Academic Kotlin

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

Modified 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 declarations

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 Concise notations and detailed descriptions

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 and input methods. 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 optional alternatives, 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 be also 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`.

By combining custom operator categories and operators with parameters, we can even accommodate the infamous example of operator complexity, the METAPOST 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)
```

4 Companion objects and type classes

Following a ridiculous tradition popularized by Smalltalk in the 70s, Kotlin interprets operators as methods of their left operands. Mathematically, operators should be attributed to the companion object of their operand(s): `2 + 3` should mean `Int.plus(2, 3)` rather than `2.plus(3)`:

```
interface Numeric<T>
  def plus(vararg xs : T) `+` : T
  ...

class Int
  ...
  companion object : Numeric<Int>

    def plus(vararg xs : Int) : Int
      ...
    def times(vararg xs : Int) : Int
      ...
```

To facilitate structure-polymorphic definitions, we introduce some syntactic sugar:

```
def <T : Numeric> double(x : T)    )  def <T> double(companion T : Numeric<T>, x : T)
  x + x                            )    x + x
```

Here, the function `double` implicitly gets an eponymous companion object for the type `T`, which provides the operator `T.plus` for values `x : T`.

Interfaces for companion objects are called type classes and represent mathematical structures:

```
structure Monoid<self M>(val compose : (vararg xs : M)-> M)
  val unit = compose()     # Unit is the nullary composition

  contracts {
    unit ‹compose› x = x
    x ‹compose› unit = x
    x ‹compose› y ‹compose› z = x ‹compose› (y ‹compose› z)
    compose(x, *xs) = x ‹compose› compose(*xs)
  }
```

Operations provided by type classes can be (re)named using a Fortress-inspired notation:

```
def <T : Monoid(::(∘))> square(x : T)      ⟩  def <T : Monoid> square(x : T)
  x ∘ x                                    ⟩    T.compose(x, x)
```

Operator-carrying companion objects of polymorphic types require higher-kinded parameters:

```
class List<T>
  ...
  companion object : Functorial<List>
    def <X, Y> List<X>.map(f : (X)-> Y) : List<Y>
```

There, the companion object extends `Functorial<List>`. The parameter of this type class is not a type, but a type former: `Functorial<T : * → *>`. We propose modeling support for higher kinds, type class inheritance, and nominal subtyping after the Arend[2] language, with extensions inspired by Fortress.

Type class subtyping should soundly represent hierarchies of algebraic structures, which leads to quite intricate cases at times. Let us illustrate the "Fortress-inspired extensions" with the example of rigs, which are challenging for the following reasons:

Diamond problem Rings and rigs (rings without negation) extend monoids in two ways: both form a monoid with respect to both addition `(+)` and multiplication `(·)`.

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 $\mathbb{N}$ (ring as an algebra over $\mathbb{Z}$, abelian group as a module over $\mathbb{Z}$, etc), which should be ideally reflected by subtyping.

Fortress resolves the diamond problem with renaming inheritance and allows circularity by allowing nominal (bi-)convertibility to be established retroactively:

```
structure Rig<self R> <: Monoid<R>(::(·)), AbMonoid<R>(::(+))
  ...
structure Module<R : Rig><self M> <: AbMonoid<M>(::(+))
  ...
typealias Algebra<R : Rig><self A> = Monoid<A>(::(·)) within Module<R>

establish
  Module<ℕ>  <:> AbMonoid(::(+))
  Rig        <:> Monoid(::(·)) within Module<ℕ>
  Algebra<ℕ> <:> Rig
```

[2] https://arend-lang.github.io/assets/lang-paper.pdf

4.1 Derived instances

Whenever the parameter type `T` comes with an order, the type of lists `List<T>` has a natural lexicographic ordering. It's a typical case of "derived type class instance". We propose declaring it as an additional companion object, e.g.

```
companion object List<T : Ord> : Ord
  ...
```

4.2 Using non-default type class instances

```
def <T : Ord> List<T>.sorted() : List<T>

def <T : Ord> desc : Ord<T> {
  def compare(x y : T) = T.compare(y, x)
}

val x : List[Int] = ...
val y = x ▸sorted
val z = x ▸sorted<desc>
```

5 Geometric notation

For the conventional representation of geometric constructions and proofs in a Kotlin-based language, we need syntactic sugar that stands apart from everything else. We propose to use the `import SegmentsNotation` flag to allow delimiterless strings of point labels as names for segments, angles, and polygons. That is, strings of consecutive uppercase letters, possibly with indices or apostrophes, such as `ABC`, `ABCA'`, and `X1X2` will be interpreted as `Segments(A,B,C)`, `Segments(A,B,C,A')`, and `Segments(X1,X2)`, respectively. Backticks can be used to access multi-letter uppercase identifiers: `` `ABC` ``.

6 Conclusion and outlook

Academic Kotlin addresses the unique needs of mathematics and computer science papers. **TODO**