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 behavior if their arguments are self-contained functions that are safe to re-execute and produce output that only depends on their arguments, so-called pure[1] functions. Pure functions are inseparable from pure datatypes, the types of inherently immutable self-contained values.

Type-level control over the purity of functions is essential to statically prevent nonsensical behavior and dangerous vulnerabilities. Control over the purity of type parameters is essential for precise signatures in the interfaces of remote services and distributed frameworks, as well as for concurrent containers, conflict-free replicated datatypes, and staged actors. As we'll show, pure functions and datatypes also enable a more flexible approach to compile-time constants.

1 Pure data

Pure datatypes are primitive datatypes (`Boolean`, `Int`, `Float`, etc.), enums, strings, pure functions, immutable arrays of pure data, and (final or sealed) classes and objects in which all fields are immutable and pure, and all methods are pure functions. We propose introducing a modifier keyword `pure` for type parameters (to require purity) and in class and interface declarations (to assert purity for them and their descendants). Pure datatypes include algebraic datatypes:

```
pure sealed class LinkedList<pure T> {

  pure data class NonEmptyLinkedList<pure T>(val head : T,
                                             val tail : List<T>) : List<T>
  pure object EmptyLinkedList : List<Any?>()
}
```

2 Pure functions

We propose introducing the `pure` modifier for functions and function types to forbid them to access any external non-pure methods and objects of not-pure datatype, including global ones like `println` and `Runtime`. Now the sort function can require the comparator to be pure:

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

It is often desirable to allow some exceptions, e.g. we can allow using `println` and the `Logger`:

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

Semantics of the `pure(exceptions)` annotation can be generalized beyond function types by interpreting `pure(args) T` as capturing types featured in Scala[2] `T^{args}` that allow fine-grained control of effects and capabilities, see Scoped Capabilities for Polymorphic Effects.[3]

3 Constants

Constant properties must be allowed to have pure types, not only strings and values of primitive datatypes, as it is currently mandated in Kotlin. Their values should be allowed to contain not only literals and other constants, but also `pure(CompileTime)`-functions applied to literals and constants, where `CompileTime` provides access to the build environment and resource files:

```
const val APP_ENV = CompileTime.getenv("APP_ENV") ?: "DEV"
const val DB_SCHEMA = DbSchema("jdbc:sqlite:./resources/prototypeDb")
```

[1]We allow runtime exceptions in pure functions.
[2]https://docs.scala-lang.org/scala3/reference/experimental/cc.html
[3]https://arxiv.org/abs/2207.03402