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 lock references inside intended scopes, 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 aliasing and resource lifecycles. Our contribution is threefold:

- a mechanism to lock 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 (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() { … }
  modal 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 Locked references and existentially dependent types

Kotlin, all objects (values of non-primitive types) are passed by reference. Both formal and informal reasoning about program behavior becomes nearly impossible if we don't know whether function calls only use the objects passed to them while being executed, or capture those references so they can be accessed by third parties after they return.

Kotlin relies on scope-based resource management, but does not prevent reference leaking:

```
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
```

Let us introduce a special notation for parameters and receivers that are not allowed to leak:

```
fun <R> Writer.use(block : (&Writer)-> R) : R
fun <R> coroutineScope(block : &CoroutineScope.()-> R) : R
```

One might argue that a contract or a self-explanatory keyword like `local` or `inplace` would be a better solution, but we strive for the most concise notation possible because the inplace parameters are pervasive, and marking them as such in function signatures makes the code more tractable and predictable. We will call the references passed this way locked inside the receiving scope or simply locked references.

Locked references can be used for invoking them or their methods (`it.println()`, `block()`) either directly by the scope they were passed to, or inside function literals that are themselves used as inplace parameters, just like the structural jumps (`return`, `break`, `continue`, and their qualified forms). Besides that, locked references can be passed as inplace parameters themselves. Locked references can be passed on as parameters explicitly marked inplace; otherwise, the compiler must check they are de facto inplace, i.e. they don't leak from the receiving function.

To cover non-trivial use cases, we need to introduce existentially dependent types. These types are parametrized by limited references (`FileOutputStream<'file>`) whose lifetime they are not allowed to exceed. For example, an output stream created for a file cannot outlive the file, and a coroutine launched inside a particular scope can never outlive that scope:

```
fun File.outputStream() : FileOutputStream<'this>
fun CoroutineScope.launch(block : Coroutine<'this>) : Job
```

This way we are allowed to invoke `file.outputStream()` on a limited reference `file` despite it being used in a non-inplace way (namely, being captured inside a new `FileOutputStream` object). Moreover, we solve the notoriously demanding problem of reconciling locality and 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 a non-inplace parameter of `launch`. This invocation is only allowed because the literal's type cannot outlive the enclosing `coroutineScope`.

3 Modal objects and ownership

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 : &(&Iterator)-> Unit)  { … }
}
```

We also propose introducing the notation modal data classes to generate mutable counterparts of data classes, where the setters are declared `modal`:

```
modal data class User(var name : String, var age : Int)
```

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

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

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

Assume we have a freshly created modal `x : M`. There are four ways we can pass it as a parameter `foo(x)` determined by the signature of `foo()`:

- 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()`.
- `foo(x : &M)` receives a locked reference to the object `x`. In this case, the object is locked in its current mode for the duration of `foo`. Neither `foo`, nor any other party can call modal or finalizing methods until `foo` returns.
  For mutable data classes, locked references are read-only references.
- `foo(x : M)` borrows `x`. Borrowed references are also locked inside the receiving scope and prevent the outer scope from invoking any modal/finalizing methods until `foo` returns, but as opposed to the previous case, `foo` is allowed to invoke modal/finalizing methods on `obj` and reborrow it. Borrowing is only allowed when no other locking references are accessible. Borrowing a `modal!` object (e.g. `foo(once (X)→ Y)`) requires `foo` to finalize it.
  For mutable data classes, borrowed references are mutable references.
- Object can be transferred permanently by passing it to `foo(x : object M)`.

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

```
buf.iterate {       // outside, `buf` is either owned or borrowed object
  buf.append(0xFE)  // inside, `buf` is a locked reference to it
  buf.clear()
}
```

Modal functions may have a `finally { … }` block after their main body, which regains access to the original reference and may invoke other modal functions but is not allowed to finalize.

If one borrows or transfers `x`, its type "smart-casts" to the nearest non-modal superclass of `M` (to `Any` as a last resort). For non-finalizable classes it "casts back" after borrowing function returns. For finalizable ones, it can be cast into a modal type manually:

```
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
}
```

Modal objects should also be allowed to have modal fields, so they can own other modal objects or borrow them (if their lifetime is covered by the borrowing scope). Modal fields are essential to reconcile modal objects with concurrency.

3.1 Embedding 'Rust'

With minimal additional effort, we can embed Rust style local mutable variables.
To embed the Rust model, we need the following ingredients:

Modal data classes (they can have modal fields, allowing nesting)

```
modal data class Var<T>(var value : T)

fun foo(user : User) { … }    // Passing a mutable reference
fun bar(user : &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 <T, R> local(t : object T, block : once (T)-> R) = block(t)
}
inline fun <R> lifetime(block : once 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, we can postulate that every function `foo` automatically introduces a lifetime scope and introduce the following syntax for `foo`-local variables:

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

In Kotlin Native, local variables can be implemented accordingly, allowing to vastly improve performance if required.

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*,[1] 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 { … }
```

[1] 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`,[2] 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 : &(&Iterator)-> Unit)

  extension AppendOnly {
    fun iterate(block : &(&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> {   // 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
  }
}
```

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.[3] It provides an embedded type-safe DSL for constructing HTML:

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

[2]The parallels to ordinary `break` and `continue` become evident when introducing type-safe actor model.
[3]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 makes concurrent interactive programming amenable to formal reasoning.[4] Our proposal has to be evaluated by developing a library of concurrent mutable collections and heaps[5] enabling fine-grained concurrent separation logic.

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

[5] 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.