Reference Capabilities for Concurrency Control

Elias Castegren, Tobias Wrigstad

ECOOP’16
Concurrency Imposes Many Concerns

List l = ...;
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

T₁

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

assert c1.value() == 43;

T₂

c2.inc();

May alias?

Properly synchronised?

LOCK ALL THE THINGS!
Aliasing and Concurrency Control

May alias?

T1

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

T2

c2.inc();

Properly synchronised?

Over-synchronisation hurts performance
Under-synchronisation hurts correctness
Aliasing and Parallelism

Data parallelism
(a parallel map)

Can two nodes alias?

Can two elements alias?

Can two elements share data?

Task parallelism
(map \( f_1 \parallel map f_2 \))

Is \( f_1(e) \parallel f_2(e) \) safe?
Concurrency and Code Reuse

class List<T>
    var first : Link<T>

    def add(elem : T) : void
        ...

    def remove(i : int) : T
        ...

    def lookup(i : int) : T
        ...

    ...

class SynchronizedList<T>
    var list : List<T>

    def add(elem : T) : void
        lock();
        this.list.add(elem);
        unlock();

    def remove(i : int) : T
        lock();
        tmp = this.list.remove(i);
        unlock();
        return tmp;

    def lookup(i : int) : T
        lock();
        tmp = this.list.remove(i);
        unlock();
        return tmp;
Summary of Motivation

- Code that needs concurrency control is indistinguishable from code that does not.
  Correct synchronisation warrants program-wide aliasing analysis.

- Concurrency control varies across different usage scenarios.
  Building concurrency control into a data-structure generates overhead.

- Business logic and concurrency control are often orthogonal concerns.
  ```
  lock(); ...; unlock();
  ```
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++;
```

- 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

- If included in a class with a field

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

  ...I will provide a method `inc()`
Classes are Composed by Capabilities

```csharp
class Counter = Inc ⊕ Get {
    var cnt : int
}
```
Aliasing and Concurrency Control (revisited)

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

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

```
class SharedCounter = locked Inc ⊕ read Get

May not alias!
Properly synchronised!
```

Implemented by a readers-writer lock
Aliasing and Concurrency Control (revisited)

```java
class LocalCounter = thread Inc ⊕
assert c1.value() == 42;
c1.inc();
assert c1.value() == 43;
```

Also in the paper!

Implemented by a readers-writer lock
Composite Capabilities

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

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

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

```rust
class Pair = linear Fst \times linear Snd {
  var fst : int
  var snd : int
}
```

- A capability *conjunction* $A \otimes B$ can be used as $A$ and $B$, possibly in parallel.
### 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
```
let p = new Pair();
let f, s = consume p;
finish{
  async{f.set(x)}
  async{s.set(y)}
}
p = consume f + consume s
let p = new Pair();
let f, s = consume p;
finish{
    async{f.set(x)}
    async{s.set(y)}
}
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
```

Also in the paper!
Aliasing and Parallelism (revisited)

\[ \text{class Tree = linear Left \otimes linear Right \otimes ...} \]
Aliasing and Parallelism (revisited)

class Tree = linear Left ⊗ linear Right ⊗ ...

No aliasing nodes

No unsafely aliased elements

No unsafe sharing between elements
### Aliasing and Parallelism (revisited)

**class** Tree = **linear** Left ⊗ **linear** Right ⊗ ...  
(borrowing)

```haskell
def foreach(t : S(Tree<T>), f : T -> T) : void
let l : S(linear Left<T>)
    , r : S(linear Right<T>)
    , e : S(linear Element<T>) = consume t;
finish {
    async { foreach(l.getLeft(), f) }
    async { foreach(r.getRight(), f) }
    e.apply(f);
}
```

No unsafe sharing between elements
Concurrency and Code Reuse (revisited)

- 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 {
    ...
  }
}
```

Can assume exclusive access

```scala
class List[T] = thread Add[T] ⊕ ...
  var first : Link<T>
}
```

Annotations in type declarations only

```scala
class SynchronizedList[T] = locked Add[T] ⊕ ...
  var first : Link<T>
}
```

No effect tracking or ownership types
Meta-Theoretic Results

- Type system formalised on top of a simple object-oriented language
  - Finish-Async style parallelism
  - Reentrant readers-writer locks

- Key result: **Well-formed programs preserve safe aliasing**
  - Any two aliases are e.g. composable, synchronised, thread-local, subordinate
  - No data-races

![Diagram of well-formed programs preserving safe aliasing](image)
Current and Future Work

- Implementation underway in the context of Encore

Reference capabilities for safe sharing between active objects
Current and Future Work

- Implementation underway in the context of Encore

  Reference capabilities for safe sharing between active objects

```java
def foreach(t : S(Tree<T>), f : T -> T) : void
let l : S(linear Left<T>)
  , r : S(linear Right<T>)
  , e : S(linear Element<T>) = consume t;
finish {
  async { foreach(l.getLeft(), f) }
  async { foreach(r.getRight(), f) }
  e.apply(f);
}
```
Current and Future Work

- Implementation underway in the context of Encore

  Reference capabilities for safe sharing between active objects

```python
def foreach(t : S(Tree<T>), f : T -> T) : void
let l, r, e = consume t;
finish {
  async { foreach(l.getLeft(), f) }
  async { foreach(r.getRight(), f) }
  e.apply(f);
}
```
Current and Future Work

- Implementation underway in the context of Encore

Reference capabilities for safe sharing between active objects

```scala
def foreach(t : S(Tree<T>), f : T -> T) : void
finish {
    async { foreach(t.getLeft(), f) }
    async { foreach(t.getRight(), f) }
    t.apply(f);
}
```
A Hierarchy of Capabilities

- Exclusive
  - Linear
  - Thread
- Shared
- Safe
- Optimistic
  - Atomic
  - Lock-Free
- Pessimistic
  - Locked
  - Active
- Subordinate
  - Unsafe
- Oblivious
  - Read
  - Immutable
A Hierarchy of Capabilities

Safe
Reference Capabilities Address Many Concerns

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? ✓

List l = ...;
l.add(x);

More examples, full proofs etc. in the technical report!