Controlling effects

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

In many cases, high-order functions such as `sortWith(comparator)` only have meaningful behaviour if their arguments are pure functions. Type-level control over the purity of functions and data is essential to statically prevent nonsensical behaviour and dangerous vulnerabilities.

1 Pure values

Pure values are values of primitive data types (`Boolean`, `Int`, `Float`, etc.), enums, strings, pure functions, immutable arrays of these, and value classes encapsulating any of these. In addition, we propose to introduce `value data class`'es and `sealed value class`'es to encode algebraic data types. Value data classes can only contain pure values, and sealed value classes can only be extended by value data classes, other sealed value classes, or objects:

```
sealed value class AlgebraicIntList {

  object EmptyAlgebraicIntList : AlgebraicIntList()

  value data class NonAlgebraicIntList(val tail : AlgebraicIntList,
                                       val head : Int) : AlgebraicIntList()
}
```

Constant properties must be allowed to contain any pure values, not only strings and values of primitive data types, as it is currently mandated in Kotlin.

2 Pure functions

Above we mentioned pure functions, so we need to introduce a modifier `pure (Xs)→ Y` to mark functions, coroutines, or lambdas that do not capture or refer to any external objects (except for pure values), both local and global ones such as `System`, `Runtime`, etc. The modifier should be applicable to any type `T` that does not capture or refer to external objects, and only matches its instances and descendants that still do not capture or refer to external objects. Now the sort function can require the comparator to be pure:

```
fun <T> Array<out T>.sortWith(comparator : pure Comparator<in T>)
```

3 Semi-purity: permissions and effects

It is often desirable to allow semi-purity instead of purity. For example, it does no harm to allow `comparator : pure(Logger) Comparator<Int>` to use the `Logger`. Parameterised purity annotation `pure(vararg permissions) T` corresponds to capability-aware types `T^{caps}` and `X →{caps} Y` of the Scala3 experimental capture checking mechanism[1] which allows fine-grained control of effects and capture checking, see Scoped Capabilities for Polymorphic Effects.[2]

Negative permissions[3] can be used to prevent collection modification while iterating:

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

Here, the `block` is allowed to access everything except for the buffer it is being called on.

[1] https://docs.scala-lang.org/scala3/reference/experimental/cc.html
[2] https://arxiv.org/abs/2207.03402
[3] https://cse.hkust.edu.hk/~parreaux/publication/draft24/