Data, constant expressions, and checked effects

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 `value` enabling

```
sealed value class LinkedList<mere T> {   // Algebraic data types, yay!

  value data class NonEmptyLinkedList<mere T>(val head : T,
                                              val tail : List<T>) : List<T>
  value data object EmptyLinkedList : LinkedList<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 `f: pure (Xs)→ Y` are self-contained functions that never alter any data except their local variables.[2] 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 Checked capturing and checked effects

Value classes and their methods are not allowed to capture or invoke any external non-value objects or functions. Let's allow controlled exceptions: `value(Logger, ::println) (Xs)→ Y`. Functions `pure(Logger) (Xs)→ Y` will be allowed to call non-pure members of listed objects.

We can also introduce `pure(ctxDecl) (Xs)→ Y` working as `pure context(ctxDecl) (Xs)→ Y` while allowing to call non-pure methods of context parameters, recovering checked exceptions:

```
pure(Handler<IOException>) fun myFunction() { … }
```

As opposed to their non-parametrized forms, `value(…)` and `pure(…)` are quite intricate to deal with. Fortunatelly, the details have already been worked out by Martin Odersky et al. in "Tracking Captured Variables in Types" and "Scoped Capabilities for Polymorphic Effects".

[1]Partial support for these features is currently being implemented by Ivan Kylchik and Florian Freitag
[2]Local `var`s and runtime exceptions are still allowed.