Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO’16
Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO’16
Concurrent Imposes Many Concerns

List $l = \ldots$;
$l$.add$(x)$;

Is it aliased?

Is it accessed concurrently?

Is it thread-safe?

Does it encapsulate its representation?

Is synchronisation implicit or explicit?

Are its subtypes thread-safe?
Aliasing and Concurrency Control

```cpp
assert c1.value() == 42;
c1.inc();
assert c1.value() == 43;
c2.inc();
assert c1.value() == 43;
```

May alias?
Aliasing and Concurrency Control

\[ T_1 \]

```
assert c1.value() == 42;
c1.inc();
assert c1.value() == 43;
```

May alias?

```
c2.inc();
```

\[ T_2 \]

Properly synchronised?

Wasteful if unaliased!

Locking too much leads to other problems

Lock EVERYONE!
Reference Capabilities

• A capability grants access to some resource.

• The type of a capability defines the interface to its object.

• A capability assumes exclusive access.
  - Thread-safety ⇒ No data-races

• How thread-safety is achieved is controlled by the capability’s mode.
Modes of Concurrency Control

- **Exclusive modes**

  - **linear** Globally unique
  - **thread** Thread-local

- **Safe modes**

  - **read** Precludes mutating aliases
  - **locked** Implicit locking
Modes of Concurrency Control

*Dominating modes*

- linear
- thread
- locked

Guarantees mutual exclusion

*Subordinate mode*

Precludes mutating aliases

Encapsulated
Capability = Trait + Mode

- Capabilities are introduced via traits

```
trait Inc
  require var cnt : int
  def inc() : void
    this.cnt++
```

```
trait Get
  require val cnt : int
  def value() : int
    return this.cnt;
```

- Modes control why they are safe

  **linear** Inc — Globally unique increment capability

  **locked** Inc — Implicitly synchronised increment capability

  **read** Inc — A read-only increment capability

  **read** Get — A read-only capability for getting the value
Classes are Composed by Capabilities

class Counter = Inc ⊕ Get {
    var cnt : int
}

linear, thread, locked, read, subordinate
Aliasing and Concurrency Control (revisited)

```java
class LocalCounter = thread Inc ⊕ read Get

assert c1.value() == 42;
c1.inc();
assert c1.value() == 43;
```

May not alias!

```java
class SharedCounter = locked Inc ⊕ read Get

c2.inc();
```

Properly synchronised!

Implemented by a readers-writer lock
Composite Capabilities

- A capability *disjunction* $A \oplus B$ can be used as $A$ or $B$, but not at the same time.
- Capabilities that do not share data should be usable in parallel...

```scala
trait Fst {
  require var fst : int
  ...
}

trait Snd {
  require var snd : int
  ...
}
```

- A capability *conjunction* $A \otimes B$ can be used as $A$ and $B$, possibly in parallel.

```scala
class Pair = linear Fst \otimes linear Snd {
  var fst : int
  var snd : int
}
```
let p = new Pair();
let f, s = consume p;
finish{
    async{f.set(x)}
    async{s.set(y)}
}
p = consume f + consume s
Packing and Unpacking

```javascript
let p = new Pair();
let f, s = consume p;
finish{
    async{f.set(x)}
    async{s.set(y)}
}
p = consume f + consume s
```
Packing and Unpacking

```javascript
let p = new Pair();
let f, s = consume p;
finish{
    async{f.set(x)}
    async{s.set(y)}
}
p = consume f + consume s
```
Subordination and Trait-Based Reuse

- Reuse traits across different concurrency scenarios
- Separate business logic from concurrency concerns

```scala
trait Add<T> {
  require var first : Link<T>
  def add(elem : T) : void
  ...
   this : subord Add<T>
}

class List<T> = Add<T> ⊕ ...
  var first : Link<T>

class SynchronizedList<T> = Add<T> ⊕ ...
  var first : Link<T>
```

Annotations in type declarations only
No effect tracking or ownership types
Can assume exclusive access
Reference Capabilities as Primitives

Ownership

External uniqueness

Regions and effects

class Tree = Left ⊗ Right ⊗ Elem
var left : Tree
var right : Tree
var elem : int

Single writer, multiple readers

... multiple disjoint writers

A ⊗ B

A

B
Kappa: Insights, Status and Future Work

Elias Castegren, Tobias Wrigstad

IWACO’16
Active Objects as a Mode of Synchronisation

- The message queue of an active object can replace the synchronisation of locks

```java
class ActiveCounter {
    var cnt : int

    def inc() : void
        this.cnt++

    def get() : int
        return this.cnt
}
```

- Active by default
Active Objects as a Mode of Synchronisation

- Opens up for new combinations

- `active` + `linear` Actor with unsynchronised initialisation methods
- `active` + `locked` Actor with priority channel
- `active` + `subord` Actor nested in another actor
- `active` ⊗ `active` Actor with parallel message queues
Array Capabilities
A Hierarchy of Capabilities

- Capability
  - Exclusive
    - Linear
  - Shared
    - Optimistic
      - Atomic
    - Pessimistic
      - Lock-Free
      - Locked
      - Active
  - Unsafe
    - Oblivious
      - Read
    - Immutable
      - ...
Conclusions

- Reference capabilities is a promising approach for thread-safe OO programming
- Brings together ideas from a wide variety of work in a unified system

Ownership/Universe types
Linear/Unique references and external uniqueness
Read-only/Immutability
Regions and effects
Fractional permissions
...

"Can your system do this?"

Yes ➔ Great!

No (not yet) ➔ Great!
Thank you!

Let’s talk more at the poster session!