VIRTUSLAB

Introduction to Scala 3's Capture Checking and Separation Checking

One of the trickiest parts of writing large-scale software is dealing with mutable data. For instance, data might be mutated in unexpected places, or resources might be used at the wrong timing.

Languages like Rust mitigate these problems through ownership and lifetimes. But how do we bring these ideas into a GC-based language like Scala in a way that doesn't break existing programs? In other words, we want to track access rights (capabilities) to resources (objects in the object-capability model), while leaving memory management to the GC.

Scala 3's answer is Capture Checking + Separation Checking.

Capture Checking

What is a "Capture"?

Before explaining Capture Checking, let's start with what "capture" means.

This refers to closure capture. In the following program, the closure `increment` references `c`, which is defined outside the closure. We say that `increment` *captures* `c`.

None

```
//> using scala 3.8.3

class Counter:

  def inc(): Counter = ???


@main def main() =

  val c: Counter = new Counter

  val increment = () => {
```

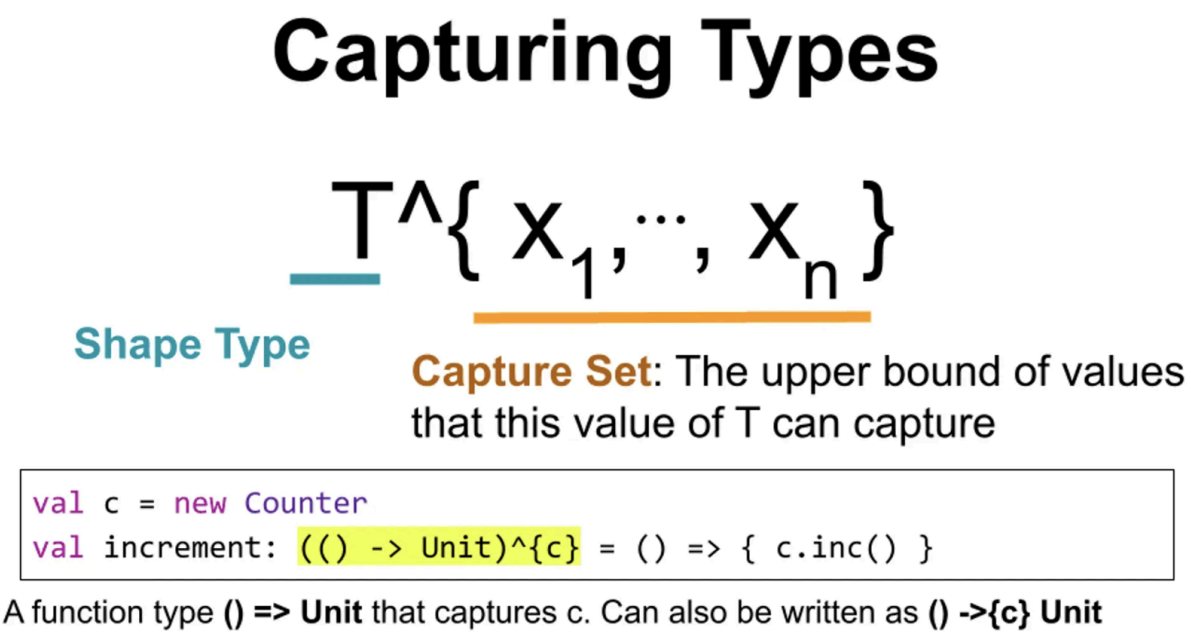

```
  c.inc()
}

increment()
```

Capturing Types

Capture Checking introduces Capturing Types to track variable captures at the type level.

```
T^{x_1, ..., x_n}
```

- `T`: the shape type — a value of this type captures the values in the following capture set
- `{x_1, ... x_n}`: the capture set — the set of values this value is allowed to capture



For example, in the code above, `increment` has the following capturing type.

```
None
//> using scala 3.8.3
```

```
import language.experimental.captureChecking

class Counter:

  def inc(): Counter = ???


@main def main() =

  val c: Counter^ = new Counter

  val increment: (() -> Unit)^{c} = () => {

    c.inc()

  }

  increment()
```

`Counter^` roughly means "`c` is a value tracked by capture checking."

And, you might wonder why `() -> Unit` instead of `() => Unit`. The `->` notation was introduced with capture checking.

- The traditional function type `A => B` represents a function that may capture arbitrary values.
- While the new `A -> B` represents a pure function that captures nothing.
- `A ->{c,d} B` is shorthand for `(A -> B)^{c,d}`, which means the function captures `c` and `d`.

See Function Types | Capture Checking for details.

(`T^` is shorthand for `T^{any}`, where `any` is a top capability representing a separate, exclusive capability.)

A value must have a non-empty capture set to be tracked by capture checking, so `any` is used to mark a value as being tracked. Although `c: Counter^ = new Counter` being able to "capture any capability" doesn't feel very intuitive, it is needed for the subtyping rules described later.

What Does Capture Checking Check?

Capture Checking verifies that closures only capture values within their declared capture set. For example, the following program produces an error.

(The capture set of `increment` is `^{c1}`, but the closure body also captures `c2`, which is not in the set)

None

```
//> using scala 3.8.3

import language.experimental.captureChecking

// ...

@main def main() =

  val c1: Counter^ = new Counter

  val c2: Counter^ = new Counter

  val increment: (() -> Unit)^{c1} = () => {

    c1.inc()

    c2.inc() // (c2: Counter^) cannot be referenced here

  }
```

None

```
[error] ./cc.scala:9:38

[error] Found:    () ->{c1, c2} Unit

[error] Required: () ->{c1} Unit

[error]

[error] Note that capability `c2` cannot flow into capture set {c1}.
```


This lets us express the constraint that `increment` can only operate on `c1` at the type level. Seen this way, the capture set (`^{c1}`) can be thought of as the set of capabilities that the closure is allowed to access.

(Subtyping is extended by capturing types: a type with a smaller capture set is a subtype. For example, `T^{} <: T^{c1} <: T^{c1,c2} <: T^{any}`. So assigning a value of type `() ->{c1,c2} Unit` to `() ->{c1} Unit` would be a type error.)

Example: Referentially Transparent Closures

As a simple use case, let's implement a `map` that guarantees referential transparency — we want to ensure that the function passed to `map` doesn't perform any destructive mutations.

Java

```
//> using scala 3.8.3

import language.experimental.captureChecking


enum MyList[+A]:

  case Nil

  case Cons(head: A, tail: MyList[A])

  // A ->{} B is shorthand for (A -> B)^{}, a function that captures
nothing

  def map[B](f: A ->{} B): MyList[B] = this match

    case Nil => Nil

    case Cons(h, t) => Cons(f(h), t.map(f))


@main def main() =

  var foo: Integer^ = 0
```


```
MyList.Cons(1, MyList.Nil).map { x =>
  foo = foo + x // Error: foo is not in f's capture set
  x
}
```

This prevents the function passed to `map` from accessing external mutable state.

Other Examples

The reference documentation includes several more use cases. By treating values tracked by capture checking as capabilities, you can do things like effect tracking:

- Escape Checking
- Checked Exceptions

(This is the most commonly shown example of capture checking, but it's not the most intuitive one.)

Alias Tracking

You might wonder whether aliases confuse capture checking, but they are tracked as well:

```
None
val c: Counter^ = new Counter
val a: Counter^{c} = c // the type tells us that a has access to c
val increment: (() -> Unit)^{c} = () => {
  a.inc()
}
```

Separation Checking

Capture Checking is an experimental feature that is getting closer to be stabilized. Separation Checking, on the other hand, is another experimental feature based on Capture Checking.

Capture Checking only tracks whether a value can be accessed. Separation Checking introduces `SharedCapability` (read-only) and `ExclusiveCapability` (writable) to track *which parts of the program can mutate data*. This helps prevent data races in concurrent programs and enables separate tracking of read-only vs. writable effects.

Mutable extends ExclusiveCapability

Separation Checking provides `caps.Mutable`. Classes that extend it can define methods with the `update` modifier. An `update def` indicates that the method mutates the class's state (or external resources, whatever), and these methods cannot be called in a read-only context.

None

```
//> using scala 3.8.3

import scala.language.experimental.captureChecking

import scala.language.experimental.separationChecking


class Ref(init: Int) extends caps.Mutable:

  private var current = init

  def get: Int = current

  update def set(x: Int): Unit = {

    current = x

  }
```

Here, `any` (`caps.any`) is a top capability, essentially the same as the universal capability (`cap`) from capture checking, but in separation checking each occurrence of `any` is treated as distinct and exclusive. `any.rd` is the read-only version of `any`: it grants read access but prohibits mutation (i.e. calling `update` methods).

[^any-separate]: Each occurrence of `any` is treated as a separate, exclusive capability. For example, in `def swap(a: Ref^, b: Ref^), a` gets $\texttt{Ref\^\{any}_1\texttt{\}}$ and `b` gets $\texttt{Ref\^\{any}_2\texttt{\}}$ — two distinct capabilities, so the compiler knows they don't alias each other.

When writing parameter types, `Ref` expands to `Ref^{any.rd}`(read-only access). `Ref^` expands to `Ref^{any}` (full access including mutation).

```
None
// Ref = Ref^{any.rd}: read-only access

def read(r: Ref): Int =

  r.get


// Ref^ = Ref^{any}: full access including mutation

def update(r: Ref^, value: Int): Unit =

  r.set(value)


@main def main() =

  val r = Ref(0)

  update(r, 4)

  println(read(r)) // 4
```

Calling an update method inside `read` is not allowed:

```
None
def read(r: Ref^{any.rd}): Int =
```

```
  // Cannot call update method set of r since its capture set {r} is
read-only

  r.set(0)

  r.get
```

The Separation Check

Consider a `par` function that executes two functions in parallel. If one argument calls an update method on a resource, the other argument must not access that resource:

```
None
def par[A, B](f: () => A, g: () => B): (A, B) = ???
```

```
None
@main def main() =

  val r = Ref(0)

  // Separation failure

  par(() => r.set(2), () => r.set(3))

  par(() => r.get, () => r.set(3))

  par(() => r.set(2), () => r.get)

  // This is OK - both are read-only

  par(() => r.get, () => r.get)
```

This constraint might seem overly strict. For a `seq` function that runs functions sequentially, such restrictions shouldn't be necessary. This is handled through a mechanism called *Hide*, which I'll explain next.

```
None
def seq(f: () => Unit, g: () => Unit): Unit = { f(); g() }

@main def main() =

  val r = Ref(0)

  // Separation failure: argument of type  () ->{r} Unit

  // to method seq: (f: () => Unit, g: () => Unit): Unit

  // corresponds to capture-polymorphic formal parameter f of type  () =>
Unit

  // and hides capabilities  {r}.

  seq(() => r.set(1), () => r.set(2))
```

Move Semantics

In Separation Checking, `T^` (`T^{any}`) represents a special type: an independent capability shared with no one.

Assigning a variable `y` to `T^` means that `y` is *hidden* by the exclusive capability `x`. As long as `x` is alive, the original `y` cannot be accessed (otherwise `x` and `y` would share the same capability).

This prevents unintended mutation through aliased references:

```
None
@main def main() =

  val y = Ref(0)

  update(y, 4) // ok


  locally {

    val x: Ref^ = y // y is hidden by x

    // Separation failure: Illegal access to {y}  which is hidden by the
previous definitio
```

```
    // of value x with type Ref^
    update(y, 4) // Error
  }

  // With an explicit capture set, hiding does not occur
  locally {
    val x: Ref^{y} = y
    update(y, 4) // OK
  }
```

(Recall the `par` definition — its arguments also have the top capability (`any`))

```
None
// () => A is shorthand for (() -> A)^{any}
def par[A, B](f: () => A, g: () => B): (A, B) = ???
```

- When calling `par(() => r.set(1), () => r.set(2))`:
  - `() => r.set(1)` is hidden by `f` (transitively, `r` captured by this closure is also hidden by `f`)
  - `() => r.set(2)` tries to access `r`, but it's hidden by `f` — so this fails

This can be resolved by explicitly granting `g` access to the capabilities hidden by `f`:

```
None
def seq(f: () => Unit, g: () ->{any, f} Unit): Unit = f(); g()
// def seq(f: () => Unit; g: () ->{f} Unit) also compiles,
// but it won't compile if g captures more than f does.

// OK
seq(() => r.set(1), () => r.set(2))
```

(Previously called `cap` (the "universal capability"), now renamed to `any` and treated as a top capability in separation checking.)

Consume Parameters

You might think you could bypass hiding by returning a `Ref^` from a function. Consider an in-place update function:

```
None
def incr(a: Ref^): Ref^ =
  a.set(a.get + 1)
  a


@main def main() =
  val a = new Ref(1)
  val b: Ref^ = incr(a) // b is actually an alias of a, but isn't hidden
  incr(a) // so mutating a also mutates b
  b.get // oops it's 3, but expected 2
```

Returning a value with the top capability from a function is only safe if the parameter is no longer used (since the function might create and return an alias). So the `incr` definition above actually produces an error:

```
None
Separation failure: method incr's result type Ref^ hides parameter a.

The parameter needs to be annotated with consume to allow this.
```

To compile this, the parameter must be marked with `consume`, explicitly indicating that the argument will be hidden (move semantics!):

```
None
def incr(consume a: Ref^): Ref^ =
```


```
  a.set(a.get + 1)
  a


@main def main() =
  val a = new Ref(1)
  val b: Ref^ = incr(a) // a is consumed (hidden by b)
  incr(a) // Error
```

Summary

Both Capture Checking and Separation Checking are still under active development, but they point toward a future where you can write Scala with GC-managed memory while selectively opting into Rust-like constraints where it matters. The best of both worlds.

References

- Capture Checking - Bringing Effect Checking to the Masses — project page with links to papers
- Capture Checking | Scala 3 Reference
- Separation Checking | Scala 3 Reference
- System Capybara: Capture Tracking for Ownership and Borrowing | ICFP/SPLASH'25