Staged Objects and Actors

Structured concurrency and resource handling

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. We propose introducing staged objects to control lifecycle and accessibility at the type level, revealing the common structure behind Rust borrow checking and Kotlin structured concurrency, and featuring a generalization of lifetimes to handle staged resources and a generalization of coroutine scopes to handle staged actors. We also propose dedicated notations for dependency injection and resource acquisition.

1 Staged objects

Staged objects have multiple lifecycle stages that differ in the member functions and properties available. Stage-specific members should only be accessible through qualified references that statically keep track of the object's lifecycle stage in their type using flow-sensitive typing, which is already available in Kotlin for smart casts. The lifecycle stage of an object must not change when there are accessible qualified references to that object, unless the change is triggered in a context where the types of all qualified references can be updated accordingly.

The simplest staged objects are those that may or must be finalized by a `@finish`-method retiring all their members. `@finish`-classes, `val`s, and arguments are required to be finalized.

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

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

For a more advanced example, consider the HTML builder[1] that provides the following notation:

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

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

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

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

Here we declare a staged class `Html` with an initial stage `ExpectingHead` that has a method `head` finishing it at the stage `ExpectingBody`, which has a `@finish`-method `body`. Being defined inside a stage, `body` only retires stage-specific members, leaving the ones inherited from `Tag` available.

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

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 stage may have abstract members which have to be overridden by all stages. It is non-abstract because the initial stage overrides it. Stages can have abstract members (both functions and properties) too. Member functions and constructors finishing at such stages must initialize them using the special syntax `init<SomeStage> {implementation of the interface Stage}` (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 stage interfaces, and must have a `@finish<T?>` annotation. The modifier `once` must be the only `@finish`ing member function in its stage, and that stage must belong to the `@finish`-class.

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

1.1 Passing qualified references

By default, references to staged objects are passed/stored as unqualified references which do not allow accessing any stage-specific properties or changing the stage. To acquire a qualified reference, a non-inline function must specify the object stage after invocation as follows:

```
fun foo(@finish<P> x : T)   // returns `x` at stage `P`
fun bar(@finish x : T)      // finalizes `x`
fun baz(@borrow x : T)      // does not affect the stage of `x`
fun zee(@staged x : T)      // returns x in an unknown stage to be disambiguated by
                            when(x) { SomeStage -> ...; OtherStage -> ...; ... }
fun bah(@unsafe x : T)      // disrupts static stage tractability for `x`
```

The types of qualified references are updated after invocations of `foo` and `bar`. In the case of `zee`, the reference becomes temporarily unqualified, i.e. can be made qualified again using casts and smart casts, which is not possible for unqualified references in general.

Unsafe references do not keep track of the object stage similar to unqualified ones, but allow access to its stage-specific members via atomic cast-invocations of the form `(x as ExpectingHead).head` and `(x as? ExpectingBody).body`. After calling `bah`, the original reference becomes `@unsafe` too and cannot be ever again used or returned as a qualified one.

```
if (x is ExpectingBody) {
  // The stage of x might still change any moment by a third
  // party, so it cannot be smart-cast to Html.ExpectingBody
}
```

Except for the `@unsafe` case, `x` can be passed as an argument only if it's a unique active qualified reference to the underlying object for the time of invocation, while the called function has to ensure no additional accessible qualified references remain after it returns.

1.2 Capturing staged objects

In Kotlin, we can define inner classes. Their instances cannot be exposed beyond the scope where their host objects are available, unless cast to a non-inner supertype such as `Any`. This property can be used to ensure that qualified references are never exposed beyond the appropriate scope. A function leaves no additional accessible qualified references if the following applies:

- qualified references can only be captured in objects of inner types defined inside the function or inner types of the objects created inside the function, and
- their non-inner supertypes 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 localizations:

```
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 staged objects must have localized 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 Variable scopes (Managed heaps)

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`:

```
class VariableScope {

  fun <T> new(t : T) : this.Var<T>(t)

  class Var<T : Immutable> private constructor(t : T) with State {
    private abstract val s : T
    init<State> { val s = t }

    get() = s    // Make read-only access possible with unqualified references

    interface State : Stage<Var<T>> {

      @finish<State>
      fun set(t : T) = init<State> { val s = 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. By introducing transactional operators, phantom variables, and/or generalizing from immutable to conflict-free replicated data types, one can obtain implementations of various flavors of separation logic.

4 Lifetimes

Lifetimes are defined just like `VariableScopes` for staged objects instead of immutable ones.

```
class Lifetime {

  fun <T> new(t : T) : this.Ref<T>(t)

  class Ref<T> private constructor(t : T) Ref<T> with State {
    private @staged val s : T
    init<State> { val s = t }

    fun get() = s                // get unqualified reference

    interface State : Stage<Ref<T>> {

      @finish<State>             // use qualified reference
      fun <R> use(block : once (@staged T)-> R) : R {
        @staged val t = s
        val r = block(s)
        init<State> {s = t}
        return r
      }
    }
  }
}
```

Native scopes `foo@` of functions and labeled blocks should automatically be `Lifetimes` for the respective staged objects, which allows lifetime-polymorphic definitions like `fun <l : &Lifetime> bar(r : l.Ref<IoStream>)`, to deal with dynamic collections of staged objects.

5 Staged 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 stages. If the job `j` is currently running inside of the labeled block `EstablishingConnection@`, we want `(j.status is EstablishingConnection)` to hold. The hierarchy of nested block labels autogenerate a corresponding interface hierarchy. Stages may also carry additional data:

```
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.status) {
    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.status` must create an instant snapshot of those properties.

6 Actors: generalized coroutines

Staged coroutines can be generalized to actors:

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

```
  fun foo() : Y {…}
```

Methods in actors are coroutines and have two ways to throw exceptions: the usual one and `throw@Foo`, which is handled by the supervisor rather than the caller.

```
  fun bar() : Async {…}
```

In addition to ordinary methods, actors allow asynchronous methods, marked 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.status
```

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

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

6.1 Concurrent joins

Inside coroutines we want to allow joins of coroutine invokations `anyOf(foo(), bar())` which evaluate to whichever returns first, `foo()` or `bar()`. Let's also introduce block joins:

```
anyOf {
  foo(), bar() -> baz()  // Launches foo() and bar(), launches baz() after both finish
  gez(), uuf() -> zee()  // Launches gez() and uuf(), launches zee() after both finish
}                      // Returns whichever returns first
```

Assume a pure function `baz(n : Int)` terminates if applied to 5 and stalls if applied to 8. If `foo()` returns first yielding 8, while `bar()` takes more time but eventually returns 5, the expression `baz(anyOf(foo(), bar()))` ⇝ `baz(8)` would never terminate. For best termination chances and best performance, pure functions and queries[2] `baz()` should expand over joins: `baz(anyOf(foo(), bar()))` ⇝ `anyOf(baz(foo()), baz(bar()))` ⇝ `anyOf(baz(5), baz(8))`, which is terminating.

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

6.2 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 : Stage<Promise<T>> {}
  interface Awaiting  : Stage<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
    }
  }
}
```

7 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")
    }
    ...
}
```

8 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)
```

9 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.