Data, constant expressions, pure functions

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

We propose introducing modifier combinations `value data class`, `value fun interface`, and `value (Xs)→ Y` to enforce hereditarily immutable and self-contained objects devoid of identity besides equality. Since such types are inherently serializable, one can allow constants of non-primitive mere types. Being self-contained, value functions can be executed at compile time, provided their arguments are known at compile time, allowing for rich constant expressions.[1]

In many cases, high-order functions such as `sortWith(comparator)` rely on on purity of their arguments. With a bit of additional effort, we can single out pure functions among self-contained ones. By enforcing purity, we can prevent odd behavior and eliminate possible vulnerabilities. Additionally, it enables optimization as pure functions are safe to compute ahead of time, postpone, re-execute if necessary, or exempt from execution altogether if their result is ignored.

1 Value types

Let us define value types as primitive datatypes (`Boolean`, `Int`, `Float`, etc.), enums, strings, immutable arrays `Array<value T>`, `value fun` interfaces including `value (Xs)→ Y`, and `value data` classes and objects. All member functions of value datatypes must be self-contained, and all their fields must be immutable and of value types, making them hereditarily immutable.

Type parameters, interfaces, abstract and sealed classes can be also declared to be `value`-types. In particular, we obtain genuine algebraic datatypes:

```
sealed value class AdtList<T> {

  value data class NonEmptyAdtList<T>(val head: T,
                                      val tail: AdtList<T>) : AdtList<T>
  value data object EmptyAdtList : AdtList<AnyVal?>()
}
```

2 Self-contained functions and pure functions

Self-contained functions `f: value (Xs)→ Y` are functions that are only allowed to invoke, access, or capture external entities that are self-contained constants. Pure functions[2] `f: pure (Xs)→ Y` are self-contained functions that never alter any data except their local variables. They can only invoke other pure functions and read properties of their receiver object and arguments.

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

3 Functions with explicit effects (possible extension)

Sometimes it is desirable to require purity modulo some fixed methods or modulo receiver methods in general[3] when dealing with typesafe builders `context(MutableList<E>) ()→ Unit`:

```
fun <T> Array<T>.sortWith(comparator: pure(Logger::trace) Comparator<T>)
fun <E> pureBuildList(build: pure(MutableList<E>) ()-> Unit): List<E>
```

This extension paves the way for explicit effects[4]:

```
pure(Raise<SomeException>) fun foo() { … }
pure(Console) fun bar() { … }
```

[1]Partial support for these features is currently being implemented by Ivan Kylchik and Florian Freitag
[2]Purity is compatible with runtime exceptions and incapsulated mutability (local `vars`).
[3]For type-level behaviour of semipurity see "Scoped Capabilities for Polymorphic Effects" (arXiv:2207.03402).
[4]See also https://arrow-kt.io/learn/typed-errors/