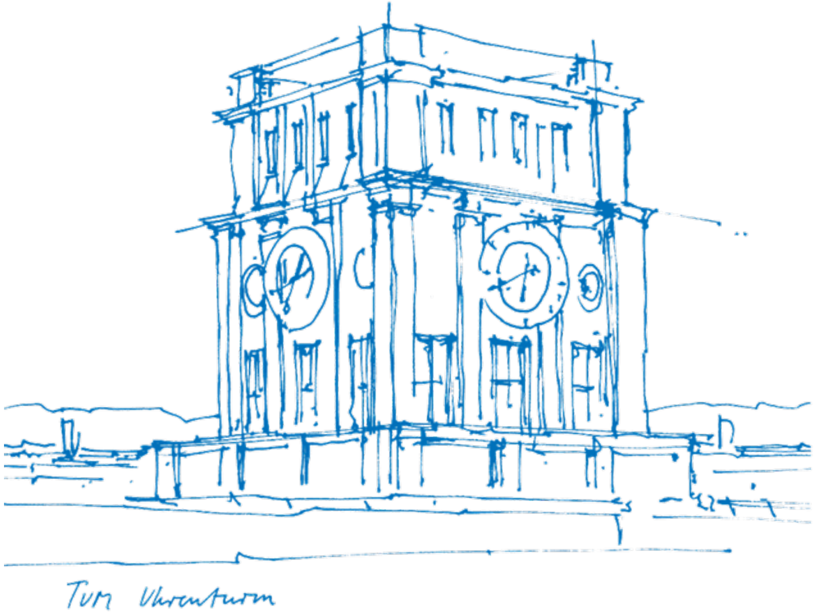
A Type System for Safe, Structured Concurrency in Scala

Sebastian Willenbrink

Munich, July 14, 2024

written at KTH, Sweden



Concurrency is hard

- Deadlocks
- Data races
- Nondeterminism
- Livelocks and more
- How do we solve these?

Motivation

- New languages lack an ecosystem
  → Extend an existing language: Scala
- Previous extensions:
  - require annotations
  - solve only some issues
- An extension for Scala which:
  - requires no annotations
  - is deterministic and free of deadlocks and data races

Fork-Join Model

- Familiar from C
- `fork` creates threads
- `join` blocks until a thread terminates
- Issues:
  - Deadlocks: Unclear termination order
  - Data races: Data shared by default

```
def compute(x : Object, y : Object) {
    var id = fork(() => x.setValue(1))
    y.setValue(2)
    join(id)
}
```

Async-Finish Model (AFM)

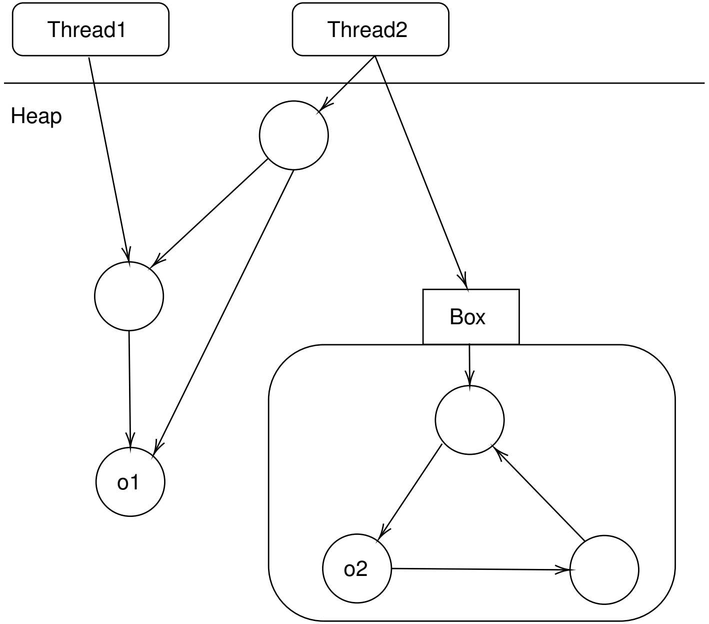
- Variant of Fork-Join
- `async` identical to `fork`
- `finish` awaits all threads started in its scope and their descendants
- The AFM is
  - deadlock free
  - deterministic if no data races occur
  - sometimes too restrictive

```
def compute(x : Object, y : Object) {
    finish {
        async(() => x.setValue(1))
        async(() => y.setValue(2))
    }
}
```

Alias Control

- Aliasing causes data races
- External uniqueness is enough
- LaCasa introduced boxes to encapsulate subgraphs
- References cannot cross box boundaries
  →No captures!
- `async` takes a box:

```
def compute(boxX : Box[Object], boxY : Box[Object]) {
    finish {
        async(boxX, x => x.setValue(1))
        async(boxY, y => y.setValue(2))
    }
}
```



Object Capabilities

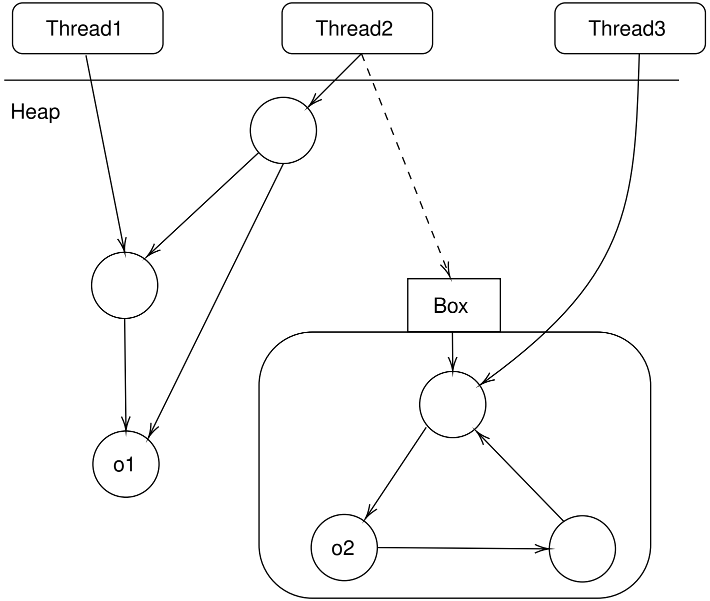
- Forbidding captures is not enough
- Global variables break encapsulation
- Object capabilities constraint:
  - Only explicit references
  - Only create objects with same constraint
- Constraint is inferrable
- Ca. 50% of OS Scala code adhere to object capabilities

Affinity

- Boxes may still be aliased
  → Guard boxes with permissions
- Create matching permission with box
- Permission gives access
- Opening a box consumes the permission
  → Affine types
- Simulate them using CPS:

```
async(boxX, permX, x => x.setValue(1)) {
    /* continuation */
}
```

- Permissions are inferred automatically

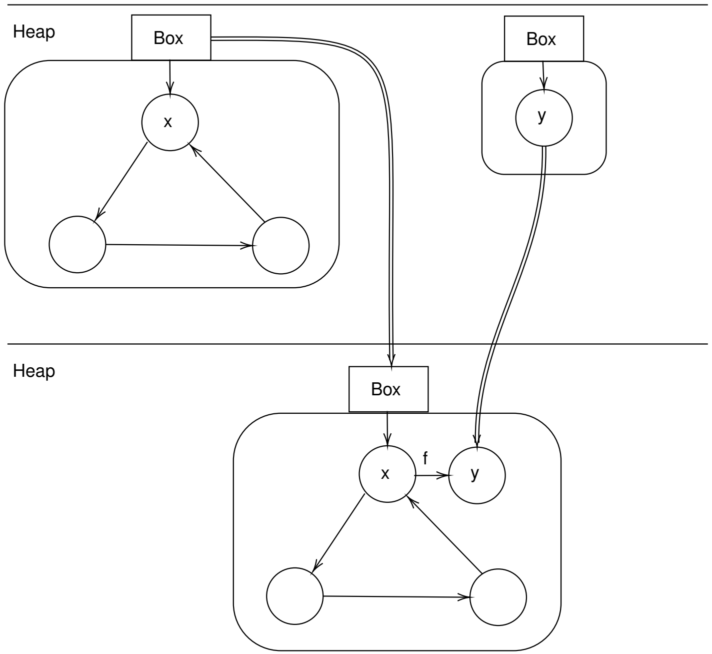


Accessing results

- Boxes contain results of threads
- But the permission was consumed
  → `finish` recovers permissions:

```
finish {
    async(boxX, permX, x => x.setValue(1)) {
        /* continuation */
    }
}
```

- Boxes can be merged:
  `capture(x.f, y)` stores `y` in `x.f`



Formalization

- Formalized using operational semantics
- Based on LaCasas formalization
- Proofs of thread isolation and progress
- Preservation and determinism not shown but AFM is deterministic

Challenges

- `capture` permanently destroy boxes
  → Continuations of `capture` never return
- `finish` recovers permissions from `async`
  → Continuations of `async` return to enclosing `finish`
- The threads have complicated invariants and interdependencies
  - Concurrently running threads must have distinct permissions
  - A parent may share permissions with its child while waiting

```
def f(boxX : Box[a], boxY : Box[b]) {
    finish {
        async(boxX, x => g()) { }
        /* Unreachable */
    }
    /* Reachable */
    finish {
        capture(boxX.f, boxY) { }
        /* Unreachable */
    }
    /* Unreachable */
}
```

Conclusion

- Type system that combines AFM with LaCasa
- Deterministic, deadlock free and data race free
- Deadlock freedom and data race freedom shown
- But:
  - Limited concurrency model
  - Preservation and determinism not shown
  - Purely theoretical

Thank you!