Structured resource management

Alexander Kuklev[1,2] ⟨a@kuklev.com⟩

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

Like many other languages, Kotlin uses scope-based resource management. In this memo, we outline how to improve a wide range of aspects of resource management, ultimately embracing the lifetime-based approach pioneered in Rust as a special case.

1 Syntactic improvements

Currently, acquisition of every resource introduces a new indentation level in Kotlin, which may lead to code like this:

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

We propose to introduce a special operator `try`, which allows to acquire multiple resources in one stroke. By default, resources will be passed as context, but if required they can be named using `as` operator. With operator either accepts a block of code in braces, or takes all the code till the end of scope as such a block:

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

Furthermore, if no opening brace follows after `try(..)`, the rest of the scope should be treated as block argument, allowing to acquire resources with no indentation at all:

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

2 Pending objects

Objects that may hold resources or block threads have to be timely finalized. The first step to ensure correctness dealing with such objects, is to mark them by inheriting from `Pending` interface and mark the `@Finalizing` members by the respective annotation:

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

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

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

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

A pending objects cannot be mentioned after it was finalized or passed as a `@pending` argument. Functions that create or obtain a pending objects, can only pass it around as pending or borrowed arguments, futhermore each possible execution path must either finalize the object or passe it as a pending argument.

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

We suggest allowing the `@pending` annotations for fields inside classes inherited from `Pending`, which are required to be finalized by all methods marked as `@Finalizing`. Objects from fields not marked this way are not allowed to be finalized. `@pending` objects `x` are only allowed to be captured inside other `@pending` objects `y` which must be explicitly finalized before `x`, unless `x` is stored inside a `@pending` field in which case it is automatically finalized when `y` is finalized.

3 Type-based accessibility management

In Kotlin, we can define inner classes inside classes and functions/coroutines. Unless cast into a non-inner parent type such as `Any`, values of those types cannot be exposed beyond the scope where 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 are only allowed to be captured inside objects of inner types defined inside the class/function/coroutine or inner types of objects created there.
- The non-inner parent types of those types must provide no access to captured objects. As a simple-to-check overapproximation we might allow only `Any`.

The first restruction prevents to capture `@borrow`'ed objects inside generic containers such as `List<T>`, while the second restriction means we cannot capture `@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 restrictions we propose automatically generating "internalized" versions `this.List<T>` and `this.((Xs)→ Y)` of all types on demand, with `Any` being their only non-inner supertype. We propose to use explicit type annotations to create internalized versions of outer 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>`
```

4 (More) structured concurrency

Structured accessability requires an adjustment to structural concurrency. Namely, we can only launch coroutines `f : this.(suspend (Xs)→ Y)` capturing borrowed objects only inside coroutine scopes `cs : this.CoroutineScope`, and the type of the resulting jobs should be not just `Job`, but `cs.Job`.

To deal with any `Job`s inside any coroutine scopes polymoprhically (and in simular such situations), we need special syntax:

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

In case, we'll need `cs` to be used not just in type signatures, but also as an objects, we should also allow

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

5 Inplace objects

Using the outlined approach to accessability 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 to capture or expose it, so let's introduce the following annotation:

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

This means, `x` can be only passed to non-inline functions as `@inplace` argument and either not captured at all, or disappear from the context after being captured, which means it can be captured at most once. We only allow it to be captured into an `@inplace`-annotated field, and with same restrictions as if it was `@borrow`'ed, to guarantee it cannot be ever exposed beyond the scope of `foo`.

With this 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 the object it refers to
@dedicated fun baz(...) : Y    // return value is guaranteed to be a unique reference
```

If `x` was created locally, obtained as a `dedicated` argument or a `dedicated` return value, and had not been not 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()`.

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

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:

```
html {
  head { ... }
  body { ... }
}
```

```
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)
}
```

6 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 reassembling `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 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 implment 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`.

7 General capture checking

Currently we have only proposed controlling capture of `@inplace`, `@borrow`'ed, and `@pending` objects, but it is possible to control capture in general. For any type `T` let `@pure T` denote values of the type `T` that do not capture or refer to any objects except for pure data (values of primitive types, strings, pure functions, data and value classes recursively containing only pure data).

Now let `@pure(x,...,z)` mean "pure except for x,...,z", where the arguments are objects: these may be local objects or the whole local scope `this`, as well as global objects such as `System`, or particular methods and/or fields of such objects e.g. `@Pure(Logger, System::println, this)`. Capture checking and effect polymoprhism can be modelled after Scala3, see https://docs.scala-lang.org/scala3/reference/experimental/cc.html.

Additionally, we one could to allow negation as first proposed by Lionel Parreaux:

```
class Buffer<T> {
  ...
  fun <R> iterate(block : @pure(~this) (Iterator<T>)-> R) : R
}
```

- here we allow `block` to access everything except for the Buffer it is being called on.

8 Conclusion

The combination of the above mechanisms provides the most comprehensive resource handling correctness guarantees of any general-purpose programming language currently available.