First-class query functions for Kotlin

Alexander Kuklev[1,2] ⟨a@kuklev.com⟩

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

We propose introducing query functions inspired by the recently developed Verse calculus,[1] a novel approach to deterministic functional logic programming combining the powers of Haskell and Prolog. In a limited form which is much easier to implement, query functions can extend `SQL`-like reactive query languages (see Exposed: Kotlin SQL Framework) by recursive queries that leverage the power of Datalog while remaining easy to read, understand, and maintain.

1 Queries and deterministic functional logic programming

The expressivity of conditions, say in filters, can be greatly improved with a first-class notion of queries. While functions compute *the* value for given arguments, queries describe what's *a* value for given arguments:

```
data class Person(val father: Person, val mother: Person)

query fun Person.parent = anyOf(this.mother, this.father)
query fun Person.ancestor = anyOf(this.parent, this.ancestor.parent) // Recursion!

persons.filter { it.ancestor == x }
```

We define what's *a* `.parent` and *an* `.ancestor` to `.filter` persons with *a* given ancestor `x`. Such code is much leaner than any imperative alternative, leaving no place for a bug to hide.

Queries `s: query (Xs)→ Y` can only be invoked inside other queries or cast into potentially infinite sequences `all{ s } : (Xs)→ Sequence<Y>`, just like[2] coroutines can be invoked only inside other coroutines or launched. Inside queries, variables can be multivalued and entangled: `x` and `y` in `{ val x = anyOf(0, 1); val y = 1 - x }` can both be `anyOf(0, 1)`, but can never be equal. As entanglement is incompatible with irreversibility, queries must be side-effect-free.[3]

Being lazy and multivalued, queries can contain implicit definitions with non-unique solutions:

```
val (n: Int, m: Int) {
  2*n + m == 5
}
// generates:
val (n: Int, m: Int) = anyOf(Pair(0, 5), Pair(1, 3), ... )
```

Implicit definitions can employ any combinations of pure functions and queries

```
val (n: Int, m: Int) {
  f(n, m) ≤ this.ancestor.age
}
```

because one can brute-force a stream of solutions by successively applying `f` to all possible pairs of integers $(0, \pm 1, \pm 2,..)$ until they satisfy the condition. Verse calculus, a novel approach based on introspectable queries, provides a reasonably effective alternative to brute-forcing.[4] Queries applied to finitary data, databases and -streams can be made very efficient owing to highly optimized Prolog and Datalog implementations.

In their full generality, queries go far beyond databases and constraint satisfaction problems into the realm of complex real-world applications, where declarative business logic description is vital for managing the inherent complexity. Their applications also include complex event processing and conflict resolution for interacting software systems.

[1]https://simon.peytonjones.org/assets/pdfs/verse-icfp23.pdf

[2]The types `query T` implement the $\diamond$ modality, the dual of the $\Box$ modality `const T` of the S4-modal type theory, while `suspend T` is the interactive modality dual to the self-contained modality `pure T`, assuming non-blocking IO.

[3]Apart from interactions with quantum objects in the case of a hypothetical quantum programming language.

[4]Variables must be of pure separable datatypes, which include all enumerable types and Polish spaces like $\mathbb{R}$.