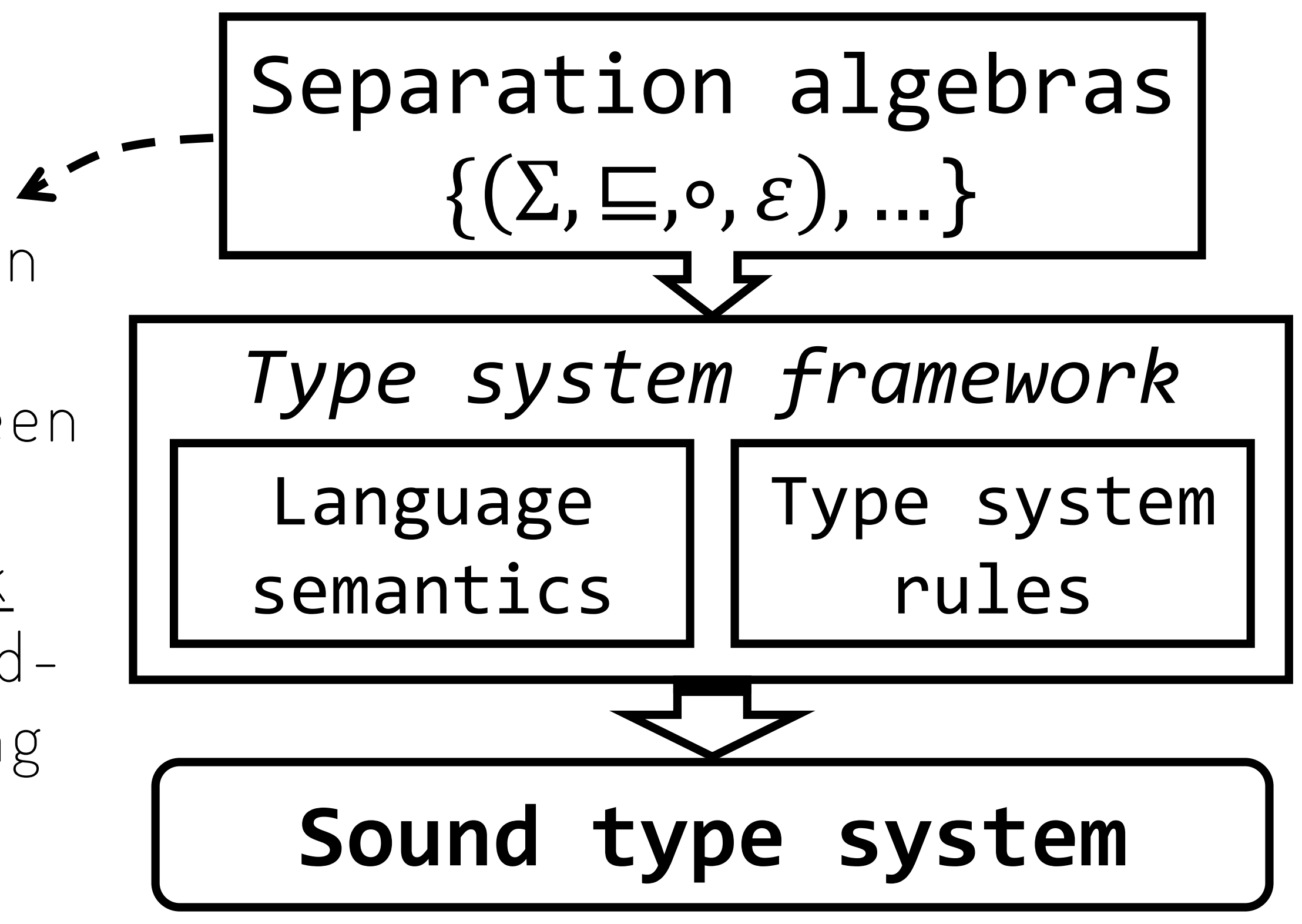
"Create another type system?"

Avoid "the next 700 type systems"

1. **Develop a type system framework** parametric over suitable **separation algebras**, allowing more expressive ways to reason about shared data. Inspired in **Iris**.
2. **Final product**
   - Integrate *typestates* with separation algebras: *access permissions* and *rely-guarantee protocols*.
   - Shareable typestate based type checker for TypeScript and Java.

Novelties to type-check the example:

- allow to distinguish between thread-local and thread-shared resources, and between affine and linear ones.
- allow to temporarily break assumptions made by thread-local resources until yielding back to the event loop.



Separation algebras
$\{(\Sigma, \sqsubseteq, \circ, \varepsilon), \ldots\}$

Type system framework

Language semantics

Type system rules

Sound type system

NOVA SCHOOL OF SCIENCE & TECHNOLOGY
NOVALINCS LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS
FCT Fundação para a Ciência e a Tecnologia

ACKNOWLEDGMENTS
This work was partially supported by NOVA LINCS (UIDB/04516/2020) via FCT (doi:10.54499/2021.05297.BD)