CAPTURE CHECKING

Oliver Bračevac
EPFL/LAMP
Scala Days 2025

THE TRY-WITH-RESOURCES PATTERN

Can we make it statically safe?

```
trait File:
  def read(): Byte
  def write(b: Byte): Unit
  def close(): Unit


def withFile[T](name: String)
               (op: File ⇒ T): T =
  val f = File(name)
  val r = op(f)
  f.close()
  r
```

```
withFile("example.txt"): file ⇒
  file.write(42)
  val b = file.read()
  println(b)
  b ✅

val leak = withFile("example.txt"): file ⇒
  () ⇒ file.read()

leak() // ← Crash! `file` is already closed
```

☠️

THE TRY-WITH-RESOURCES PATTERN

Can we make it statically safe? Yes!

```
import language.experimental.captureChecking
```

THE TRY-WITH-RESOURCES PATTERN

Can we make it statically safe? Yes!

```
import language.experimental.captureChecking

trait File extends caps.SharedCapability:
  def read(): Byte
  def write(b: Byte): Unit
  def close(): Unit


def withFile[T](name: String)
               (op: File^ ⇒ T): T =
  val f = File(name)
  val r = op(f)
  f.close()
  r
```

```
withFile("example.txt"): file ⇒
  file.write(42)
  val b = file.read()
  println(b)
  b ✅

val leak = withFile("example.txt"): file ⇒
  () ⇒ file.read()

leak() // type error!
```

THE TRY-WITH-RESOURCES PATTERN

Can we make it statically safe? Yes!

```
-- [E007] Type Mismatch Error: local/scaladays.scala:58:40 ----------------------
58 |    val leak = withFile("example.txt"): file =>
   |                                        ^
   |Found:    (file: Safe.File^'s1) ->'s2 () ->{file} Byte
   |Required: Safe.File^ => () ->'s3 Byte
   |
   |where:    => refers to a fresh root capability created in value leak when checking argument to parameter op of method withFile
   |          ^  refers to the universal root capability
   |
   |Note that capability file cannot be included in outer capture set 's3.
59 |      () => file.read()
   |
   | longer explanation available when compiling with `-explain`
1 error found
```

CAPTURING TYPES

What's captured by a value?

```
val out: IO^      // tracked
val file: File^   // tracked
val i: Int        // untracked
```

```
() ⇒ file.read()                : () →{file} Byte
() ⇒ out.print(file.read())     : () →{file, out} Unit
(f: File^) ⇒ f.read()           : File^ →{} Byte = File^ → Byte
() ⇒ i.toString()               : () → String
```

```
trait File extends caps.SharedCapability:
    def read(): Byte
    // ...

trait IO extends caps.SharedCapability:
    def print(b: String): Unit
    // ...
```

Key property: Tracked values can never become untracked!

CAPTURING TYPES `T^{c1, …, cn}`

Notations

```
import scala.caps.cap // ← The root capability

// Impure types:

File^    = File^{cap}

() →{file, out} Unit    = (() → Unit)^{file, out}

T ⇒ U   = T →{cap} U    = (T → U)^{cap}  // “Any effects”

() ?→{file, out} Unit   = (() ?→ Unit)^{file, out}

T ?=> U  = T ?→{cap} U  = (T ?→ U)^{cap}

// Pure types:

U^{} = U    T → U   T ?→ U   Int, String, Boolean, …
```

CAPTURING TYPES T^{c1, …, cn}

Notations

```
import scala.caps.cap // ← The root capability

// Impure types:

File^    = File^{cap}    = File // ← only if it extends caps.Capability

() →{file, out} Unit    = (() → Unit)^{file, out}

T ⇒ U   = T →{cap} U    = (T → U)^{cap}  // “Any effects”

() ?→{file, out} Unit   = (() ?→ Unit)^{file, out}

T ?=> U  = T ?→{cap} U  = (T ?→ U)^{cap}

// Pure types:

U^{} = U    T → U   T ?→ U   Int, String, Boolean, …
```

SUBCAPTURING 101

```
trait Logger:
  def log(msg: String): Unit


class FileLogger(using File^) extends Logger:
  def log(msg: String) =
    summon[File].write(msg)

class NullLogger(@constructorOnly using f: File^) extends Logger:
  def log(msg: String) = ()
  // val myFile = f ← illegal
```

SUBCAPTURING 101

Capturing is Contagious

SUBCAPTURING 101

Capturing is Contagious

```
val file: File^ = …

val file2: File^ = …

val impure: Logger^{file} = FileLogger(using file)

val pure: Logger^{} = NullLogger(using file)

val sock: Socket^ = …

val l: Logger^{file, sock} =

  new Logger(using file) { def log(msg: String) = sock.send(msg) }

val l2: Logger^{file2} = FileLogger(using file2)

val fun = (s: String) ⇒ l.log(s)
```

SUBCAPTURING 101

$$T\text{^}C_1 \mathrel{<:} U\text{^}C_2$$

if $T \mathrel{<:} U$ and $C_1 \mathrel{<:} C_2$

```
val file:   File^{cap}
val file2:  File^{cap}
val impure: Logger^{file}
val pure:   Logger^{}
val sock:   Socket^{cap}
val l:      Logger^{sock, file}
val l2:     Logger^{file2}
val fun:    (String → Unit)^{l}
```

SUBCAPTURING 101

$$T\text{^}C_1 \mathrel{<:} U\text{^}C_2 \quad \text{if} \quad T \mathrel{<:} U \quad \text{and} \quad C_1 \mathrel{<:} C_2$$

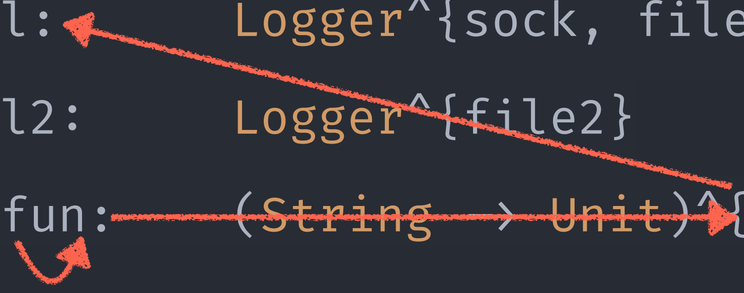
```
val file:   File^{cap}
val file2:  File^{cap}
val impure: Logger^{file}
val pure:   Logger^{}
val sock:   Socket^{cap}
val l:      Logger^{sock, file}
val l2:     Logger^{file2}
val fun:    (String → Unit)^{l}
```

```
fun: (String → Unit)^{fun}
```

SUBCAPTURING 101

$$T\text{^}C_1 \;<:\; U\text{^}C_2 \quad \text{if} \quad T <: U \;\text{ and }\; C_1 <: C_2$$

```
val file:    File^{cap}
val file2:   File^{cap}
val impure:  Logger^{file}
val pure:    Logger^{}
val sock:    Socket^{cap}
val l:       Logger^{sock, file}
val l2:      Logger^{file2}
val fun:     (String -> Unit)^{l}
```

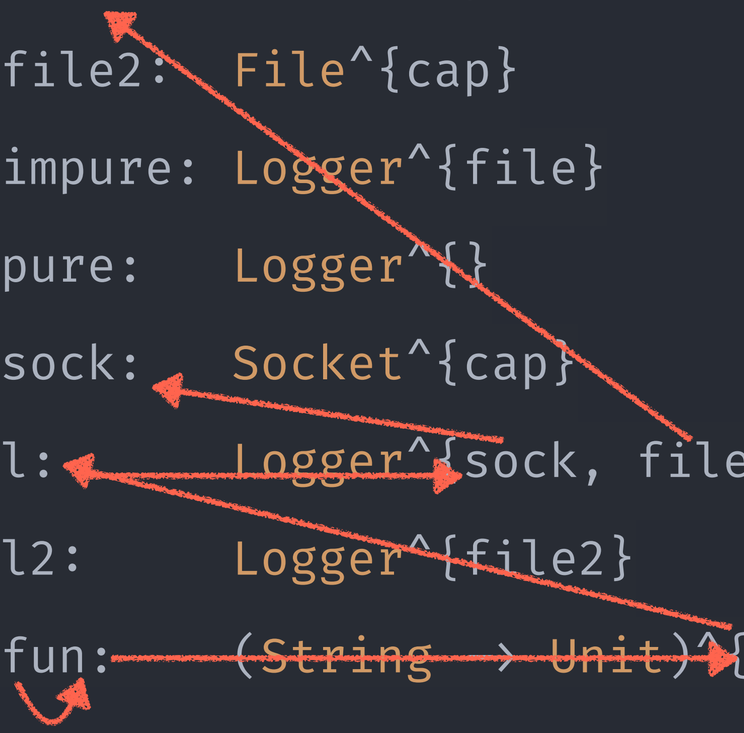


```
fun: (String -> Unit)^{fun}
  <: (String -> Unit)^{l}
```

SUBCAPTURING 101

$$T\text{^}C_1 <: U\text{^}C_2 \quad \text{if} \quad T <: U \quad \text{and} \quad C_1 <: C_2$$

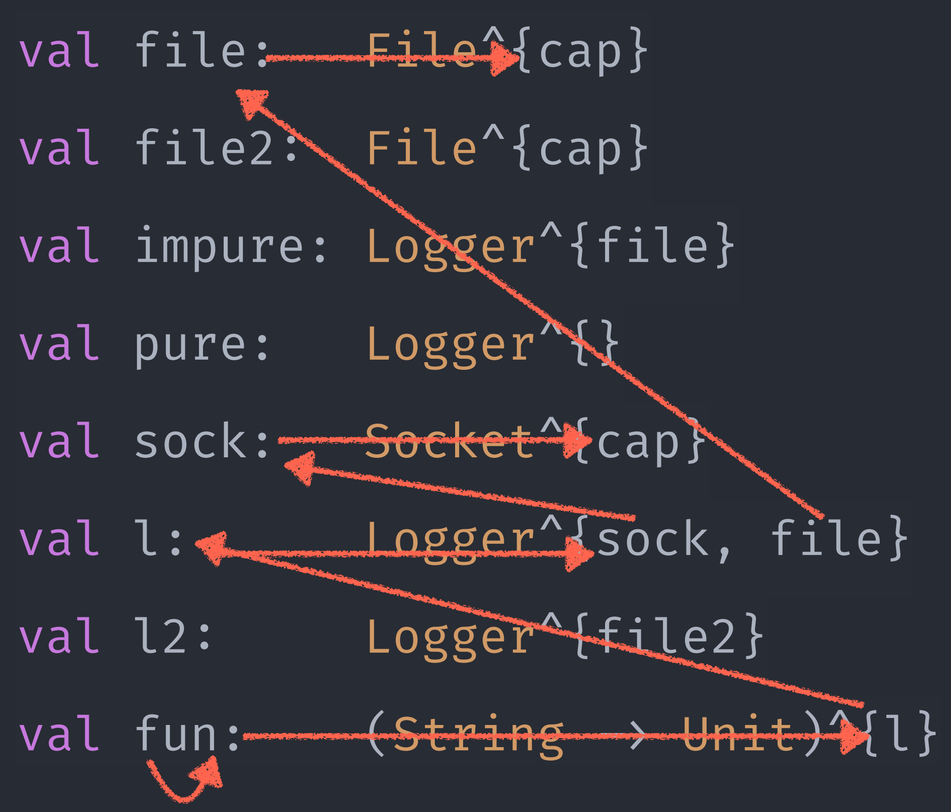
```
val file:    File^{cap}
val file2:   File^{cap}
val impure:  Logger^{file}
val pure:    Logger^{}
val sock:    Socket^{cap}
val l:       Logger^{sock, file}
val l2:      Logger^{file2}
val fun:     (String → Unit)^{l}
```



```
fun: (String → Unit)^{fun}
  <: (String → Unit)^{l}
  <: (String → Unit)^{file, sock}
```

SUBCAPTURING 101

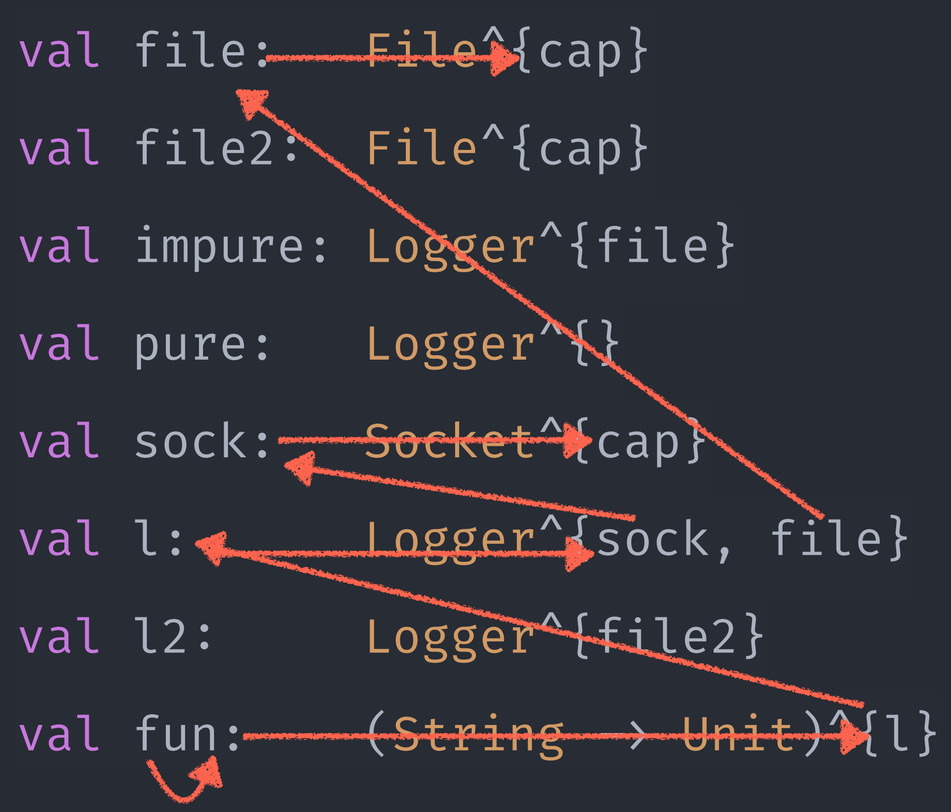
$$T\text{^}C_1 <: U\text{^}C_2 \quad \text{if} \quad T <: U \quad \text{and} \quad C_1 <: C_2$$




```
fun: (String → Unit)^{fun}
  <: (String → Unit)^{l}
  <: (String → Unit)^{file, sock}
  <: (String → Unit)^{cap}
```

SUBCAPTURING 101

$$T\text{^}C_1 <: U\text{^}C_2 \quad \text{if} \quad T <: U \quad \text{and} \quad C_1 <: C_2$$



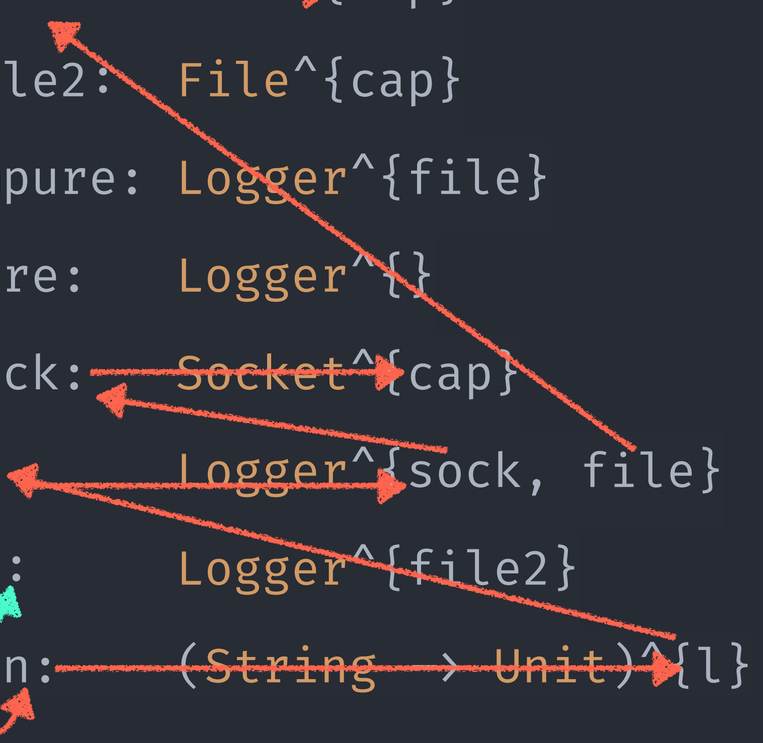

```
fun: (String → Unit)^{fun}
  <: (String → Unit)^{l}
  <: (String → Unit)^{file, sock}
  <: (String → Unit)^{cap}
```

SUBCAPTURING 101

$$T\text{^}C_1 <: U\text{^}C_2$$

if $T <: U$ and $C_1 <: C_2$

```
val file:    File^{cap}
val file2:   File^{cap}
val impure:  Logger^{file}
val pure:    Logger^{}
val sock:    Socket^{cap}
val l:       Logger^{sock, file}
val l2:      Logger^{file2}
val fun:     (String -> Unit)^{l}
```

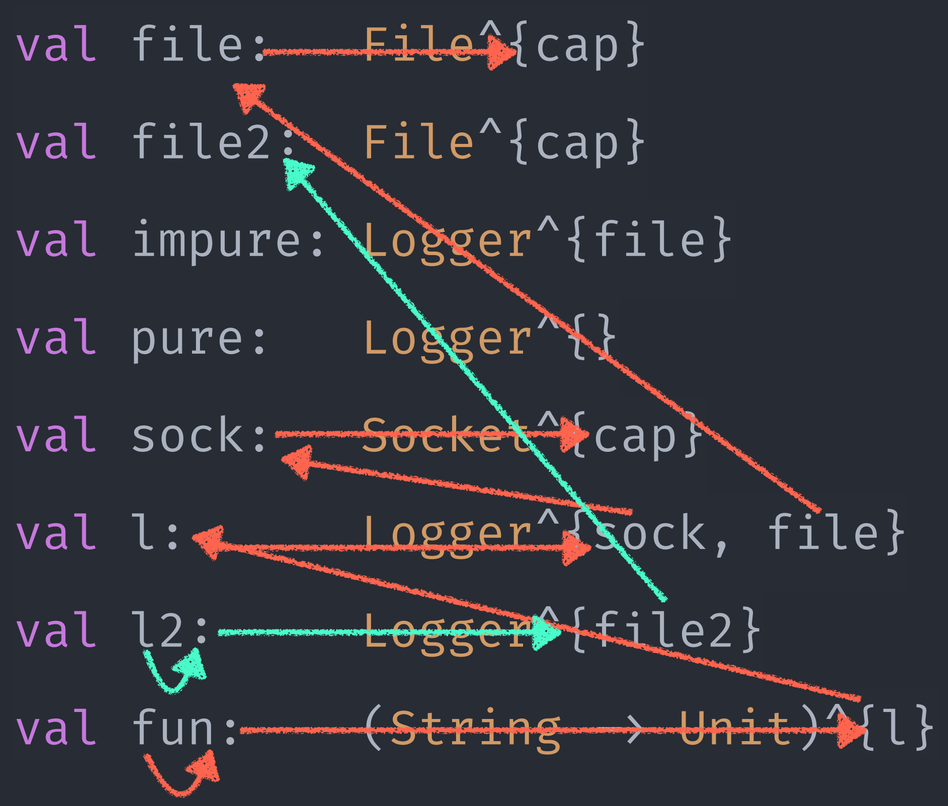


```
fun: (String → Unit)^{fun}
  <: (String → Unit)^{l}
  <: (String → Unit)^{file, sock}
  <: (String → Unit)^{cap}

l2 : Logger^{l2}
```

SUBCAPTURING 101

$$T\text{^}C_1 <: U\text{^}C_2$$

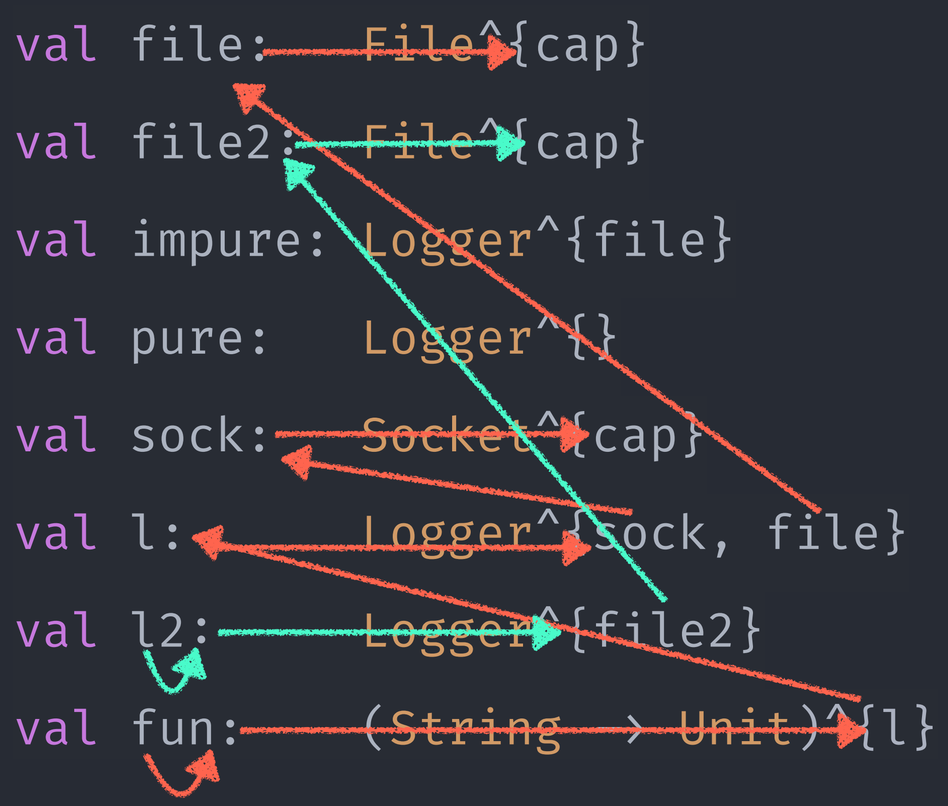
if $T <: U$ and $C_1 <: C_2$




```
fun: (String → Unit)^{fun}
  <: (String → Unit)^{l}
  <: (String → Unit)^{file, sock}
  <: (String → Unit)^{cap}

l2 : Logger^{l2}
  <: Logger^{file2}
```

SUBCAPTURING 101

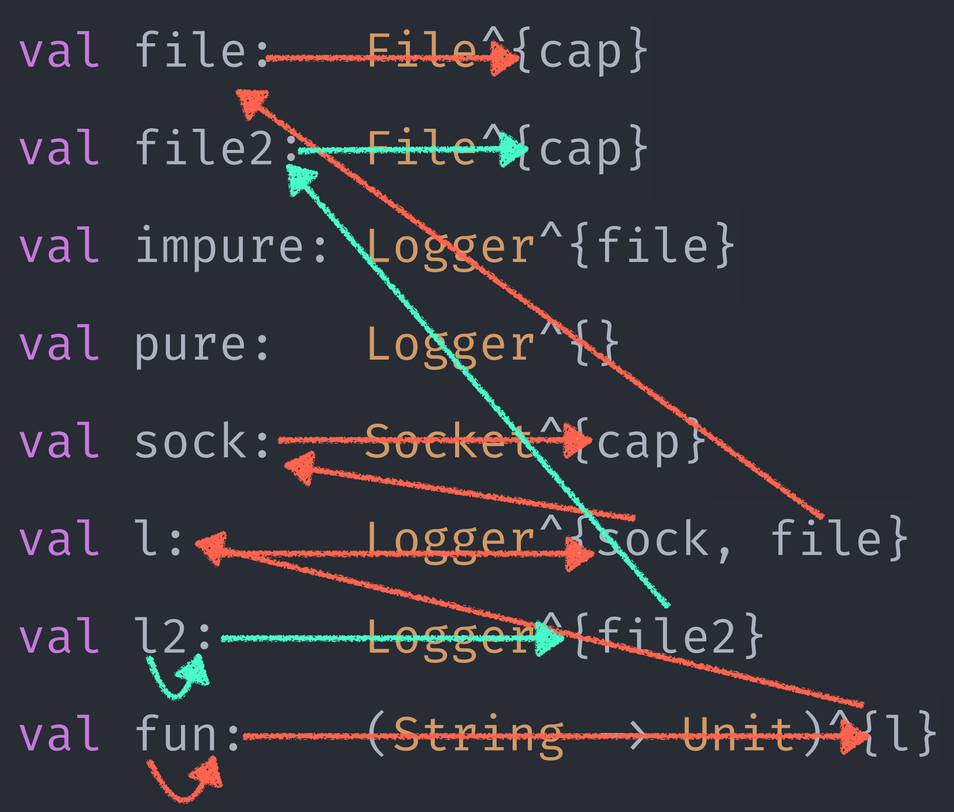
$$T\text{^}C_1 <: U\text{^}C_2 \quad \text{if} \quad T <: U \quad \text{and} \quad C_1 <: C_2$$




```
fun: (String → Unit)^{fun}
  <: (String → Unit)^{l}
  <: (String → Unit)^{file, sock}
  <: (String → Unit)^{cap}

l2 : Logger^{l2}
  <: Logger^{file2}
  <: Logger^{cap}
```

SUBCAPTURING 101

$$T\text{^}C_1 \;<:\; U\text{^}C_2 \quad \text{if} \quad T <: U \quad \text{and} \quad C_1 <: C_2$$




```
fun: (String → Unit)^{fun}
  <: (String → Unit)^{l}
  <: (String → Unit)^{file, sock}
  <: (String → Unit)^{cap}

l2 : Logger^{l2}
  <: Logger^{file2}
  <: Logger^{cap}
```

Note that fun and l2 are separate!

`{fun, l, file, sock}` `{l2, file2}`

IMPLICIT CAPTURE POLYMORPHISM

Means Implicit Effect Polymorphism (Effects-as-Capabilities Model)

Before Capture Checking:

```
trait List[+T]:
  def map[U](f: T ⇒ U): List[U]
```

Is After Capture Checking :)

```
trait List[+T]:
  def map[U](f: T ⇒ U): List[U]
```

T →{cap} U
Good old
subtype
polymorphism!

Examples:

```
val l = List(1, 2, 3)

l.map {x ⇒ log(x); x}              : List[Int]

() ⇒ l.map {x ⇒ log(x); x}         : () →{log} List[Int]

() ⇒ l.map {x ⇒ x * x}             : () → List[Int]

() ⇒ l.map {x ⇒ Future(x * x)}     : () →{async} Future[Int]
```

EAGER VS. LAZY COLLECTIONS

```
trait List[+A]:

  def map[B](f: A ⇒ B): List[B]

  def filter(p: A ⇒ Boolean): List[A]

  def concat(ys: List[A]): List[A]

  def drop(n: Int): List[A]
```

```
trait LazyList[+A]:

  def map[B](f: A ⇒ B): LazyList[B]^{this, f}

  def filter(p: A ⇒ Boolean): LazyList[A]^{this, p}

  def concat(ys: LazyList[A]^): LazyList[A]^{this, ys}

  def drop(n: Int): LazyList[A]^{this}
```

EXPLICIT CAPTURE POLYMORPHISM

Example: Serializing Futures

```
def collect[T, C^](fs: Set[Future[T]^{C}]): Stream[Future[T]^{C}] =
                                        /* ^ Typically needed when functions
  val channel = Channel()                  transform generic containers
                                           of capturing types */
  fs.forEach(_.onComplete(v ⇒ channel.send(v)))

  Stream.of(channel)

def collect[Int, {io, log}]  // : Set[Future[T]^{io, log}]
                                      → Stream[Future[T]^{io, log}]
```

EXPLICIT CAPTURE POLYMORPHISM

Example: Composing Futures in Direct Style

```
extension [T, C^](xs: Seq[ Future[T]^{C} ])
  def all(using ec: Async^): Future[Seq[T]]^{ec, C}

Async: async ?=>
  withIO: io ⇒
    withThrow: throw ⇒
      val futs =
        val f1 = Future(useIO())    // : Future[Int]^{io,async}
        val f2 = Future(useThrow()) // : Future[Int]^{throw,async}
        Seq(f1, f2)                 // : Seq[Future[Int]^{io,throw,async}]
      val join = futs.all           // : Future[Seq[Int]]^{io,throw,async}
      await(join)                   // : Seq[Int]
```

PATH-DEPENDENT CAPTURES

Example: Safe Lexically Delimited Continuations

```
trait Label extends SharedCapability:
  type Fv^ // ← Free variables of the boundary delimiter

def boundary[T, C^](block: Label{type Fv = {C}} →{C} T): T
                                    // ^      ^ Sync free variables

def suspend[U](label: Label)(handler: () →{label.Fv} U): U
                                    //    ^  Can only mention label's free vars

val x = 1
  boundary: outer ⇒
    val y = 2
    boundary: inner ⇒
      suspend(outer): () ⇒
        println(inner) // error (would leak the inner label)
        x + y
```

SHALL I BE IMPLICIT OR EXPLICIT?

```
def apply(T ⇒ U): V
```

vs.

```
def apply[C^](T →{C} U): V
```

When porting the Scala standard library,

zero explicit capture parameters

were needed and

overall changes were minimal!

Type of Change	Added/Changed	Total in lib-cc	*Total in lib*
Capture sets on definitions	691 functions 91 classes	6189 functions 684 classes	*11113 functions* *1518 classes*
- Only universal captures	351		
- Only capture set on returns	379		
Capture sets on local variables	53	4583	*6262*
Restrict functions to pure	52	797	*1203*
Reach capabilities	19		
`@use` annotations	12		
Unsafe casts	5	810	*1325*
Unsafe capture set removal	25		
Class hierarchy changes	2		
- New classes	3		
Total lines	+1418/-1407 meaningful	52160 total 31395 code	*94635 total* *48210 code*

Porting the Scala standard collections lib
(Paper to appear at OOPSLA 2025)

What makes this possible? A mechanism called **boxing**.

BOXES

Enable Capability Tunneling

Recall the futures example:

```
Seq(f1, f2) : Seq[ Future[Int]^{io, throw, async} ]  // ← NO capture set ! Pure !
```

Instead of extra capture quantifiers, we embed the capturing types using box types:

```
Seq(f1, f2) : Seq[box[Future[Int]^{io, throw, async}]]
```

Boxes count as pure/effect free! Until they are accessed:

```
val $x = Seq(f1, f2).head : box[Future[Int]^{io, throw, async}]

⇝ unbox $x : Future[Int]^{io, throw, async}  // ← Unboxing by compiler
```

BOXES AND HATS

Prevent Capability Leaks

```
def try[A](f: CanThrow^ ⇒ A): A

try[CanThrow^](c ⇒ c)

⇝ val $b: box[CanThrow^] = try[box[CanThrow^]](c ⇒ c)
  unbox $b // error: cannot unbox ‘^’


try[() →{logger} Unit](c ⇒ () ⇒ logger.log("ok"))

⇝ val $b: box[() →{logger} Unit] = try[…](c ⇒ c)
  unbox $b // ok, capture set is known
```

Boxing is invisible to programmers, but can be printed with compiler options `-Vprint:cc -Ycc-verbose` for debugging purposes

SCALA STANDARD LIBRARY

- Collections: Fully capture checked (OOPSLA 2025 paper).
  - Boxes and cap avoid pollution with extra type variables.
  - Annotated capture sets provide concise & useful information.
- Remaining standard library: control and concurrency.
  - Solution: Classifier Capabilities, explained in the following.
  - As of today, merged and part of Scala 3.8 nightly :)

CLASSIFIER CAPABILITIES

Motivation: scala.util.Try

```
object Try:
  def apply[T](body: ⇒ T): Try[T]^{???}

val t = withFile { file ⇒
  boundary { label ?=>
    Try {
      file.write(42)
      break(using label); 0
    }
  }}
t.get
```

CLASSIFIER CAPABILITIES

Motivation: scala.util.Try

```
object Try:
  def apply[T](body: ⇒ T): Try[T]^{???}

val t = withFile { file ⇒
  boundary { label ?=>
    Try { // captures {file, label}
      file.write(42)
      break(using label); 0
    }
  }}
t.get
```

CLASSIFIER CAPABILITIES

Motivation: scala.util.Try

```
object Try:
  def apply[T](body: ⇒ T): Try[T]^{???}

val t = withFile { file ⇒
  boundary { label ?=>
    Try { // captures {file, label}
      file.write(42)
      break(using label); 0
    }
  }}
t.get // ← would rethrow label outside of boundary
```

CLASSIFIER CAPABILITIES

Motivation: scala.util.Try

```
object Try:
  def apply[T](body: ⇒ T): Try[T]^{???}

val t = withFile { file ⇒
  boundary { label ?=>
    Try { // captures {file, label}
      file.write(42)
      break(using label); 0
    } // : Try[Int]^{}
  }}
t.get // ← would rethrow label outside of boundary
```

CLASSIFIER CAPABILITIES

Motivation: scala.util.Try

```
object Try:
  def apply[T](body: ⇒ T): Try[T]^{body.only[Control]}

val t = withFile { file ⇒
  boundary { label ?=>
    Try { // captures {file, label}
      file.write(42)
      break(using label); 0
    } // : Try[Int]^{}
  }}
t.get // ← would rethrow label outside of boundary
```

CLASSIFIER CAPABILITIES

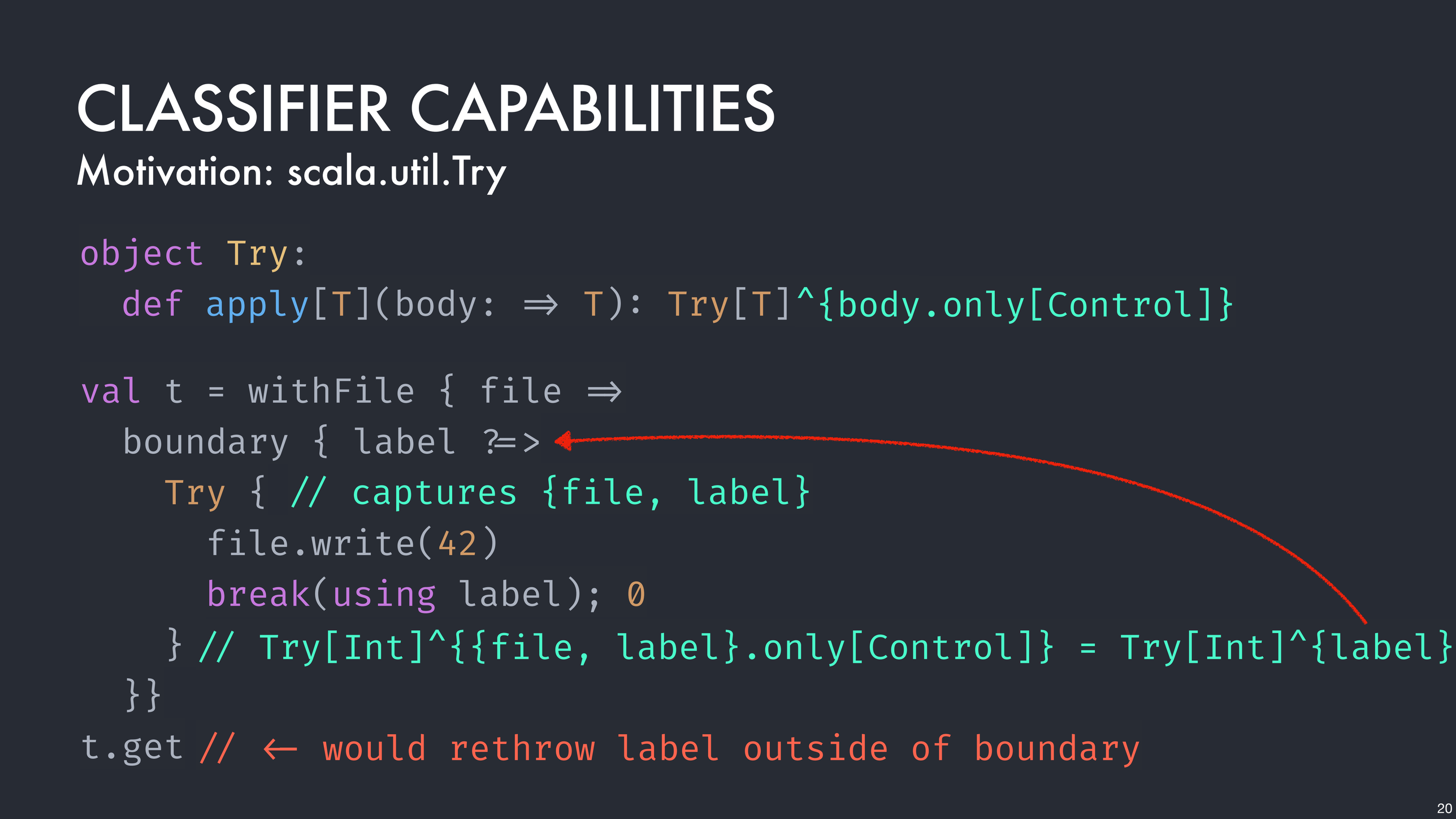
Motivation: scala.util.Try

```
object Try:
  def apply[T](body: ⇒ T): Try[T]^{body.only[Control]}

val t = withFile { file ⇒
  boundary { label ?=>
    Try { // captures {file, label}
      file.write(42)
      break(using label); 0
    } // Try[Int]^{{file, label}.only[Control]} = Try[Int]^{label}
  }}
t.get // ← would rethrow label outside of boundary
```

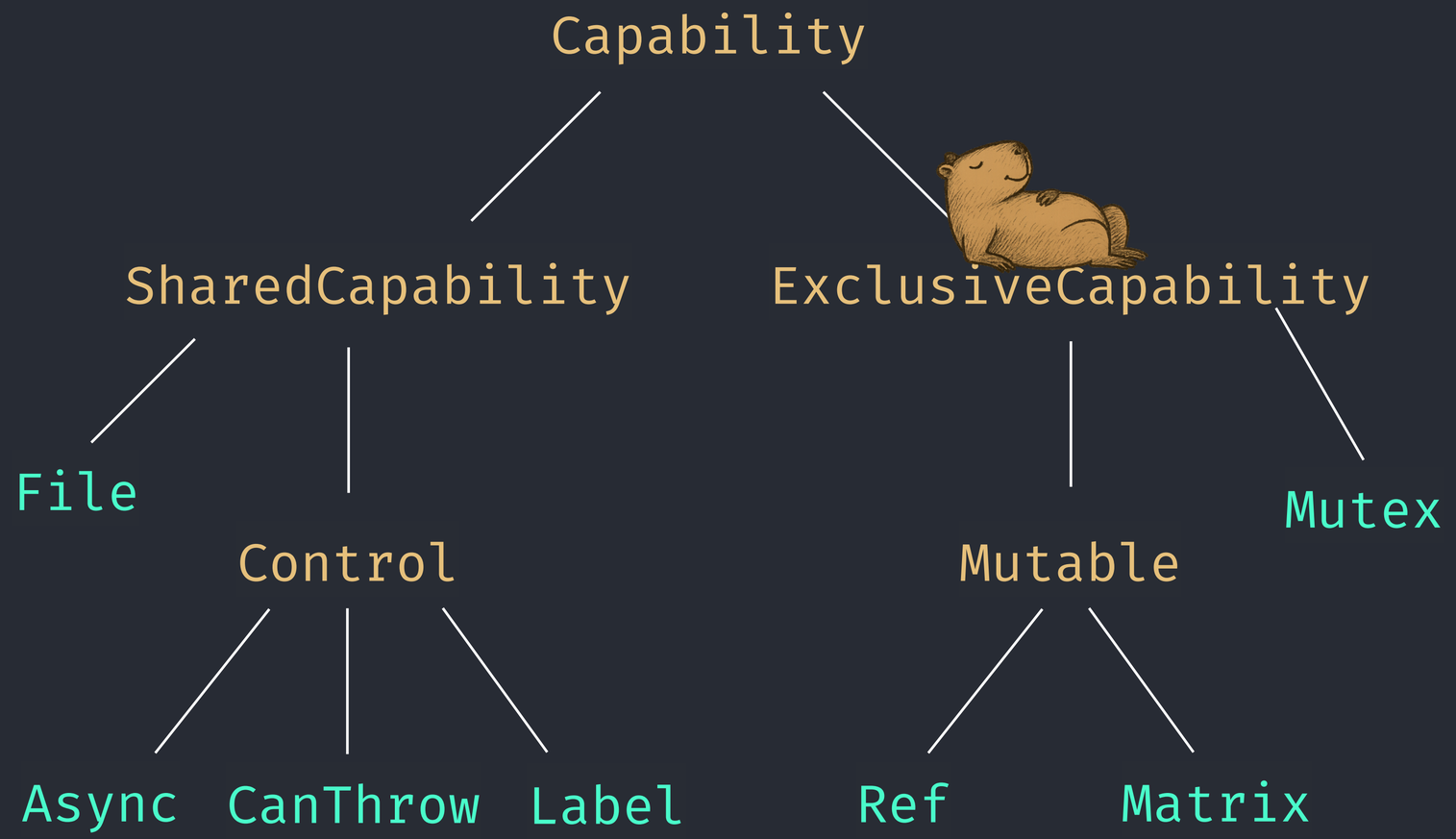
CLASSIFIER CAPABILITIES

Motivation: scala.util.Try

```
object Try:
  def apply[T](body: ⇒ T): Try[T]^{body.only[Control]}

val t = withFile { file ⇒
  boundary { label ?=>
    Try { // captures {file, label}
      file.write(42)
      break(using label); 0
    } // Try[Int]^{{file, label}.only[Control]} = Try[Int]^{label}
  }}
t.get // ← would rethrow label outside of boundary
```



CLASSIFIER CAPABILITIES



CLASSIFIER CAPABILITIES

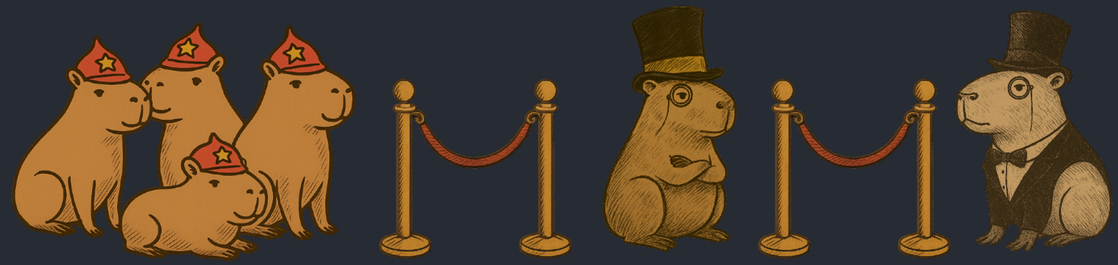
Advanced Example: Access Privilege

```
trait Read extends Classifier, SharedCapability
trait ReadWrite extends Classifier, SharedCapability

trait File(val name: String):
  erased val r: Read^
  erased val rw: ReadWrite^

  // Operations guarded by capability members:
  val read: () ->{r} Int
  val write: Int ->{rw} Unit

// Unrestricted use of files & other capabilities (as before)
def withFile[U](name: String)(block: File^ ⇒ U): U = …

// Only Read-classified allowed due to potential races
def parReduce[U](xs: Seq[U])(op: (U, U) ->{cap.only[Read]} U): U = xs.reduce(op)
```

```
withFile("foo.txt"): f ⇒
  f.read() // ok
  parReduce(1 to 1000): (a, b) ⇒
    a * b * f.read()    // ok
  parReduce(1 to 1000): (a, b) ⇒
    f.write(42)         // error
    a + b + f.read()    // ok
  f.write(42)           // ok
```

EXCLUSIVE CAPABILITIES

Sharing vs. Uniqueness



```
import language.experimental.captureChecking
import language.experimental.separationChecking

val m1, m2, m3: Matrix^

def mul(a: Matrix, b: Matrix, c: Matrix^): Unit
  // multiply a and b, storing the result in c


mul(m1, m2, m3) // ok
mul(m1, m1, m3) // ok: parameters a and b are read-only!
mul(m1, m1, m1) // error: separation failure


def mul(a: Matrix^{cap.rd}, b: Matrix^{cap.rd}, c: Matrix^{cap}): Unit
```

MUTABILITY TRACKING

```
class Ref(init: Int) extends Mutable:
  private var current = init
  def get: Int = current
  update def put(x: Int): Unit = current = x
```

```
class Matrix(nrows: Int, ncols: Int) extends Mutable:
  update def update(i: Int, j: Int, x: Double): Unit = ...
  def apply(i: Int, j: Int): Double = ...
```

CONSUME MODIFIER

Lightweight Ownership Transfer

```
trait Buffer[+T] extends Mutable:
  consume def append(x: T): Buffer[T]^

def concat[T](consume buf1: Buffer[T]^, buf2: Buffer[T]): Buffer[T]^


val a   = Buffer("a")
val b   = Buffer("b")
val a2  = a.append("x")
val b2  = b.append("y")
val ab  = concat(a2, b2)
val ab2 = concat(a, b) // error: a, b inaccessible
```

CAPTURE CHECKING

Resources

CC Language Reference & Scaladoc (!) of CC Standard Library
https://nightly.scala-lang.org/

Try CC + Standard Library out **today**
https://github.com/lampepfl/scala3-cc-template

ETA next week: Available in `scala-cli`

```
//> using scala 3.nightly
import language.experimental.captureChecking
```