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

Like many languages, Kotlin uses scope-based resource management. Yet, is currently not capable of ensuring disposable resources are properly finalized and never accessed thereafter, nor that exclusive resources are used exclusively. We outline how to manage objects' lifecycle on the type level, ultimately revealing the common structure underlying structured concurrency in Kotlin and lifetime-based borrow checking in Rust. In conclusion, we generalize Kotlin's structured concurrency to a type-safe structured actor model.

1 Pending objects

Objects that may hold resources or block threads must be finalized in a timely manner. The first step in ensuring correctness when dealing with such objects, is to mark them by inheriting from the `Pending` interface, and to mark the `@Finalizing` members with the appropriate annotation:

```
class SomeResource : Pending {
  ...
  @Finalizing fun release() {...}
}

abstract class Continuation<T> : Pending {
  @Finalizing abstract fun invoke(value : T)
}
```

When we pass such objects to functions, we can either require them or forbid them to finalise the objects:

```
fun foo(@pending x : T)   // foo MUST finalize `x`
fun bar(@borrow  x : T)   // foo is not allowed to finalize `x`
```

A pending object can no longer be mentioned after it has been finalised or passed as a `@pending` argument. Functions that create or obtain pending objects, can only pass them around as pending or borrowed arguments, and each possible execution path must either finalise the object or pass it as a pending argument.

However, these requirements are not sufficient to ensure correctness, because a reference to a pending or borrowed object could be captured as an element in a list, within a field of an object, or as a variable within a closure which could be run as a separate job. We need to ensure that these jobs are finished and that these objects are either finalised or inaccessible before we finalise the object.

We propose to introduce `@pending` annotations also for fields within classes that inherit from `Pending`. These must be finalised by all methods that finalise the host object, while the objects contained within non-`@pending` fields are not allowed to be finalised. `@pending` objects `x` can only to be captured inside other `@pending` objects `y` which must be explicitly finalised before `x`, unless `x` is stored inside a `@pending` field in which case they are automatically finalised with `y`.

2 Type-based accessibility management

In Kotlin, we can define inner classes inside classes and functions/coroutines. Unless they are cast into a non-inner parent type such as `Any`, values of these types cannot be exposed beyond the scope in which their host objects are available. If we also allow abstract inner type members in interfaces and abstract classes as in Scala, we can use these to ensure that

`@borrow`'ed objects are never exposed beyond the scope they were borrowed into by imposing the following restrictions:

- `@borrow`'ed objects can only be captured within objects of inner types defined within the class/function/coroutine or within inner types of objects created there.
- The non-inner parent types of these types may not provide access to `@borrow`'ed 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 `this.List<T>` and `this.((Xs)→ Y)` 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 : this.MutableList<(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>`
```

3 (More) structured concurrency

Type-based accessibility management requires an adjustment to structural concurrency. Namely, we can only launch coroutines `f : this.(suspend (Xs)→ Y)` that capture borrowed objects only inside local coroutine scopes `cs : this.CoroutineScope`, and the type of the resulting jobs should be not just `Job`, but `cs.Job`.

To handle `Job`s 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)
```

4 Inplace objects

Using the outlined approach to accessibility management, we can implement the approach proposed in Linearity and Uniqueness: An Entente Cordiale[1] combining it with ideas by Lionel Parreaux.

It is often desirable to pass an object without allowing it to be captured or exposed:

```
fun foo(@inplace x : X)     // `x` can be only used inside `foo` while `foo` is executed
```

Inside `foo`, `x` can only be passed to non-inline functions as an `@inplace` argument, can only allowed be captured in an `@inplace`-annotated field, and with the same restrictions as if it were `@borrow`'ed, to ensure that it can never be ever exposed beyond the scope of `foo`. Additionally, `x` disapears from the root context after being captured, which means it can be captured at most once.

With `@inplace` annotation we can control reference uniqueness. Let us introduce the following annotation:

```
fun bar(@dedicated x : X)      // `x` is required to be a unique reference to its object
@dedicated fun baz(...) : Y    // return value is guaranteed to be a unique reference
```

[1] https://link.springer.com/chapter/10.1007/978-3-030-99336-8_13

If `x` was created locally, obtained as a `dedicated` argument or a `dedicated` return value, and had neither been captured nor passed outside (except as an `@inplace`-argument), we can be sure it is a unique reference and therefore pass it as `dedicated` argument or return as a `dedicated` return value.

Let us also introduce the interface `Inplace` to mark classes and interfaces of objects not intended to be captured or exposed, for instance, the contexts for type-safe builders. A variable of an inplace type can be used only inplace unless cast to a parent type that does not yet inherit from `Inplace`, e.g. `Any`. A variable can be permanently cast into an inplace type only if it is `@dedicated`, otherwise, it can be cast for a single method invocation only `(x as T).someMethodOfT()`.

Since calls to methods of `Inplace`-interfaces of `@dedicated` objects are linearly ordered, with inplace types, we can introduce type-level state machines. For instance, we can make a type-safe builder for HTML that requires exactly one head and exactly one body after it:

```
fun html(init : (@dedicated HtmlAwaitingHead).() -> Unit) : HTML {
    val html = HTML()
    html.init()
    return html
}

class HTML @NextState<HtmlAwaitingHead> constructor() {} : TagWithText("html") {}

interface HtmlAwaitingHead : HTML, Inplace {
  @NextState<HtmlAwaitingBody>
  fun head(f : Head.()-> Unit) = initTag(Head(), init)
}

interface HtmlAwaitingBody : HTML, Inplace {
  @Finalizing
  fun body(f : Body.()-> Unit) = initTag(Body(), init)
}

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

5 Lifetimes

In Kotlin, objects are normally only removed by GC after they are inaccessible, but using introduced machinery we can enforce inaccessibility outside of a specific scope. In analogy to coroutine scopes, we can introduce managed lifetimes with `Lifetime.new` closely resembling `CoroutineScope.launch`, namely

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

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

  interface MutRef<T : Immutable> : Inplace {
    fun get() : T
    fun set(t : T)
  }
```

```
  class Var<T : Immutable> : Inplace {
    inline fun useRo(@once block : (@borrow this.Ref<T>)-> R) : R {...}
    inline fun useRw(@once block : (@dedicated this.MutRef<T>)-> R) : R {...}
  }
}
```

Using `new`, we can create scoped objects `t : T` which can be only accessed as long as the lifetime is accessible and safely disposed after the scope closes. Variables created by `new` can be "opened" either in read-only or in read-write mode, but not at the same time since `Var : Inplace`. Opening in read-only mode acquires a `@borrow`'ed read-only reference, which can be shared between multiple concurrently running jobs, but is guaranteed to become inaccessible before the scope of the block that acquired the read-only reference closes, and read-write access can be granted. Opening in `read-write mode` acquires an `Inplace` reference, so that all reads and writes must be linearly ordered. And again, this reference is guaranteed to become inaccessible before the variable can be reopened in read-only mode. Using the already presented syntax

```
fun <lt : &Lifetime> foo(v : lt.Var<Int>)
```

we can enjoy the same level of lifetime polymorphism as in Rust.

Our implementation only supports immutable objects, but using annotations we can actually implement lifetimes as they work in Rust, and beyond. To deal with inplace objects `t : T` implementing type-level state machine functionality, we will actually need Lifetimes `<T : Inplace> new(@dedicated t : T)` that provide read-write references giving access to the real `t` (`useRw(@once block : (@dedicated T)→ R) : R`)) and "read-only" references giving access to `t` cast to the nearest ancestor not inheriting `Inplace`.

6 Introspectable 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 states. If the job `j` is currently running inside of the labeled block `EstablishingConnection@`, we want `(j.state is EstablishingConnection)` to hold. The hierarchy of nested blocks in the coroutine should autogenerate a corresponding interface hierarchy.

Those states 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.state) {
    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.state` must create an instant snapshot of those properties.

7 Actors: generalized coroutines

Coroutines are single-shot variants of more general entities known as 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 state via `@Finalizing` and `@NextState<S>` annotations. Some methods may only be available in some states.

```
  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 : @dedicated 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.state
```

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

With structured resource management, we can finally have a proper type system embracing stateful actors. For full-flegded declarative concurrency, we need two additional primitives.

7.1 Parallel joins

Inside coroutines we want to allow parallel joins of coroutine invokations `( foo() | bar() )` which evaluates to whichever returns first, `foo()` or `bar()`. Pure coroutine and queries[2] `baz()` must expand over parallel joins: `baz( foo() | bar() )` ⇝ `( 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(foo() | bar())` ⇝ `baz(8)` would never terminate, while the expended version `baz( foo() | bar() )` ⇝ `( baz(5) | baz(8) )` is terminating.

7.2 Rendezvous blocks

A rendezvous block is a simultaneous definition of single-shot coroutines with common body:

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

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

The above rendezvous block is morally equivalent to:

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

@singleShot suspend fun f(x : Int) { xp.complete(x); return fp.await() }
@singleShot 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 no `return@foo` and `throws@foo`, `foo` returns immediately:

```
def together r(x : Int) : Int & f(y : Int) & g(z : Int) {
  return@r (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.

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

8 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 overriden:
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
}
```

9 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)
```

10 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.