Resources, lifecycles, and structured concurrency

Alexander Kuklev[1,2] ‹a@kuklev.com›

[1]Radboud University Nijmegen, Software Science
[2]JetBrains Research

Kotlin relies on scope-based resource management, but lacks mechanisms to enforce scope confinement, guarantee lifecycle safety, and rule out conflicting actions statically. We devise a mechanism addressing these issues in a manner compatible with structured concurrency and localized capturing in general. Our model can be seen as a generalization of Rust approach.

1 Introduction

We propose an extension to the Kotlin type system and flow-sensitive typing mechanism providing static control over resource lifecycles. Our contribution is threefold:

- a mechanism to confine references inside the receiving scope, which opens the way for
- `modal` methods, the statically checked counterpart of Java's `synchronized` methods, and
- modes (= typestates) such as `File.Open` to keep track of object lifecycle at compile time.

Example 1. Well-scoped resource acquisition

```
var rogueWriter : File.Writable?
file.open {
  file.write("Hello!")
  rogueWriter = file    // Error: `file : File.Writable` is confined inside open@
}
```

Example 2. Mutual exclusion of conflicting actions using modal methods

```
modal class Buffer {
  fun append(item : Byte) { … }
  modal fun clear() { … }
  modal fun iterate(block : inner (inner Iterator)-> Unit)  { … }
}

buf.iterate { iterator ->
  buf.append(0xFE)  // This is ok
  buf.clear()       // Error: Modal `buf.clear()` can't be invoked while
}                   //  `buf` is borrowed by the modal `buf.iterate()`
```

Example 3. Staged builders

```
class Html : Tag("html") with AwaitsHead {
  modal! extension AwaitsHead {                                  // Html.AwaitsHead
    break continue@AwaitsBody                                    //  |
    fun head(head : once Head.()-> Unit) = initTag(Head(), head) //  |  head { ... }
  }                                                              //  ↓
  modal! extension AwaitsBody  {                                 // Html.AwaitsBody
    break                                                        //  |
    fun body(body : once Body.()-> Unit) = initTag(Body(), body) //  |  body { ... }
  }                                                              //  ↓
}                                                                // Html

fun html(block : once Html.AwaitsHead.()-> Unit) = Html().apply(block)

// Usage:
val h = html {
  head { ... }
  body { ... }
}                                                                }
```

2 Scope confinement

Kotlin relies on scope-based resource management, but does not enforce scope confinement:

```
var rogueWriter : PrintWriter?
file.printWriter().use {    // Acquire a Writer as an argument
  it.println("Hi!")         // Write to a file
  rogueWriter = it          // Uh-oh, that shouldn't be possible
}
rogueWriter?.println("Peekaboo!")  // java.io.IOException: Stream closed
```

Kotlin already provides a contract called `CallsInPlace` which allows declaring that a parameter lambda is invoked in place. To enforce confinement, we have to generalize this contract to arbitrary types. We propose using `inner` modifier in argument types for inplace arguments:

```
fun <R> Writer.use(block : (inner Writer)-> R) : R
```

`CallsInPlace` also provides a function-specific feature: limiting invocation multiplicity. Being essential for lifecycle-safety checking, it also belongs to the signature:

```
fun <T> Array<T>.forEach(action : inner (T)-> Unit)   // supersedes the CallsInPlace
fun <T> Maybe<T>.forEach(action : once? (T)-> Unit)   // ... with AT_MOST_ONCE,
fun <T> Boxed<T>.forEach(action : once  (T)-> Unit)   // ... EXACTLY_ONCE, or
fun <T> NonEmptyArray<T>.forEach(action : once+ (T)-> Unit) // ... AT_LEAST_ONCE
```

Just like the structural jumps (`return`, `break`, `continue`, and their qualified forms), inplace arguments cannot be stored (`someVar = action`) or captured, except by function literals that are themselves used as inplace arguments. In fact, inplace arguments can be used only for method invocations (like `it.println()`, or `action()`) or passed on as inplace arguments. We rule out the assignment `rogueWriter = it` as desired, but unfortunately, we also rule out concurrency:

```
file.printWriter().use {
  coroutineScope {
    for (i in 1..99) launch {
      delay(Random.nextInt(0, 100))
      it.println("${i.th} asynchronous bottle of beer")   // <- Problem
    }
  }
  it.println("No more bottles of beer!")
}
```

Here, we acquire a `PrintWriter`, launch 99 jobs populating it by `"${i.th} asynchronous bottle of beer"` after a random delay, and add `"No more bottles of beer!"` when they're all done.

We attempt to invoke `it.println(…)` inside a function literal that outlives the invocation of the `launch` function we pass it to. The invocation `it.println(…)` is still safe because it does not outlive the enclosing `coroutineScope`. But how can we make the compiler see this fact?

This form of confinement can be expressed using member classes and interfaces. Kotlin features their half-hearted form, the inner classes. Their constructors `val x = obj.Foo(…)` are members of the host object, but the classes themselves appear as if they were nested (`x : HostType.Foo`), rather than `x : obj.Foo` as in Scala. Expressions of a member type `obj.Foo` can never appear outside of the scope where the host object `obj` is available.[1] If `obj` is an inplace argument, it is impossible to expose an object `x : obj.Foo` beyond the receiving scope of `obj` without upcasting to a base type such as `Any`, that is a supertype which is not a member type of `obj`.

Therefore, `obj.Foo` members that are not exposed via overridden external base type members are unreachable beyond that scope. Such members can be easily detected and allowed to store and capture inplace arguments from enclosing scopes.

[1] Type inference of and smart casts to member types must respect that.

Crucially, objects that store or capture inplace arguments need not be inplace themselves and can be stored inside other objects such as `List<obj.Foo>`: typability guarantees confinement.

Now let us refine the definition of the `CoroutineScope` taking advantage of member interfaces:

```
interface CoroutineScope {
  fun <R> coroutineScope(block : once (inner CoroutineScope).()-> R) : R
  fun <R> async(block : this.Coroutine<R>) : this.Deferred<R>
  fun launch(block : this.Coroutine<Unit>) : this.Job

  inner fun interface Coroutine<T> {
    abstract suspend fun invoke() : T
  }
  inner interface Job {…}
  ...
}
```

Now the function `coroutineScope` passes a fresh coroutine scope as an inplace argument `this`, and `launch` expects coroutine literals of its member type `this@coroutineScope.Coroutine<Unit>`, which are now allowed to capture inplace arguments from the parent scopes of the `coroutineScope`, in particular the `it : PrintWriter` in the above example.

Advanced applications such as mutable concurrent collection libraries and actor frameworks additionally require host-object polymorphism and open / abstract nominal type members.

2.1 Host-object polymorphism

Host-object polymorphism is a typed counterpart of the Rustacean lifetime-polymorphism:

```
fun <cs : *CoroutineScope> join(x : cs.Job, y : cs.Job) : cs.Job
class Foo<cs : *CoroutineScope>(job : cs.Job)  // Rudimentary dependent type
```

Here, `cs` is an erasable compile-time parameter used solely for type checking and it can only be used in type annotations,[2] but not in expressions: `join` cannot invoke `cs.launch`.

To make it accessible inside the function, `reified` modifier can be used:

```
fun <T, reified cs : *CoroutineScope> bar(x : cs.Deferred<T>) // Dependent function
```

The `HostType.InnerType`-notation can still be used in argument signatures as a shorthand:

```
fun foo(x : CoroutineScope.Job)  -->  fun <cs : *CoroutineScope> foo(x : cs.Job)
```

2.2 Type members (a cautionary note)

Type members were first introduced into mainstream programming by Scala, and had a pretty bad reputation ever since. In Scala, abstract type members can be used to express both nominal and structural subtyping, as well as bounded parametric polymorphism, but lead to undecidable subtyping and exhibit very tricky behavior. While attractive regarding the economy of concepts, they are quite the opposite regarding the principle of least astonishment.

Unlike Scala, we propose to introduce only nominal type members[3] (classes and interfaces), which behave in a fairly predictable way and coexist with type parameters (as in `List<T>`) rather than replacing them. This not only reduces the mental overhead, but is also more future-proof: Unlike abstract type members in Scala, parametric polymorphism in Java (and Kotlin) features transitive decidable subtyping assuming material-shape separation,[4] and remains so when we generalise shapes to typeclasses and add higher and parameterised kinds.

[2] Which can be seen as `zero` runtime multiplicity, complementing `once?`, `once`, `once+`, and ordinary arguments.
[3] as introduced in https://dl.acm.org/doi/pdf/10.1145/3371134, 3.4.3
[4] https://www.cs.cornell.edu/~blg59/resources/doc/effing-bound-polymorphism.pdf

2.3 Overridable member classes and interfaces

Presently, Kotlin does not allow overriding nested[5] and inner classes. Only open or abstract inner and nested classes can be overriden. To allow overriding an open or abstract inner or nested class, we propose the syntax `open?` and `abstract?`, while `abstract!` additionally forces concrete descendants to override an abstract inner or nested class by a concrete one. We also need to allow `open?` and `abstract!` for inner and nested interfaces.

For a usage example, consider the `KopyKat` plugin which enables nice syntax for creating updated copies of immutable objects described as data classes:

```
data class House(val squareMeters: Int, val kitchen: Kitchen)
data class Kitchen(val cat: Cat)
data class Cat(val name: String)

fun main() {
  val house = House(
    squareMeters = 100,
    kitchen = Kitchen(
      cat = Cat(
        name = "Garfield",
      )
    )
  )
  val house2: House = house.copy {
    squareMeters = 200
    kitchen.cat {
      name = "Felix"
    }
  }
}
```

It generates a mutable cousin and a copy function for each data class and handles collections manually. Using our proposal, there is no need to handle collections and generate functions, just generate a mutable cousin, making each data class implement the following interface[6]:

```
interface CopyUpdatable<T> {

  fun T.copy(update : once (inner Mutable.()-> Unit)
   = Mutable(this).apply(update).extract()

  abstract! class Mutable(v : T) {
    abstract fun extract() : T
    fun invoke(update : once (inner MutableSelf.()-> Unit) = this.update()
  }
}
```

Now the mutable cousins of data classes can be referred to as `House.Mutable`, `Cat.Mutable`, etc., and we can define custom mutable cousins for immutable collections.

Note that here we also use the `inner` modifier, ensuring that mutability is encapsulated. Inplace arguments allow mutability and concurrency to be used within pure functions. Our proposals regarding static purity-checking and its generalization are discussed in https://akuklev.github.io/kotlin_purity.pdf; we make sure all our proposals fit together, forming a variant of Kotlin throughout amenable to formal reasoning using both theorem provers and model checking based automated and semi-automated verification.

[5] Nested classes and interfaces can be seen as companion object members.
[6] In terms of material-shape separation, it is a shape. Shapes are a special case of type classes.

3 Modal objects

Kotlin type system, as it stands, does not reflect the fact that object members may become unavailable after certain actions, or for the duration of certain actions. For instance, closeable resources cannot be accessed after being closed and mutable collections cannot be mutated while being iterated. Structured confinement opens the way to represent that at type level.

Let us use `break` modifier for methods that finalize their host object and `modal` for methods that lock their host object for the duration of their invocation, ruling out invocation of other modal or finalizing methods. Classes with modal methods must also be `modal`:

```
modal class Buffer {
  fun append(item : Byte) { … }
  modal fun clear() { … }
  modal fun iterate(block : inner (inner Iterator)-> Unit)  { … }
}
```

Classes with finalizing methods must be `modal!` if finalization is mandatory or `modal?` otherwise:

```
modal! fun interface ExactlyOnceFunction<X, Y> {   // Implementation of once (X)-> Y
  break fun invoke(x : X) : Y   // must be invoked exactly once
}

modal? fun interface OnceOrLessFunction<X, Y> {   // Implementation of once? (X)-> Y
  break fun invoke(x : X) : Y   // can be invoked at most once
}
```

References `m : M`, where `M` is a `modal` or `modal?` class, will be called modal references. A modal reference is always either principal or locking. Modal and finalizing methods can only be invoked through the principal reference, which can only happen if there are no other accessible locking references. Locking references are always confined to some scope, but can be freely aliased and captured within this scope. For a mutable object where all setters are modal and getters non-modal, locked and principal references behave as read-only and mutable references in Rust.

Confinement of locking references is essential to ensure that principal references remain essentially unique, i.e. unique modal references to the underlying object if we don't count references captured inside orphaned objects waiting to be garbage-collected or inside fields that cannot be accessed without reflection. Beware that we only control aliasing for modal aliases, and since every class has at least one non-modal supertype (namely `Any`), modal references can always coexist with non-modal references to the same object, providing additional flexibility.

Non-modal reference can be used to access non-modal non-finalizing members of a modal type using atomic guarded invocations `(r as M).foo()` and `(r as? M)?.foo()`, but can never be cast to `M` because it could violate principal refernce uniqueness.

The original reference obtained by creating an object is automatically principal. When used to invoke a finalizing method or passed to a new owner, primary references cease to be primary and change their type to the nearest non-modal superclass (possibly `Any`). We propose the syntax `fun foo(o : object M) { … }` for ownership transfer. When used to invoke a `modal` method or passed as an inplace argument, principal references are shadowed by locking references:

```
buf.iterate {       // outside, `buf` was a principal reference
  buf.append(0xFE)  // inside, `buf` is a confined locking reference
  buf.clear()
}
```

Modal functions may have a `finally { … }` block after their main body, which is executed after the confined locked reference ceases to exist and the principal reference is recovered. This block may invoke other modal functions but is not allowed to finalize the principal reference.

When a principal reference is passed as an argument without `inner` or `object` it is "borrowed". The lent reference immediately ceases to be principal and changes its type to the nearest non-modal superclass, and the receiving scope acquires a confined principal reference. If the reference is `modal!`, the receiving scope is obliged to finalize the object. After the invocation, the original reference regains its principality. In case of non-finalizable classes, it also regains its original type. For `modal!` and `modal?` its modal type can be reestablished by (smart)casts:

```
val f = OnceOrLessFunction<Int, Unit> { println(it) }
foo(f)
when(f) {
  is OnceOrLessFunction<*, *> -> // f was not consumed by foo
  else ->                        // f was consumed by foo
}
```

We also want to briefly mention that `modal!` objects confined in a particular scope are allowed to capture principal references confined in the same scope. Non-confined principal references can be captured by any objects of the same modality. To deal with captured principal references, we need to introduce `modal` fields which can only be accessed in modal methods and thus can temporarily lose principality or change type inside those methods. The ability to capture principal references inside modal objects is essential for many concurrent applications.

3.1 Recovering 'Rust'

To embed the Rust model, we need the following ingredients:

Modal data classes (their setters are declared `modal`, and they can have fields of modal types)

```
  modal data class User(var name : String, var age : Int)
  modal data class Var<T>(var value : T)

  fun foo(user : User)  { … }       // Passing a mutable reference
  fun bar(user : inner User) { … }  // Passing a read-only reference
```

Indentation-sparing `using var` syntax from C#

```
  using file.open                          )  file.open {
  using val connection = withConnection    )    withConnection { connection ->
  ... /* to the end of the scope */        )      ...
                                           )  } }
```

Now we can define lifetimes analogously to `coroutineScope` above:

```
class Lifetime {
  inline fun <modal T, R> local(t : object T, block : once (T)-> R) = block(t)
}
inline fun <R> lifetime(block : once (inner Lifetime).()-> R) = Lifetime().block()

// usage:
fun foo() {      // lifetimes introduce no runtime overhead
  lifetime {
    using val user = local(User("John Doe", 39))
    using val intVar = local(Var<Int>(40))
    user.age = intVar.value
    ...
  } // user and intVar are guaranteed to be unreachable by now
}
```

For a native experience, introduce qualified variable syntax and implicit lifetime scopes:

```
val@foo user = User("John Doe", 39)
var@foo intVar = 40
```

3.2 Join-blocks for parallel initialization and finalization

Quite often, independent resources are being initialized and finalized sequentially:

```
withA { a ->
  withB { b ->
    ...
  }
}
```

In many cases, parallel initialization and finalization would be beneficial:

```
join(withA, withB) { (a, b) ->
  ...
}
```

To implement `join` in a structured concurrent way we need the notion of a *join block*,[7] a simultaneous definition of local suspending once-functions with a common body:

```
join fun f(x : Int) & fun g(y : Int) {
   return@f (x + y)
   return@g (x - y)
}

launch {
  delay(Random.nextInt(0, 100))
  println(f(5))
}
launch {
  delay(Random.nextInt(0, 100))
  println(g(3))
}
```

And that's how one implements parallel resource initialization and finalization:

```
suspend fun <R> join(withA : (once (A)-> Unit)-> Unit,
                     withB : (once (B)-> Unit)-> Unit,
                     block : once (A, B)-> R) : R {
  join fun f(a : A) & g(b : B) & r() : R {
    return@r block(a, b)
  }
  coroutineScope {
    launch { withA(::f) }
    launch { withB(::g) }
    return r()
  }
}
```

Join-blocks allow elegant implementations of arbitrary communication and synchronization patterns highly amenable to structured formal reasoning.

3.3 Modes

To represent objects with a complex lifecycle, we propose borrowing (pun intended) yet another mechanism from Scala, namely the extension classes, described in https://docs.scala-lang.org/tour/self-types.html. Kotlin-style semantics of extension classes could be easily described if inheritance by delegation were available not only for interfaces but also for classes:

```
extension Parent.Mode(…) { … } -->  class Mode(p : Parent, …) : Parent by p { … }
```

[7] See https://en.wikipedia.org/wiki/Join-calculus

Extensions can be declared inside the class they extend, in which case the `Parent.` prefix is omitted. They can also be nested. Extensions are used to refine objects (that is, add and override members) after they have been constructed. Extensions can be constructed using `with`-clauses: `Parent(…) with Mode`. We'll be using extensions to introduce method modifiers `continue@Mode`, `break@Mode`,[8] and `modal@Mode`. It will be crucial to allow overriding modal methods by non-modal ones in extensions.

The `Buffer` definition above is not quite satisfactory since the modality of the `iterate()` method rules out nested and parallel iteration. Let us introduce an extension `AppendOnly` where `iterate()` is non-modal and change the signature of the original `iterate()` to `modal@AppendOnly`. With this modifier, the locked reference that shadows the original principal reference while `iterate()` is executed will be substituted by its `AppendOnly`-extension. The body of `iterate()` can be omitted to delegate to its overriding method in `AppendOnly`. Additionally, we'll introduce the modal method `open` that locks the buffer in `AppendOnly` state to enable parallel iteration:

```
class Buffer {
  fun append(item : Byte) { … }
  modal fun clear() { … }
  modal@AppendOnly fun open(block : once ()-> R) : R = block()
  modal@AppendOnly fun iterate(block : inner (inner Iterator)-> Unit)

  extension AppendOnly {
    fun iterate(block : inner (inner Iterator)-> Unit) { … }
  }
}
```

Now we can use `buffer.iterate {…}` inside `buffer.iterate {…}`, and perform parallel iteration:

```
buffer.open {
  coroutineScope {
    for (i in 1..99) launch {
      buffer.iterate {…}
    }
  }
}
```

Methods with `continue@Mode` modifier substitute the principal reference used to invoke them by its `Mode`-extension. Delegation by omission is allowed as well:

```
modal! fun interface AtLeastOnceFunction<X, Y> {   // Implementation of once+ (X)-> Y
  continue@Unlocked fun invoke(x : X) : Y   // must be invoked at least once
  extension Unlocked : Function<X, Y> {
    fun invoke(x : X) : Y
  }
}
```

Since extensions can be nested, we also need qualified `break@Mode`. Unqualified `break` finalizes the outermost modal parent.

Both `break` and `modal` can be combined with `continue@Mode` allowing arbitrary typelevel state automata. For an example let us consider an HTML builder.[9] It provides an embedded type-safe DSL for constructing HTML:

```
val h = html {
  head { ... }
  body { ... }
}
```

[8] The parallels to ordinary `break` and `continue` become evident when introducing type-safe actor model.

[9] If you are unfamiliar with this example, please consult https://kotlinlang.org/docs/type-safe-builders.html

To require exactly one head and exactly one body after it, we'll need a staged builder:

```
class Html : Tag("html") with AwaitsHead {
  modal! extension AwaitsHead {                                    // Html.AwaitsHead
    break continue@AwaitsBody                                      //  |
    fun head(head : once Head.()-> Unit) = initTag(Head(), head)   //  |  head { ... }
  }                                                                //  ↓
  modal! extension AwaitsBody  {                                   // Html.AwaitsBody
    break                                                          //  |
    fun body(body : once Body.()-> Unit) = initTag(Body(), body)   //  |  body { ... }
  }                                                                //  ↓
}                                                                  // Html
```

Here we declare a staged class `Html` that extends `Tag("html")` and has two additional modes `AwaitsHead` and `AwaitsBody` with methods `head()` and `body()` respectively. Both methods are finalizing methods, but `head()` additionally continues to the `AwaitsBody`, while `body()` leaves the bare non-modal `Html` object which provides members inherited from `Tag`. The initial mode of this object is specified using a `with`-clause borrowed from Scala.

Finally, we want to mention that non-abstract `class Parent with Mode` is allowed to have abstract members as long as they are implemented by `Mode`. Also note that if the extension `Mode` has constructor arguments and/or abstract methods, `continue@Mode` functions, `modal@Mode` and the constructor of `class Parent with Mode` must contain an `init Mode(args) {methods}` block providing those arguments and/or methods. `modal continue@NextMode` functions initialize `NextMode` in their `finally { … }` block. This is also where `modal@Mode` functions have/can to finalize `Mode` if it is `modal!` or `modal?` respectively.

3.4 Join methods

Join blocks can be also used to define methods in multi-modal objects, e.g.

```
class Promise<T> with Awaiting {

  abstract suspend fun await() : T

  extension Completed(val result : T) {
    override fun await() = result
  }
  modal? extension Awaiting {
    join break continue@Completed fun complete(x : T)
       & override fun await() : T {
      init Completed(x)
      return@await x
    }
  }
```

Modal objects and modes tightly intertwine type-checking and control-flow analysis.

4 Conclusion

We have outlined a coherent system of mechanisms for lifecycle-aware resource handling that provides comprehensive correctness guarantees and takes type-safety to a whole new level.

Combined with explicit effects and purity checking, it makes concurrent interactive programming in Kotlin amenable to formal reasoning. Our proposal has to be evaluated by developing a library of concurrent mutable collections and heaps[10] enabling fine-grained concurrent separation logic.

[10] Heaps with various transactional operators, heap-native structures, ghost variables, and conflict-free replicated data types in addition to pure ones, provide implementations for arbitrary flavors of separation logic.