Academic Kotlin

Alexander Kuklev[1,2] ⟨a@kuklev.com⟩

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

Modified form of Literate Kotlin can be used for computer science papers and in mathematics as a language for constructions and proofs. These applications require dedicated extensions, we'll outline in this memo. High abuse potential makes some of them undesirable for a general-purpose programming language, so we propose introducing a separate dialect: Academic Kotlin.

1 Concise signatures

In academic applications, the signatures of functions and polymorphic types get quite convoluted. For reasonable readability, it is essential to keep them as short as possible. Significant improvements can be achieved with space-separated lists of variables sharing the same type

```
def plus(x y : Int) : Int
```

and name-based default type conventions:

```
reserve m : Int, prefix n : Int, suffix count : Int
```

With this declaration, all identifiers reading `m` (and also indexed ones, like `m2`), and multipart identifiers with the first part `n` or the last part `count` (e.g. `nUsers` and `pointCount`) are assumed `Int` by default. Dependent default type conventions[1] facilitate concise polymorphic signatures.

2 Aliases

Mathematics and academic computer science require short variable names and concise notation for formulas, but short names need descriptions, and fancy operators need pronounceable names. We propose a dual naming scheme, with pronounceable alphanumeric default identifiers and descriptions/concise notations (which may include non-`ASCII` characters and introduce custom symbolic operators) as aliases, written in backticks afterwards.

```
val n `element count` = ...                # field/variable descriptions

class List<T `element type`>               # parameter/argument descriptions

enum class Boolean `𝔹`                     # unicode identifiers

def not(b : 𝔹) `¬b`                        # prefix operators

def factorial(n : ℕ) `n!`                  # postfix operators

data class Pair<out X, out Y> `X × Y`      # infix operators

def floor(x : Float) `⌊x⌋`                 # closed operators with parameters

def List<T>.get(idx : ℕ) `this[idx]`       # postfix operators with parameters
```

Note that operators can have parameters, e.g. the indexed access operator `arr[i]` is a postfix operator with parameter `( [_])` . In mathematics, many binary operators, such as tensor and semidirect products, may have optional parameters represented as subscripts or superscripts. Parsing techniques developed for the Agda programming language allow to handle such operators.

To refer to operators directly, we propose the following notation:

```
 ::(-) for ::minus        ::(- ) for ::unaryMinu        ::( --) for ::dec
```

Spaces on the right or left indicate prefix and postfix operators, respectively.

[1]http://agda.readthedocs.io/en/v2.7.0/language/generalization-of-declared-variables.html

3 Operator tightness

By default, operators should have unspecified precedence, so expressions like `-n!` should be treated as syntax errors due to the ambiguity `(-n)!` vs `-(n!)`. Expressions `a ∘ b ∘ c` for binary infix operators should also be rejected due to the ambiguity `(a ∘ b) ∘ c` vs `a ∘ (b ∘ c)`. However, we propose to support vararg infix operators ``plus(vararg x : Float) `+` : Float``, in which case chains `a + ⋯ + c` are interpreted as `plus(a, ..., c)`.

We propose to specify the tightness of operators by annotations extending the `OperatorCategory` interface. Unlike numbers, operator categories are merely required to form a directed acyclic graph and do not have to be pairwise comparable, which is a good thing: non-obvious expressions should not be given arbitrary meanings. Furthermore, operator categories can specify custom interpretations for chains of operators belonging to that category: The category `@EqRel` of comparison operators resolves their chains `a < b ≤ c` into conjunctions `(a < b and b ≤ c)`.

Infix operators can have different right and left tightness. Minus always binds tighter on the right, so that `a - b - c` would resolve to `(a - b) - c`. It can also be defined to bind tighter than `(+)` on the right, but not on the left, so `a + b - c + d` would parse as `((a + b) - c) + d`.

Combining all these techniques we can even embrace the infamous Donald Knuth's path notation

```
draw a -- b -- c --cycle           # A triangle, (--)-lines are straight
draw a ~~ b ~~ c ~~cycle           # A circle through abc, (~~)-lines are curved
draw a ~~ b ~~ c ~- d -- e --cycle  # (~-) connect smoothly only on the left side

draw a ~~ b ~~[tension: 1.5, 1]~~ c ~~ d
draw a [curl: k]~~ c ~~[curl: k] d
draw a ~~ b [up]~~ c [left]~~ d ~~ e.
draw (0,0) ~~[controls: (26.8,-1.8), (51.4,14.6)]~~
 (60,40) ~~[controls: (67.1,61.0), (59.8,84.6)]~~ (30,50)
```

enabling a METAFONT/Ti*k*Z-compatible declarative reactive technical illustration framework for dynamic figures in interactive textbooks and educational apps.

4 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          let gcd : Int            try let x ?t : Float
  x + 2y = 5               n % gcd = 0              x = a + b·t
  x - y = 4                m % gcd = 0              x = c + d·t
                           maximizing { gcd }
```

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,[2] 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.

Implicit definitions were also introduced into programming by Donald Knuth's METAFONT.

[2]Take `a = c, b = d = 0` in the rightmost example. Its solution `x = c` qualifies as unique because `t` is existential.

5 Type classes and structure hierarchies

Academic applications require typeclasses we introduce elsewhere.[3] We propose modeling type class inheritance and nominal subtyping after the Arend[4] language. Type class subtyping should soundly represent hierarchies of algebraic structures, which leads to quite intricate cases, which can be illustrated on the rig (semiring, ring without negation) example:

Diamond problem Ri(n)gs extend monoids in two ways: they form a monoid with respect to both addition and multiplication, which can be expressed using by call-site field renaming:

```
structure Rig<this R> : Monoid<R>(::times), AbMonoid<R>(::plus)
```

Circularity We can define the class of modules over a given rig, and define (unital associative) algebras over a given rig as a monoidal object in modules over that ring. Ultimately, we observe that a rig can be seen as an algebra over ℕ (ring as an algebra over ℤ, abelian group as a module over ℤ, etc), which should be ideally reflected by subtyping. This issue can be addressed by allowing nominal (bi-)convertibility to be established retroactively:

```
structure Module<R : Rig, this M> : AbMonoid<M>(::plus) { … }

typealias Algebra<R : Rig, this A> = Monoid<A>(::times) within Module<R>

establish Algebra<ℕ> ≡ Rig
          Module<ℕ>  ≡ AbMonoid(::plus)
          Rig        ≡ Monoid(::times) within Module<ℕ>
```

6 Juxtaposition as an operator (and embedded APRiL[5])

In mathematics, sequences of uppercase or cursive variables are conventionally treated not as multi-letter identifiers, but chains of individual variables connected with juxtaposition operator `juxtapose<T>(vararg ops: T)`. We propose following this convention in academic Kotlin:

```
fun avg(𝑥, 𝑦) = 𝑥𝑦/2
```

It also allows treating delimiterless strings of point labels as names for segments (`AB`), angles (`∠AOB`), and polygons (`ABCD`) as it is customary in geometrical proofs. Digits and apostrophes following a letter should be treated as part of the identifier with digits rendered as indices: ($\triangle A_1B'A_2$). Backticks can be used to access multi-letter uppercase identifiers: `` `ABC` ``. If we intend to use academic Kotlin for proofs with Lean4-style tactics, we can write them like this:

```
establish Pythagoras’ Theorem
given △AOB with ∠AOB = ¼ turn
prove |AB|² = |AO|² + |OB|²
  take P on OA with |PA| = |OB|
  take Q on OB with |QB| = |OA|
  take R on (○|OP⟩ ∩ ○|OQ⟩) with R ≠ P
  prove OPRQ is square by
    |OP| = |PR| = |RQ| = |QO|
    |OR| = |PQ|
  ...
```

[3] http://akuklev.github.io/kotlin_typeclasses.pdf
[4] http://arend-lang.github.io/assets/lang-paper.pdf
[5] http://april-lang.org — APRiL: A geometric PRoof Language