Structured resource management

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 confine resources to their respective scopes, control their lifecycle statically, and rule out conflicting actions. We outline how to introduce such mechanisms, augmenting structured concurrency by structured confinement and lifecycle staging. Our model can be seen as a generalization of Rust approach.

1 Structured confinement

Kotlin uses scope-based resource management, but does not enforce resource confinement:

```
var tmp : Any?
file.printWriter().use {     // Acquire a Writer as an argument
  it.println("Hi!")          // Write to a file
  tmp = it                   // Uh-oh, that shouldn't be possible
}
(tmp as PrintWriter).println("Peekaboo!")  // java.io.IOException: Stream closed
```

Kotlin readily supports structural jump expressions like `return@foo true` which are only allowed to occur within their respective scopes and inline function literals that are used as arguments of inline functions. Let's introduce `borrow`-arguments following the same usage policy restrictions:

```
fun <R> Writer.use(block : (borrow Writer.Open)-> R) : R
```

The limitation to inline function literals excludes a variety of use cases, including 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 coroutine literal passed to the `launch` function is obviously not being inlined, but it still should be allowed to capture `it : PrintWriter`. To enable such use cases, we propose an approach based on a recently proposed form of path-dependent types with decidable subtyping.[1]

Kotlin supports inner classes whose constructors receive the host object: `val x = obj.Foo()`, but the types of the resulting instances can be written as if they were not inner, but merely nested classes: `x : HostObjectType.Foo`. A better approach, pioneered by Scala, treats their types as members of the host object itself: `x : obj.Foo`, a *path-dependent type*. This way it is impossible to refer to the inner type `obj.Foo` outside of the scope where `obj` is available. Declaring a variable of such a type or casting a value `(tmp as obj.Foo)` obviously requires `obj` (which should be respected by type inference and smart casts). If `obj` is a confined object, it is impossible to expose an object of its inner type `obj.Foo` beyond the scope of `obj` without upcasting to an external base type such as `Any`, which makes `obj.Foo`-specific members inaccessible without reflection unless exposed by overridden base type members. Thus, we can safely allow such members to refer to the confined objects from the scope of `obj`, which turns out to be sufficient to reconcile borrowing with structured concurrency and a broad variety of other use cases.

[1]https://dl.acm.org/doi/10.1145/3371134

Let us refine the definition of the `CoroutineScope` taking advantage of inner types:

```
interface CoroutineScope {
  fun <R> coroutineScope(block : borrow CoroutineScope.()-> R) : R
  fun <R> async(block : this.Coroutine<R>) : this.Deferred<R>
  fun launch(block : this.Coroutine<Unit>) : this.Job

  inner fun interface Coroutine<T> {
    abstract suspend fun invoke() : T
  }
  inner interface Job {…}
  ...
}
```

Now the function `coroutineScope` passes a fresh coroutine scope `cs` as a confined context object, and `launch` expects coroutine literals of its inner type `cs.Coroutine<Unit>`. Since this type is confined to the same scope as `cs`, these coroutine literals are allowed to refer to confined objects available where `cs` was created, in particular the `it : PrintWriter` in the above example.

1.1 Path-dependent and path-polymoprhic functions

With a global interface `Deferred<T>` we can easily write a function taking any deferred value:

```
fun <T> foo(x : Deferred<T>)
```

To deal with inner deferred types `cs.Deferred<T>`, the host coroutine scope `cs` must be required either as an additional argument or as a compile-time parameter. We propose the following notation:

```
fun <T> foo(cs : CoroutineScope, x : cs.Deferred<T>)

fun <T, cs : *CoroutineScope> foo(x : cs.Deferred<T>)

fun <T, reified cs : *CoroutineScope> foo(x : cs.Deferred<T>)
```

In the first case, `cs` is an additional argument and can be used at runtime. In the second case, `cs` is an erasable compile-time parameter. It can only be used in type expressions. The third case is syntactic sugar for the first one: the CoroutineScope `cs` is passed as a true argument and can be used at runtime, but need not be passed explicitly as it can be derived knowing `x`.

2 Structured borrowing and principal references

There is more to borrowing than merely acquiring objects that are not allowed to escape the acquiring scope: overborrowing (but not re-borrowing) should be impossible. It allows using borrowing to rule out conflicting actions such as clearing a buffer while iterating it:

```
class Buffer {
  fun append(item : Byte) { … }
  borrow fun clear() { … }
  borrow fun <R> iterate(block : (borrow Iterator)-> R) : R
}

buffer.iterate { iterator ->
  buffer.append(0xFE)  // This is ok
  buffer.clear()       // Compile-time error: the buffer can't be borrowed by
}                      // .clear() as it is already borrowed by .iterate()
```

We make `clear()` and `iterate()` borrow their host object to rule out clearing while iterating. Borrowing methods are statically checked overhead-free analogs of Java synchronized methods.

Borrowing requires the notion of principal references, a generalization of reference uniqueness. A manifestly unique reference to an object is also the principal one, but aliasing `val r2 = r` repeals the uniqueness for both, while allowing one of them to remain principal. We'll be using `object` as a type-modifier keyword signifying that principal references are required or returned:

```
fun foo(x : object X) : object Y {…}
val x = Foo(…)            // A constructor always creates a principal reference
val y : Foo = x           // Create a non-principal alias, which is the default case
val z : object Foo = x    // Make a new principal reference, x ceases to be principal
foo(z)                    // Pass the principal reference, z ceases to be principal
```

While principal references allow non-principal references to the same objects, they are effectively unique principal references, i.e. unique if we don't count references captured inside orphaned objects waiting to be garbage-collected or inside fields that cannot be accessed without reflection.

Functions that borrow their context or arguments require a principal reference to be invoked, and this reference ceases to be principal for the time of their execution. Structured confinement is necessary and sufficient to guarantee that the original interface can be safely made principal again after the function finishes.

2.1 Borrowing modes

The `Buffer` definition above is not quite satisfactory, because making `iterate()` a borrowing method rules out nested and parallel iteration. We want `buffer.iterate()` to borrow `buffer` while still allowing `buffer.iterate()` but not `buffer.clear()` inside. To achieve this, we'll need to borrow (pun intended) yet another mechanism from Scala, namely the mixins, described in `https://docs.scala-lang.org/tour/self-types.html`. Mixins are used to refine objects (that is, add and override members) after they have been constructed. A mixin interface `Foo` extending a class `Base` is an inner interface of `Base`, but it can be defined not only inside `Base` but also afterwards similar to extension functions (`mixin interface Base.Foo {…}`). As opposed to classes extending `Bar`, mixins do not create an instance of `Bar` and cannot be instantiated on their own, but only in addition to `Bar`-instantiation in `with`-clauses: `Bar(…) with Foo`

With this mechanism, we can introduce a special form of borrowing `borrow<BorrowingMode>`. In addition to borrowing the object, it (temporarily) refines the (temporarily non-principal) lent reference with `BorrowingMode` mixin using flow-sensitive typing.

```
class Buffer {
  fun append(item : Byte) { … }
  borrow fun clear() { … }
  borrow<Tame> fun tame(block : ()-> R) : R = block()
  borrow<Tame> fun <R> iterate(block : (borrow Iterator)-> R) : R = this.tame {
    this.iterate(block)
  }

  mixin interface Tame {
    fun <R> iterate(block : (borrow Iterator)-> R) : R { … }
  }
}
```

Now we can use `buffer.iterate {…}` inside `buffer.iterate {…}`, and perform parallel iteration:

```
buffer.tame {
  coroutineScope {
    for (i in 1..99) launch {
      buffer.iterate {…}
    }
  }
}
```

3 Staged objects

Scope-based resource management paradigm assumes that finalization scenario is unique. To deal with cases where the finalization scenario is non-unique, e.g. when finalization requires arguments, let us introduce a modifier for finalizable objects and their finalizing methods:

```
finish fun interface OnceFunction<X, Y> {
   finish fun invoke(x : X) : Y   // must be invoked exactly
}
finish? fun interface OneshotFunction<X, Y> {
   finish fun invoke(x : X) : Y   // can be invoked at most once
}
```

A `OneshotFunction` can be invoked only once, because calling `invoke()` finishes the object, making sure that the `invoke()` method itself is no longer accessible. Every `OnceFunction` *must* be finalized, so the `invoke()` method must be called, and exactly once.

Invoking a finish method should require a principal reference, to guarantee that no object can be finished twice. To guarantee that no members of finalizable types are accessed after the object's finalization, we only allow principal references to have finalizable types. To invoke such a method through a non-principal reference, atomic guarded calls `(w as Writer.Open).write()` or `(w as? Writer.Open)^?.close()` must be used.

Invoking a finish method should require a non-borrowed principal reference: when you borrow something, you're not supposed to destroy it. Since an object's finalization cannot happen inside of the scope where this object was borrowed, we can have non-principal references of finalizable types inside such scopes, which is essential for type-safe concurrency.

For the cases when the receiver is supposed to finalize the object, the `finish` modifier can also be applied to arguments and receiver context, signifying that the function has to finalize this argument or the context, respectively. Similarly to `borrow`, these modifiers also require a principal reference and introduce a confined principal reference inside. By making the lent reference non-principal, they also change its type to its nearest non-finalizable supertype.

With mixins, we can generalize to objects with multiple lifecycle stages. For an example let us consider an HTML builder.[2] It provides an embedded type-safe DSL for constructing HTML:

```
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 AwaitsHead {

  finish? mixin interface AwaitsHead {                         // Html.AwaitsHead
    finish<AwaitsBody>                                         //   │
    fun head(block : Head.()-> Unit) = initTag(Head(), block)  //   │   head { ... }
  }                                                            //   ↓
  finish? mixin interface AwaitsBody  {                        // Html.AwaitsBody
    finish                                                     //   │
    fun body(block : Body.()-> Unit) = initTag(Body(), block)  //   │   body { ... }
  }                                                            //   ↓
}                                                              // Html
```

Here we declare a staged class `Html` that extends `Tag("html")` and has two named lifecycle stages `AwaitsHead` and `AwaitsBody` having methods `head()` and `body()` respectively.

[2]If you are unfamiliar with this example, please consult https://kotlinlang.org/docs/type-safe-builders.html

Both methods are transition methods, that is they `finish` their respective stages. In the case of `head()`, we encounter a parameterized annotation `finish<AwaitsBody>`, which means that the method does not only finish the stage it belongs to, but also subsitutes it by the stage `AwaitsBody` after execution, while `body()` simply finishes the stage it belongs to leaving the bare `Html` object which only provides non-stage specific members inherited from `Tag`.

The initial stage of this object is specified using a `with`-clause borrowed from Scala, where it is used for mixin interfaces. We use this notation because lifecycle stages can be considered a special case of mixins.

The function `html` should require a code block receiving an `Html.AwaitsHead` it must finish:

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

Parametrized `finish<NextStage>` modifier can be also used for arguments and contexts, requiring the function to transition its argument to the specified stage. A special form `finish<*>` allows the function to finish the object, perform any stage transitions, or just leave it as it is. After calling a `finish<NextStage>`-function, the primary reference should be smart-cast to `NextStage`, and in case of `finish<*>` manual casts are required.

For backwards compatibility and some cases of low-level interoperability, we also allow unrestricted (`finish<*>`) principal references to be passed or captured as `unsafe` references, which can be used to invoke any methods (including finishing and borrowing ones) using atomic guarded invocation. In this case, the original reference ceases to be principal forever.

We have seen how the `finish` modifier applies to methods, classes, and arguments. The same annotation can also be applied to fields of finalizable objects, denoting that they are required to be finalized (`finish`) or put into the required stage (`finish<NextState>`) by finalizing methods of the respective class or stage. Such fields can be used to store objects that must be finished. The `finish` and `finish?` modifiers can also be applied to function literals `finish { … }` and function types `finish ()→ Unit`, making the respective functions invokable exactly or at most once respectively, and thus allowing them to invoke other exactly-once or at-most-once functions.

Finally, it is important to briefly explain how the proposed notation interacts with the preexisting modifiers. Staged classes can also be declared abstract, open, or sealed, with inheritance working as expected. We leave details outside of the scope of the present memo. However, we wanted to highlight that non-abstract staged classes are allowed to have abstract members if they are implemented in all stages. The stages in their own right are also allowed to have abstract members. Methods and constructors finishing at such stages must provide their implementations using `init<SomeStage> { … }`-blocks shown in more detail in the Heap example below.

3.1 Variable scopes (Heaps)

Let us showcase what can be achieved using the introduced mechanisms. In Kotlin, we can only have local variables of primitive types, while objects are dynamically allocated and garbage-collected. With our new machinery, we can emulate local variables of non-primitive types by enforcing inaccessibility outside of the variable scopes closely resembling coroutine scopes from above.

```
class Heap {

  class Var<pure T>(initialValue : T) {
    private var value : T = initialValue
    fun get() = value
    borrow fun set(newValue : T) { value = newValue }
  }
}
fun <R> varScope(block : borrow Heap.()-> R) : R = (Heap().block)

fun foo() {
  varScope {
    val v = Var(5)
    v.set(15)
    .. here we can do everything with v
  }
  .. v can be safely disposed of after the scope closes without any further checks
}
```

The behavior of principal and non-principal references to `Var`s precisely replicates that of mutable and read-only references in Rust. With a bit more hustle, we can extend our heap to allow reference-valued variables, which can contain references to the variables inside this heap:

```
  class Ref<T>(initializer : ()-> object T) Ref<T> with State {
    private abstract val s : object T
    init<State> { override val s = initializer() }

    fun get() : T = s                  // get non-principal reference

    finish? mixin interface State {
      finish<State> fun <R> use(block : finish (finish<*> T)-> R) : R {
        val t : object T = s     // Take the principal reference from the field
        val r = block(t)         // Run the block with primary reference
        init<State> { override val s = t } // Store the updated primary reference
        return r
      }
    }
  }
}
```

We can define custom forms of heaps. By adding various transactional operators, heap-native structures, ghost variables, and using conflict-free replicated data types in addition to immutable ones, one can obtain implementations for arbitrary flavors of separation logic.

4 Conclusion

We have outlined a coherent system of mechanisms for lifecycle-aware resource handling, providing the most comprehensive correctness guarantees of any general-purpose programming language currently available. Our approach has to be evaluated by developing a library of concurrent mutable collections and heaps supporting fine-grained concurrent separation logic.