Structured lifecycle management

Lifecycle-aware resource handling and concurrency

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

Kotlin uses scope-based resource management, but lacks mechanisms to ensure that disposable resources are properly finalized and never accessed thereafter, and that exclusive resources are only used exclusively. We propose mechanisms to control object lifecycle and accessibility at the type level, ultimately revealing the common structure underlying structured concurrency in Kotlin and lifetime-based borrow checking in Rust. In conclusion, we propose a generalization of Rustacean lifetimes to handle phased passive objects, and a generalization of Kotlinesque structured concurrency to handle phased active objects, i.e., stateful coroutines and actors. We also propose dedicated notations for dependency injection and managed resource acquisition.

1 Phased objects

Flow-sensitive typing is already available in Kotlin for smart casts, and can be extended to track object lifecycle phases. Currently, it is possible to call `.dispose()` on objects that have already been disposed, causing an exception. Let's introduce an annotation to mark `dispose()`-like methods, so they would recast their host objects to `Unit` preventing any further method calls:

```
fun interface AtMostOnceFunction<X, Y> {
   @finish fun invoke()   // can be invoked only once
}
```

Using `@finish` annotation on function arguments and classes requires the objects to be eventually finished, which is precisely what we need for objects that hold resources or block threads:

```
fun foo(@finish x : T)      // `foo` is required to finish `x`

@finish fun interface ExactlyOnceFunction<X, Y> {
   @finish fun invoke()   // can and must be invoked exactly once
}
```

Objects with `@finish`-members have two lifecycle phases: the initial one where all their members can be accessed, and the final one where no members can be accessed anymore. In general, objects may have more lifecycle phases. Consider the HTML builder[1] providing the notation

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

However, to require exactly one head and exactly one body after it, we'll need a phased builder:

```
class Html : Tag("html") with ExpectingHead {

  interface ExpectingHead : Phase<Html> {                    // Html.ExpectingHead
    @finish<ExpectingBody>                                   //  |
    fun head(block : Head.()-> Unit) = initTag(Head(), block)//  |  head { ... }
  }                                                          //  ↓
  interface ExpectingBody : Phase<Html> {                    // Html.ExpectingBody
    @finish                                                  //  |
    fun body(block : Body.()-> Unit) = initTag(Body(), block)//  |  body { ... }
  }                                                          //  ↓
}                                                            // Html
```

[1] https://kotlinlang.org/docs/type-safe-builders.html

We declare a phased class `Html` with initial phase `ExpectingHead`. The phase `ExpectingHead` has a member `head` finishing at phase `ExpectingBody`, which has a member `body` finishing the lifecycle, thus exposing only methods available through the parent class `Tag`. The function `html` should require a code block receiving an `Html.ExpectingHead` it must finish:

```
fun html(block : @finish Html.ExpectingHead.() -> Unit) : Html {
    val html = Html()
    html.block()
    return html
}
```

A non-abstract class with an initial phase may have abstract members which have to be overridden by all phases. It is non-abstract because the initial phase overrides it. Phases can have abstract members (both functions and properties) too. Member functions and constructors finishing at such phases must initialize them using the special syntax `init<Phasename>{implementation of the interface PhaseName}` (see examples below). We also propose modifier keywords `once` and `oneshot` for functions and lambdas (as `suspend`) to mark functions that are `ExactlyOnce` or `AtMostOnce` respectively. Modifiers `oneshot` and `once` for member functions are only allowed in phase interfaces, and must have a `@finish<T?>` annotation. The modifier `once` must be the only `@finish`ing member function in its respective phase, and that phase must belong to the `@finish`-class.

`@finish`-classes and their phases are also allowed to contain `@finish`/`@finish<T>`-fields, which are required to be finalized/put into the required phase by `@finish`-functions of the respective classes/interfaces. Such fields can be used to store objects that must be finished.

1.1 Passing phased objects

If we pass a phased object as an argument, how do we know in which phase it is afterwards? Every non-inline function accepting a phased object must explicitly declare its usage mode:

```
fun bah(@unsafe x : T)      // see below
fun foo(@borrow x : T)      // not allowed to switch the lifecycle phase of `x`
fun bar(@finish x : T)      // must finish `x`
fun baz(@finish<P> x : T)   // must change the phase of `x` to `P`
fun zee(@finish<*> x : T)   // x comes back in an unknown phase and must
    // be handled using `when(x) { Phase1 -> ...; Phase2 -> ...; ... }`
```

A reference `x : T` to a phased object is called qualified if `T` inherits from `Phase<*>`, and unqualified otherwise. For example, `val x : Tag = Html()` is an unqualified reference, while `val y : Html.AwaitingBody = Html()` is qualified, and so is `val z = Html()` since `Html` itself also inherits from `Phase<*>` through its initial phase `Html.AwaitingHead`.

To ensure that phase changes can only occur via qualified references, we forbid casts `x as P` and `x as? P` for unqualified references `x` if `P` inherits from `Phase<*>`, as well as respective smart casts. If `Html()` is downcasted to `x : Tag`, its phase cannot be changed through `x`. Unqualified references declared `@unsafe` support atomic casted invocations like `(x as? ExpectingBody).body` and `(x as ExpectingBody).body`). If a qualified reference was passed/stored as an `@unsafe` one it becomes `@unsafe` and unqualified, so that each invocation of a phase-specific member function must be an atomic cast invocation. Unsafe references can never (smart-)cast into qualified ones: `if (x is ExpectingBody) { phase of x might still change any moment by a third party }`

Except for the `@unsafe` usage mode, `x` can be passed as an argument only if it's a unique active qualified reference to the underlying object. On the other hand, the called function has the guarantee that it has the only active qualified reference to the underlying object and has to ensure no additional accessible qualified references remain after it returns, which can be achieved using inner classes.

1.2 Capturing phased objects

In Kotlin, we can define inner classes. Unless cast to a non-inner parent type such as `Any`, their instances cannot be exposed beyond the scope where their host objects are available. If we allow abstract inner type members (as in Scala), this property can be used to ensure phased objects are never exposed beyond the appropriate scope. To ensure that no additional qualified references remain accessible after a function/coroutine returns, we must restrict capturing of qualified references as follows:

- Qualified references can only be captured in objects of inner types defined inside the function/coroutine or inner types of the objects created inside the function/coroutine;
- The non-inner parent types of these types may not provide qualified access to captured objects.

The first restriction prevents capturing `@borrow`'ed objects inside generic containers like `List<T>`, while the second restriction prevents capturing `@borrow`'ed inside closures `f : (Xs)→ Y`, because their non-inner parent type provides the `invoke` method which has access to captured `@borrow`'ed objects. To address both limitations, we propose to automatically generate "localised" versions `List@foo<T>` and `((Xs)→ Y)@foo` of required types on demand, with `Any` being their only non-inner supertype. We suggest using explicit type annotations to create localised versions of external types:

```
val l : MutableList@foo<(Int)-> Int> = emptyList()
l.append(f) // here we can append a closure that captures a borrowed argument,
            // which would be impossible for `l` of the type `MutableList<(Int)-> Int>`
```

2 Scope-polymorphic structured concurrency

Coroutines capturing phased objects must have localised types `f : (suspend (Xs)→ Y)@someScope` and can only be launched inside a local coroutine scope `cs = object : CoroutineScope(params)`, and the type of the resulting jobs should be not just `Job`, but `cs.Job`. To handle `Jobs` regardless of their scope, we need to introduce host object polymorphism:

```
fun <cs : &CoroutineScope> foo(j : cs.Job)
```

Or, if we need `cs` not only in type signatures, but also as an object:

```
fun <reified cs : &CoroutineScope> foo(j : cs.Job)
fun <reified fs : &FileSystem> bar(f : fs.File)
```

3 Lifetimes

In Kotlin, we can only have local variables of primitive types. Objects are dynamically allocated and only removed by GC after they become inaccessible. With our new machinery we can emulate local variables of non-primitive immutable types by enforcing inaccessibility outside of the scope. In analogy to coroutine scopes, we have to introduce managed variable scopes with `VariableScope.new` closely resembling `CoroutineScope.launch`, namely

```
class VariableScope {
  fun <T : Immutable> new(t : T) : this.Var<T>(t)

  interface Ref<T> {
    fun get() : T
  }

  interface MutRef<T> {
    val variable : Var<T>
  }
```

```
  class Var<T : Immutable> private constructor(t : T) Ref<T> with Mutable {
    private val value : T
    fun get() = value
    fun getRef() : Ref<T> = this

    interface Mutable : Phase<Var<T>> {
      @finish<Mutable>
      fun set(t : T) {
        init<Mutable> {
          val value = t
        }
      }
      inline fun <R> use(once block : (MutRef<T>)-> R) : R
       = block(object : MutRef<T> { val variable = this })
    }
    init<Mutable> {
      val value = t
    }
  }
}
```

Using `new`, we can create effectively local variables `t : T` which can only be accessed as long as the lifetime is accessible and can be safely disposed of after the scope closes without any further checks; the behavior of mutable and immutable references replicates Rust.

Lifetimes are defined just like `VariableScope`s for phased objects instead of immutable ones, with `Ref<T>` and `MutRef<T>` representing (un)qualified references. Native scopes `foo@` of functions and labeled blocks should automatically be `Lifetime`s for the respective phased objects, which allows lifetime-polymorphic definitions like `fun <l : &Lifetime> bar(r : l.Ref<IoStream>)`, which are instrumental for dealing with dynamic collections of phased objects.

4 Phased coroutines

Being equipped with these extensions, we propose to expand the capabilities of coroutines. We suggest using labeled blocks (`name@ { code }`) in coroutines as runtime-introspectable execution phases. If the job `j` is currently running inside of the labeled block `EstablishingConnection@`, we want `(j.phase is EstablishingConnection)` to hold. The hierarchy of nested blocks in the coroutine should autogenerate a corresponding interface hierarchy.

Those phases may also carry additional data that can be used to track the progress of the job:

```
val j = launch {
  ...prepare
  Moving@ for (i in files.indices) {
    public val progress = i / files.size
    fs.move(...)
  }
}
val u = launch {
  when (val s = j.phase) {
    Moving -> println("Moving files, ${s.progress * 100}% complete")
    ...
  }
}
```

Public properties must have pure (hereditarily immutable and serializable) datatypes to allow instant copy-on-write. Invoking `j.phase` must create an instant snapshot of those properties.

5 Actors: generalized coroutines

Phased coroutines can be generalized to actors:

```
suspend class Foo(params) : Actor {
```

```
  fun foo() : Y {…}
```

All methods in actors are automatically coroutines and have two ways to throw exceptions: the usual one and `throw@Foo`, which is handled by the supervisor rather than the caller. Methods are allowed to change the actor's phase via `@finish` annotations. Some methods may only be available in some phases.

```
  fun bar() : Async {…}
```

In addition to usual methods, actors contain asynchronous methods, distinguished by their return type `protected object Actor.Async`. Asynchronous methods are not allowed to return a value. Their exceptions are always passed to the supervisor.

```
  val p : P = ...
```

Public properties must have pure (hereditarily immutable and serializable) datatypes, getting them produces a stale copy.

```
}
```

```
var r = cs.launch Actor(params)  // r : cs.Ref<Actor>
```

Actors can be launched within (supervised) arenas just as coroutines.

```
r.foo()
```

Actors' members can be called as usual, but their execution happens in the actor's fiber and works via message passing.

```
r.bar()
r.bar().await()
```

Async methods return immediately, passing the respective message. If necessary, their execution can be awaited.

```
val s =  r.phase
```

Produces a stale object of the type representing the current state of the actor with its public properties.

We generalize structured concurrency to embrace stateful actors. Coroutines are essentially actors with acyclic state diagrams. For full-flegded declarative concurrency, we need two additional primitives: parallel joins and rendezvous blocks.

5.1 Parallel joins

Inside coroutines we want to allow parallel joins of coroutine invokations `anyOf(foo(), bar())` which evaluates to whichever returns first, `foo()` or `bar()`. Pure coroutine and queries[2] `baz()` must expand over parallel joins: `baz(anyOf(foo(), bar()))` ⇝ `anyOf(baz(foo()), baz(bar()))`.

Assume `baz(n : Int)` terminates when applied to 5 and stalls when applied to 8. Assume `foo()` returns first yielding 8, while `bar()` takes more time but eventually returns 5. Without expansion `baz(anyOf(foo(), bar()))` ⇝ `baz(8)` would never terminate, while the expended version `baz(anyOf(foo(), bar()))` ⇝ `anyOf(baz(5), baz(8))` is terminating.

[2] see https://akuklev.github.io/kotlin_declarative.pdf

5.2 Rendezvous blocks

A rendezvous block is a simultaneous definition of one-shot coroutines 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))
  val u = f(5)
  println(u)
}
launch {
  delay(Random.nextInt(0, 100))
  val v = g(3)
  println(v)
}
```

The above rendezvous block is roughly equivalent to:

```
val xp = Promise<Int>();   val yp = Promise<Int>()
val fp = Promise<Int>();   val gp = Promise<Int>()
launch {
  fp.complete(xp.await() + yp.await())
  gp.complete(xp.await() - yp.await())
}

oneshot suspend fun f(x : Int) { xp.complete(x); return fp.await() }
oneshot suspend fun g(y : Int) { yp.complete(y); return gp.await() }
```

Rendezvous blocks can also contain non-deterministic joins and `throw@foo` instructions. If a block contains neither `return@foo` nor `throws@foo`, `foo(…)` returns immediately:

```
join fun r(x : Int) : Int & fun f(y : Int) & fun g(z : Int) {
  return@r anyOf(x + y, x + z)
}
```

Coroutines with parallel joins and rendezvous blocks provide the expressiveness of join-calculus, allowing elegant implementations of arbitrary communication and synchronization patterns.

```
suspend class Promise<T> : Actor with Awaiting {
  abstract fun await() : T
  interface Completed : Phase<Promise<T>> {}
  interface Awaiting  : Phase<Promise<T>> {
    @finish<Completed> fun complete(x : T) : Async
  }
  init<Awaiting> {
    join fun await() : T
       & fun complete(x : T) : Async {
      init<Complete> {
        val result = x
        fun await() = result
      }
      return@await x
    }
  }
}
```

6 Dependency injection

Resource management also involves dealing with external services and components. The most straightforward way is to frame external services and components as global singletons:

```
object Database : DbConnection("jdbc:mysql://user:pass@localhost:3306/ourApp")
```

Global singletons are visible to each other and initialized when first used, so no dependency injection is required. While this approach is unbeatably concise, it has serious drawbacks:

- parameters must be known in advance, ruling out config files and command-line arguments;
- tight coupling hinders unit testing and reusability;
- initialization happens in an uncontrolled manner.

These issues are solved by introducing the application class which initialises necessary singletons in the right order according to their dependencies (which may include simultaneous initialisation) and interconnects them. We believe that the language should provide a specialised syntax to make this approach a drop-in replacement for the naïve one with no syntactic penalty, e.g.

```
init ConfigPath() = "./etc/config.yaml"    // init Foo(Dependencies) {initialiser}

init Config(ConfigPath) = Yaml.fromFile(ConfigPath)

init Database(Config) = DbConnection(Config.dbConnString)

class OurApp(params) init(Config, Database) {   // class Bar init(Dependencies)
  ...
  // Listed dependencies (Config and Database) are available as if they were
  // introduced as objects inside OurApp. Unlisted transitive dependencies
  // (ConfigPath) are also initialized on creation, but not accessible by name.
}

fun main() {
  OurApp(params).run()
}

// Both listed and transitive dependencies can be overridden:
fun main(args : Array<String>) {
  OurApp(ConfigPath = args[0] if args.size > 0)
  .run()
}

class UnitTests {
  val app = App(params,
                ConfigPath = "tests/config.yaml",
                Database = MockDb)
  ... tests
}
```

We also want to allow declaring a constructor of an application class as `main()`:

```
class OurApp(args : Args)
      init(override ConfigPath = args.configPath if args.configPath != null,
           Config, Database) : Application {

    @Main constructor(args : Array<String>) : this(parseArgs<Args>(args))

    private class Args(p : ArgsParser) {
      val configPath by p.storing("-C", "--config", "config file path")
    }
    ...
}
```

7 Try-with-resources

Last but not least, we want to improve the syntax of resource acquisition. Currently, there is no support for simultaneous (optionally parallelizable) acquisition of independent resources and acquiring each resource introduces a new level of indentation:

```
localStorage.withState(TrustedPluginsStateKey) {
  withDockIdentity { dockIdentity ->
    withContext(mockDockExit() + mockDockApi() + mockDockPaths(testClass.name)) {
      withFusAllowedStateFlow {
        withSelectedThemeState {
          withSpaceTokensStorage {
            withRemoteClientAndKernel(dockIdentity) {
              withTestFrontend(..args) {
                ...
              }
            }
          }
        }
      }
    }
  }
}
```

We can address both shortcomings by introducing a specialized syntax for resource acquisition allowing simultaneous acquisition of multiple resources. By default, resources would be passed as context, but they can also be named using `as` operator:

```
try(localStorage.State(TrustedPluginsStateKey), DockIdentity as di,
    Context(mockDockExit() + mockDockApi(di) + mockDockPaths(testClass.name)),
    FusAllowedStateFlow, withSelectedThemeState, withSpaceTokensStorage,
    RemoteClientAndKernel(dockIdentity), TestFrontend(..args)) {
  ...
}
```

If there is no block after `try(..)`, the rest of the scope should be treated as the block argument, allowing resource acquisition without indentation:

```
try(FileInputStream(FILENAME))
return readText(Charsets.UTF_8)
```

8 Conclusion

We have outlined a comprehensive system of mechanisms for lifecycle-aware resource handling, and coroutine/actor-based concurrency. The combination of the above mechanisms provides the most comprehensive correctness guarantees of any general-purpose programming language currently available.