Distributed Kotlin:

Declarative joins, staged coroutines, and typesafe actors

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

Having introduced explicit representation and structured control over object lifecycles, we turn to "active" objects that change their lifecycle stage themselves rather than being manipulated from outside: staged coroutines and actors. We introduce language primitives that allow implementing arbitrary communication and synchronization patterns, which are presently treated as external mechanisms provided by the platform (primarily, JVM). This way we obtain a self-contained typesafe actor-based declarative structured concurrency model.

1 Staged coroutines

Having developed an approach to staged objects, we propose to expand the capabilities of coroutines accordingly. 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.

```
interface CoroutineScope {

  fun launch(block : this.Coroutine<Unit>) : this.Job<block.Status>

  inner fun interface Coroutine<T> {
    abstract suspend fun invoke() : T
    open? interface Status {}
  }
  inner interface Job<Status> {
    val status get() : Status
    ...
  }
  ...
}
```

2 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[1] `baz()` should expand over joins: `baz(anyOf(foo(), bar()))` ⇝ `anyOf(baz(foo()), baz(bar()))` ⇝ `anyOf(baz(5), baz(8))`, which is terminating.

3 Join blocks

A join 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 join 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())
}

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

Join 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 concurrent joins and join blocks provide the expressiveness of join-calculus, allowing elegant implementations of arbitrary communication and synchronization patterns.

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

To give an example, let us consider a definition of a promise:

```
class Promise<T> with Awaiting {

  abstract suspend fun await() : T

  extension Completed(val result : T) {
    override suspend fun await() = result
  }
  modal? extension Awaiting {
    join override fun await() : T
       & break continue@Completed fun complete(x : T) : Async {
      init Completed(x)
      return@await x
    }
  }
```

4 Introducing Actors

Let us introduce actors, which can be seen as generalized coroutines:

```
suspend class Foo(params) {...}
-->
inner class Arena.Foo(params) : this@Arena.Actor { ... }

arena {
  val r = Foo(params)
}

suspend class Foo(params) {

  fun foo() : Y {…}
```

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.

```
  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 = ar.MyActor(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.

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: concurrent joins and join blocks.