Constants, mere data, and checked effects

Alexander Kuklev[1,2] ⟨a@kuklev.com⟩

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

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

Pure functions are self-contained functions that never alter directly or indirectly any data except their local variables. They can only invoke other pure functions and read out mere data-valued fields of their receiver object and arguments. Self-contained functions that only have arguments of mere data types are automatically pure.[2] Pure functions are safe to compute ahead of time, postpone, re-execute if necessary, or exempt from execution altogether if their result is not required. In many cases, high-order functions such as `sortWith(comparator)` rely on the purity of their arguments, so explicit purity annotations enable preventing potential vulnerabilities.

1 Mere data

Mere data types are primitive datatypes (`Boolean`, `Int`, `Float`, etc.), enums, strings, immutable arrays `Array<mere T>`, `mere fun` interfaces including `mere (Xs)→ Y`, `mere data` classes and objects, and `value` classes wrapping mere data. All member functions of mere classes and interfaces must be self-contained, and all their fields must be immutable and have mere data types, making them hereditarily immutable. Interfaces, abstract and sealed classes can be also declared `mere` and type parameters can be restricted to mere types enabling algebraic data types:

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

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

2 Self-contained functions and pure functions

Self-contained functions `f: mere (Xs)→ Y` are functions that are only allowed to invoke, access, or capture external entities that are self-contained constants. To allow exceptions for particular objects or functions, let's introduce its parametric form `f: mere(Logger, ::println) (Xs)→ Y`.

With additional restrictions stated above, it can be strengthened to a purity modifier for functions and function interfaces:

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

It is also possible to introduce its parametric form `pure(ctxDecl) (Xs)→ Y` which is essentially the same as `pure context(ctxDecl) (Xs)→ Y`, but allows invoking non-pure member functions of context parameters. This extension allows introducing checked exceptions/algebraic effects:

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

As opposed to their non-parametrized forms, `mere(…)` and `pure(…)` are quite intricate to deal with. Fortunatelly, their semantics has already been developed by Martin Odersky et al. in "Tracking Captured Variables in Types"[3] and "Scoped Capabilities for Polymorphic Effects".[4]

[1]Partial support for these features is currently being implemented by Ivan Kylchik and Florian Freitag
[2]We allow runtime exceptions in pure functions.
[3]https://arxiv.org/abs/2105.11896
[4]https://arxiv.org/abs/2207.03402