# Controlling purity

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 data

Pure data includes 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 data, 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()
}
```

Currently `const val`'ues in Kotlin are only allowed to be strings or values of primitive data types. We propose to extend the notion to arbitrary pure data.

## 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 data), 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 the sort function can require the comparator to be pure:

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

## 3 Semi-purity: Capabilities, effects, and capture checking

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

Capability negations[3] can be used to statically prevent modification while reading:

```
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/