Declarative Kotlin

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

Declarative programming is an approach striving to enable describing what you want to get rather than how to get it. It facilitates concise, transparent, straight-to-the-point code and is an indispensable tool for tackling inherent complexity. Here, we outline how to introduce declarative programming capabilities into Literage Kotlin, a language with great support for declarative `DSL`s including an `SQL`-like reactive query language. Full-scale implementation of our proposals brings the full power of functional logic programming (roughly "Haskell + Prolog"), while already limited support extends the query language with an accessible form of Datalog.

1 Implicit definitions

Implicit definitions enable declarative programming whenever objectives can be described by conditions. They are ubiquitous in mathematical texts, so supporting the widest possible class of them is highly desirable for a language used in academia. We propose the following notation:

```
let x y : Float
  x + 2y = 5
  x - y = 4
```

```
let gcd : Int
  n % gcd = 0
  m % gcd = 0
  maximizing { gcd }
```

```
try let x ?t : Float
  x = a + b·t
  x = c + d·t
```

A `let` block contains conditions imposed on the indeterminates declared in its header. Conditions must uniquely determine the values of the indeterminates except for so-called existential variables (marked like `?t`), which are scoped within the block and not exposed. A `let` block can only be compiled if there is an appropriate solver for conditions of the given form on indeterminates of given types. Solvers have to ensure the existence of a unique solution,[1] either at compile-time (`let` blocks) or at run-time (`try let` blocks). At present, we envision three specialized solvers:

- the *-semiring linear equation solver,
- the mixed integer and real linear arithmetic solver,
- an SAT/SMT (boolean satisfiability/satisfiability modulo theories) solver.

2 Queries

The expressivity of conditions, not only in `let` blocks but also in ordinary conditionals and filters, can be greatly improved with 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)

def a Person.parent = anyOf(this.mother, this.father)              # Just 2 values
def an Person.ancestor = anyOf(this.parent, this.ancestor.parent)  # Recursion!

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

We define what's *a* `.parent` and *an* `.ancestor` of a person to `▸filter` those with *an* 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]

[1]Take `a = c, b = d = 0` in the rightmost example. Its solution `x = c` qualifies as unique because `t` is existential.
[2]The types `query T` implement the ◇ modality, the dual of the □ 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.

3 Implicit definitions in queries

By the very nature of queries, existence and uniqueness restrictions for `let` blocks can be lifted, and solvers are merely required to produce an effectively exhaustive[4] stream of solutions x_i. Given a computable `f : (Int, Int)→ Int`, queries can employ `let` blocks like this:

```
let n m : Int
  f(n, m) = c
```

It is valid because one can brute-force a stream of solutions by successively applying `f` to all possible pairs of integers (0, ±1, ±2,..) until the result turns out to be equal to the constant `c`. Verse calculus,[5] a novel approach based on introspectable queries, provides a reasonably effective alternative to brute-forcing, which also supports any computationally verifiable conditions instead of `(= c)`, a query `f` instead of a function, any surveyable[6] data type `T` instead of `Int`.

Such generality brings the full power of functional logic programming, extending the applications far beyond academic purposes, into the realm of complex real-world applications, where the declarative description of business logic is vital for managing the inherent complexity.

Owing to an extensive body of research on Answer Set Programming, Prolog, and Datalog, queries applied to finitary data such as databases and -streams can be made very efficient, but it's not the performance which matters most. The prime purpose of declarative programming is to provide self-evident reference definitions instead of error-prone intricate ones. Reference definitions can be optimized by hand, if necessary, provided they exist in the first place.

4 Conclusion and outlook

By leveraging declarative programming, we can substantially improve the expressivity of Kotlin and extend its capabilities to a whole new level. These extensions make it a superior choice not only for literate programming and illustrating ideas in teaching and research, but also for a variety of real-world applications where the taming of inherent complexity is urgently needed.

Indeterminate functions in queries virtually eliminate the need for subqueries and dramatically reduce the code in size and complexity, making it easier to understand and less error-prone. Functional logic programming provides a tractable approach to dealing with vast combinatorial complexity, e.g. in conflict resolution for interacting software systems and constraint satisfaction.

When it comes to academic and educational use, implicit definitions help to keep code as close to the text as possible, or to provide straightforward equivalents for intricate algorithms so as to elucidate all subtleties and catch all the bugs while establishing the equivalence.[7]

While being remarkably useful, both pure functional programming (à la Haskell) and logic programming (à la Prolog) have a reputation for being arcane academic gimmicks. The recently developed Verse calculus has finally made it possible to combine them into functional logic programming, which is generally assumed to be even less capable of gaining broad adoption in the foreseeable future. Our approach aims to wield the full power of functional logic programming without expecting the users to have any understanding of the underlying calculus. To this end, we discreetly introduce functional logic programming into a mainstream programming language using merely two non-intrusive, reader-friendly, and self-explanatory constructs.

[4] Dense in the space of solutions `S` with respect to the topology given by computationally verifiable predicates. Reduces to simple exhaustiveness if `S` is enumerable or has verifiable equality.

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

[6] Separable with respect to the same topology as above. Includes enumerable types and Polish spaces like $\mathbb{R}$.

[7] For instance, `gcd` as defined in §1 is an equivalent for the non-trivial Euclidean algorithm (except `n = m = 0`): `def gcd(n m : Nat) = if(m = 0) n else gcd(m, n % m)`