Typeclasses for Kotlin

Alexander Kuklev[1,2] ⟨a@kuklev.com⟩

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

1 Interfaces and typeclasses

In object-oriented programming, interfaces serve as contracts that define a set of member functions that a class must implement. It works well if we only need functions that require only one argument of the said type. Whenever we need a binary operator on a type, we do this:

```
public interface Comparable<in T> {
  public operator fun compareTo(other: T) : Ternary
}

class String : Comparable<String> { … }
```

We use somewhat problematic recursive inheritance for mimicking self-types and represent highly symmetric binary operators (such as `x + y` and `x == y`) asymmetrically as methods of the left operand. Moreover, there is no way to express that `List<T>` implements `Comparable` whenever `T` does or to abstract the `map` function for collections. Typeclasses solve these issues:

```
data class Comparable<this T>(val compare: (T, T)-> Ternary)

class String {
  …
  companion object : Comparable { x, y -> … }
}
```

Here `Comparable` is not an ancestor of `String` itself but of its companion object. Instead of ugly `a.compareTo(b)`, we have a nice symmetric `String.compare(a, b)`. We use the `this` to mark the “self type” parameter of `Comparable` to be able to write

```
fun <T : Comparable> List<T>.sorted() : List<T>
```

as if it were an ancestor of the type `T` itself rather than a requirement for `T` to have a companion object (also called T) extending `Comparable<T>`. Such definitions are called structure-polymorphic.

Using this machinery we can express alternative instances and “conditional inheritance”:

```
fun <T : Comparable> desc : Comparable<T> { x, y -> T.compare(y, x) }

class List<T>
  …
  companion object List<T : Comparable>: Comparable { x, y -> … }
}

listOf(1, 2, 3).sorted<desc>
```

With higher-kinded parameters we can also require collections to have the `map` function:

```
class Functorial<this F : out * -> *> {
  fun <T, R> F<T>.map(transform: (T)-> R) : F<R>
  contracts { … }
}

class List<T> : Functorial { … }
```

Kotlin approach to operators has to be adjusted accordingly, so operators can be placed inside companion objects, e.g. `Int.plus(a, b)` instead of `a.plus(b)`.

2 Call-site field renaming and fake type members

Type classes are perfectly suited to express mathematical structures, e.g.

```
data class Monoid<this M>(infix val compose: (vararg xs : M)-> M) {
  val unit = compose()     // Unit is the nullary composition
  contracts {
    compose(x) = x
    compose(*(xs + ys)) = compose(*xs) compose compose(*ys)
  }
}
```

For better syntactical support of mathematical structures, we can allow renaming their fields on the call site:

```
fun <M : Monoid(operator plus)> foo(m: M) = m + m

fun <M : Monoid(operator times)> bar(m: M) = m * m
```

3 Using `this` as field a modifier

Kotlin supports implementing interfaces by delegation `class Foo : Bar by baz`. In some cases, we want to implement an interface `Bar` by delegating to the constructor parameter `val baz : Bar`.

Similar to its usage in type parameters, we can enable `this` as a field modifier for this purpose:

```
data class RetractibleFunction<X, Y>(this val invoke : (X)-> Y,
                                          val revoke : (Y)-> X)) {
  contracts {
    revoke(invoke(x)) == x
  }
}
```

This way, `RetractibleFunction<X, Y>` implements the interface `(X)→ Y` by delegating to `invoke`.

4 Conclusion and outlook

We should assess and thoroughly discuss with Ross Tate how typeclasses compare to shape interfaces, how they interact with type inference/outference and whether they can introduce type checking undecidability or any other kind of problems.