Bound references, modal objects, polyphonic coroutines: A structured approach to resource management

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 prevent leaking, guarantee lifecycle safety, and rule out conflicting actions statically. We devise a mechanism addressing these issues in a manner compatible with and inspired by structured concurrency. Our approach subsumes Rust's borrowing and is closely related to Capture Checking in Scala and OxCaml, but lays more focus on shifting the burden from library users to library developers.

1 Introduction

We propose an extension to the Kotlin type system and flow-sensitive typing mechanism providing static control over aliasing, resource lifecycles, synchronization, and communication:

- `bound` references that cannot leak from their host scope, which open 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 lifecycles at compile time;
- polyphonic structured concurrency for synchronization and simultaneous initialization.

Example 1. Well-scoped resource acquisition (writable reference locked inside)

```
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() { … }
  using fun iterate(block: (&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 (illustrating lifecycle safety for a custom lifecycle)

```
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)
```

2 Bound parameters and object-bound types

Higher-order functions `f(block: (X)-> Y): Z` typically only use their parameter function `block()` inside without ever leaking it outside. Nowadays it can be specified by the `CallsInPlace` contract, but we think this property must be an integral part of the signature so we can know if `block` is allowed to perform non-local jumps and access local variables, etc. Besides, it greatly improves the behavioural predictability of high-order functions. We propose the following notation:

```
fun foo(block: bound (X)-> Y)
```

`CallsInPlace` also allows restricting invocation multiplicity, which we propose denoting as follows:

```
fun foo(block: once (X)-> Y)   // exactly once
fun foo(block: once? (X)-> Y)  // at most once
fun foo(block: once+ (X)-> Y)  // at least once
```

Elsewhere[1] we also propose the modifier

```
fun foo(block: pure (X)-> Y)
```

for functions that do not access anything non-pure at all, so their invocation multiplicity must play no role whatsoever. A typical example would be `sortWith(c: pure Comparator<T>)`.

There is one essential case where we need more flexibility, namely `CoroutineScope.launch`:

```
interface CoroutineScope {
  ...
  fun launch(block: bound(this) suspend ()-> Unit): Job
}
```

Here, `block` is not bound inside `launch` itself but rather inside the surrounding coroutine scope.

Knowing if a parameter leaks the receiving scope is crucial not only for parameters `block : (X)-> Y` of function types. In Kotlin, all objects (that is, values of non-primitive types) are passed by-reference, so both formal and informal reasoning about program behaviour becomes nearly impossible if we don't know whether the parameters are allowed to leak or are used internally only. So let's allow using `bound` for parameters of any non-primitive types, and we'll also need to allow parametrized `bound` for return types:

```
fun foo(f: bound File)
fun bar(f: File): bound(f) FileOutputStream
```

By allowing `bound` in `val`s we can introduce local variables of non-primitive types:

```
fun f() {
  val user: bound MutableUserData
}
```

Bound parameters are so pervasive that the notation `f(x: bound OutputStream)` would bloat signatures if used for every bound parameter, so we propose an even shorter notation: `f(x: &OutputStream)` unless used with function types or parameters:

```
fun <T, R> T.use(block: once (&T)-> R): R
fun <R> coroutineScope(block: once &CoroutineScope.()-> R): R
```

Bound parameters are only allowed to be captured inside objects and function literals which are themselves of bound types and cannot outlive the scope the parameters are bound inside. When bound parameters are passed on as arguments, the compiler must check they are de facto[2] bound, i.e. they don't leak from the receiving function.

[1] https://akuklev.github.io/kotlin/kotlin_purity.pdf

[2] Alowing to pass only to functions that explicitly state boundness would ruin backwards compatibility and interoperability.

Object-bound parameters are crucial for structured concurrency:

```
file.printWriter().use {
  coroutineScope {
    for (i in 1..99) launch {
      delay(Random.nextInt(0, 100))
      it.println("${i.th} asynchronous bottle of beer")
    }
  }
  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. The function literal where `it.println(…)` is invoked is not a simple `bound` parameter, but one bound to the enclosing `coroutineScope`. The invocation is still allowed because the `coroutineScope` is itself bound inside the scope where `it` is bound.

Object-bound return types allow capturing arguments inside freshly created objects:

```
file.use { f ->
  val out = f.outputStream()
}
```

Object-binding of types must be allowed in inheritance lists as well. Let us consider the case that shows up in frameworks like JPA/Hibernate (courtesy of Tunahan Pınar), where operations are run within a transaction, which manages a temporary database session (`EntityManager`):

```
fun <R> transaction(block: once (&EntityManager)-> R): R
```

A bug occurs when an object with lazy-loaded fields, which depends on this live session, is returned from the transaction scope. Any later attempt to access the lazy data will fail because the session has been closed, causing a `LazyInitializationException`. We still want to be able to return such an object, yet stripped of lazily loaded properties. We can do this as follows. We can have a universal class for non-lazy fields and an interface for lazy-loaded ones:

```
class BaseEntity<T>(val id: Long) { … }

interface User : DbEntity{
  val id: Long

  @OneToMany(mappedBy = User::class)
  val posts: List<Post>
}
```

`EntityManager` members that retrieve database entities would not generate their proxy objects:

```
return object : BaseEntity<User>(id), bound(this) User {
  val posts: ProxyList<Post> by em.column("posts", id)
}
```

Let us consider the following piece of code:

```
val user = DatabaseManager.transaction { em ->
    val user = em.find<User>(1L)
    println(user.posts.size) // OK!
    return user // Upcast to the nearest non-bound superclass `BaseEntity<User>`
}
println(user.posts.size) // <- Property .posts not found!
// — what used to be a runtime exception is now being caught already by the IDE.

println(user.id)  // Still OK!
```

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:

- Closeable resources cannot be accessed after being closed;
- Closeable resources cannot be closed while being used;
- Mutable collections cannot be structurally modified while being iterated.

To enforce these constraints, we'll introduce modal methods: methods that are not allowed to be invoked while their host object is being "used by a third party". We propose using the `break` modifier for modal methods that finalize their host object, and the `modal` modifier for methods that lock their host object for the duration of their invocation. Classes with modal methods will also be declared modal. If they have any finalizing methods, it has to be declared if finalization is mandatory (`modal!`) or optional (`modal?`):

```
modal? class ProtectedStore<T> {
  operator fun get(index: Int): T { … }
  modal operator fun set(index: Int, value: T) { … }
  break fun close() { … }
}
```

Whenever pass a modal object as a bound parameter, no modal methods can be called as long as the bound reference exists:

```
val store = ProtectedStore<String>()
store[1] = "Hello"
store.use { store ->
  println(store[1])             // OK
  coroutineScope {
    launch {println(store[1])}  // Also OK
  }
  store[2] = "World"    // Forbidden!
  store.close()         // Also forbidden!
}
store[2] = "World"    // OK!
store.close()         // OK!
println(store[1])     // Store has been closed
```

If `M` is a modal type, we will treat passing parameters `foo(obj : M)` very differently from the case of a non-modal type: as borrowing. As opposed to the case of `bar(obj : &M)`, `foo` will be allowed to call modal methods of `obj` and even required to finalize it if `M` is a modal type with mandatory finalization. Borrowed parameters can be reborrowed to some other functions or objects (see Borrowing by Modal Objects below) or temporarily passed on as bound parameters. Borrowed parameters are not allowed to be captured at all, unless bound first.

Optionally finalizable objects must be cast manually after being borrowed and returned:

```
foo(store)
when(f) {
  is ProtectedStore -> // store was not consumed by foo
  else ->              // f was consumed by foo
}
```

There is a third way to pass a modal object as a parameter: we can upcast them to a non-modal supertype. If `T` is a non-modal supertype of `M`, `foo(x : T)` receives a usual shared reference to `T`, which cannot be used to invoke any modal or finalizing methods of `x`. References of non-modal types `x: T` should never be allowed to be cast to modal types `M`, except in atomic guarded invocations `(r as M).foo()` and `(r as? M)?.foo()`.

At-most-once and exactly-once functions can be defined in terms of modal objects:

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

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

Using modal methods, we can introduce mutable objects with the same usage policies as in Rust. This use case is so ubiquitous we want to introduce a special notation:

```
modal data class MutableAddress(var street: String, var city: String)
// Desugars to
modal interface MutableAddress {
  var street: String
  var city: String
  companion object {
    fun invoke(val initStreet, val initCity) = object : MutableAddress {
      override var street = initStreet
      modal set(value) { field = value }

      override var city = initCity
      modal set(value) { field = value }
    }
  }
}
```

Now if we use `MutableAddress` as a type for a local `val`, it is automatically a local variable (never leaks the scope, can be garbage-collected as soon as the function returns). Mutable/read-only references in Rust exactly match the semantics of our borrowed/bound references, respectively.

One can even go further and extend the definition of normal data classes to automatically generate a modal mutable variant and a deep `copy()` method:

```
data class User(val name: String, val posts: List<Posts>)
// Desugars to
class User(val name: String, val posts: List<Post>) {
  modal data class Mutable(var name: String, val posts: List.Mutable<Post.Mutable>)
  fun copy(block: Mutable.()-> Unit) : User { … }
  ...
}
```

We propose introducing a new modifier keyword `using` to mark receiver (`this`) as a bound parameter. Let us illustrate the usage on the example of the buffer, which can be grown but not cleared while being iterated:

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

For the duration of a `modal` method, the original reference is shadowed by a bound reference:

```
buf.iterate {
  buf.append(0xFE)  // inside, `buf` is a bound reference
  buf.clear()       // Forbidden! Bound reference cannot be used to invoke modal methods
}
```

We also propose using the `using` keyword for indentation-sparing syntax from `C#`:

```
using file.open
using val connection = withConnection
restOfTheBlock
// Desugars to
file.open {
  withConnection { connection ->
    restOfTheBlock
  }
}
```

4 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 { … }
```

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`,[3] and `using@Mode`. It will be crucial to allow overriding modal methods by non-modal ones in extensions.

Methods with `continue@Mode` modifier substitute the host reference by its `Mode`-extension. Delegation by omission is allowed as well:

```
modal! fun interface AtLeastOnceFunction<X, Y> {   // Shorthand: 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
  }
}
```

Methods with `using@Mode` temporarily substitute the host reference by a bound reference to the `Mode`-extension:

```
class File {
  using@Open fun <R> open(block: once ()-> R)
}
```

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

Both `break` and `using` can be combined with `continue@Mode` allowing arbitrary typelevel state automata. For an example let us consider an HTML builder.[4] It provides an embedded type-safe DSL for constructing HTML:

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

[3] The parallels to ordinary `break` and `continue` become evident when introducing type-safe actor model.
[4] 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 `using@Mode` functions have/can to finalize `Mode` if it is `modal!` or `modal?` respectively.

5 Polyphonic structured concurrency

Presently, resources have to be initialized and finalized sequentially even if they are independent:

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

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

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

A structurally concurrent implementation of `join` requires a polyphonic definition, that is a simultaneous definition of multiple single-shot suspend 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))
}
```

Here is how we can implement simultaneous 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()
  }
}
```

We can also allow polyphonic method definitions 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
    }
  }
```

Polyphonic definitions tightly intertwine type-checking and control-flow analysis, but it is the only known way to express arbitrary initialization, finalization, communication, and synchronization patterns in a structurally concurrent way.

6 Conclusion and future work

Both structured programming (with blocks instead of `goto`) and structured concurrency enforce basic correctness by construction and make programs more amenable to both formal and informal reasoning. We have outlined a coherent framework for structured resource management that enforces lifecycle safety by construction, facilitates sound mental models for complex behaviours, and makes concurrent interactive programming amenable to formal reasoning.

Practicality of our proposal has to be evaluated by developing a library of concurrent mutable collections and a declarative actor-based distributed systems framework akin to the P Language.[5]

Besides enforcing correctness by construction, there is another mainstreamable way to ensure correctness: verifiable contract programming,[6] for which structured resource management paves the way. A broad class of contracts only utilizes a decidable fragment of logic, so a static checker can either verify that our program adheres to the contract or generate a minimal counterexample. This way, most functions can be checked to terminate for all valid arguments, sorting methods can be checked to produce a sorted list, etc. We assume this way it will be possible to develop an extensive verified library of conflict-free replicated data types (CRDTs) and lock-free data structures, and ultimately a fine-grained concurrent separation logic framework.

[5] `https://p-org.github.io/P/`

[6] See "Flux: Liquid Types for Rust" by N Lehmann, A Geller, N Vazou, R Jhala