Introspectable Type Providers for Kotlin

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

With type-safe builders, Kotlin provides a powerful infrastructure for embedded domain-specific languages, enabling accessible declarative specifications for objects and logic specific to their application domains. These DSLs can be used to synthesize data and functions, but not types. Overcoming this limitation requires type providers: compile-time functions that synthesize classes, interfaces, mixins, and type aliases. Type providers enable DSLs like embedded `SQL`:

```
val users = Db.table("users")
users.select(users.name,
             NewColumn("login") { it.account.name },
             users.age)
     .where { age > 18 and login ≠ null }
```

We outline how type providers could be integrated into Kotlin and illustrate their usefulness with relevant use cases. We argue that confusing error messages and poor debuggability commonly associated with type providers can be overcome by requiring generated code to be introspectable and fully traceable to the data and user-written code from which it was generated.

1 Compile-time expressions

Type providers must be run at compile time, so their arguments must be compile-time expressions. Kotlin already has the `const` modifier to mark compile-time constants. Currently, constants can be only primitive values or strings, but we propose[1] to extend this notion by value classes, pure functions and algebraic data types. Besides that, we need to introduce compile-time initializers for constant properties, using a syntax similar to setters and getters:

```
const val APP_ENV init() { Comptime.getenv("APP_ENV") ?: "DEV" }
```

The init blocks are only allowed to invoke methods of the respective APIs of the compiler and compiler plugins and must otherwise be pure. The `const` modifier should be allowed for function arguments and receiver types to require them to be instantiated at compile-time.

2 Nominal type providers

Nominal type providers generate anonymous classes from the data available at compile-time:

```
fun (const JsonSchema).parseJson(json : String) : JsonClass<[this]> {…}

object Db : KtDatabase<["kqdb:ourApp.prototypeDb"]> {
  this(endpoint = Config.dbEndpoint)
}
```

Here, the compiler would generate anonymous classes for a JSON object and for a database connection respectively, using their schema and compile-time available prototype. The compiler would look for definitions of the respective type providers:

```
init data class ExprCtx.JsonClass(JsonSchema schema) : this.NewClass<Any> {…}
init class ExprCtx.KtDatabase(schemaSource : String) : this.NewClass<DbBase> {…}
```

These must be compile-time executable functions. The compiler would execute them passing the context at the expression call site as `this : ExprCtx`. The function must generate a declaration of a new class or interface with matching modifiers, mapping each non-static subexpression of the generated code to the specific location in its arguments from which it was generated,

[1]see Controlling purity in Kotlin, http://akuklev.github.io/kotlin$_p$*urity.pdf*

e.g. `JsonSchema("{properties: {name: {type: ['string']}}}")` should generate `data class That(val name : String)`, while `KtDatabase(s)` would simply generate

```
class That(args) : DbBase(dbSchema, args) {
  companion object {
    const val dbSchema init() { Comptime["KtDatabasePluign"].retrieveSchema(s) }
  }
}
```

To properly understand type providers, it is crucial to realize that they produce anonymous types just like object expressions `object {…}`, and thus can only be used to inherit from, as type bounds, and as covariant ("`out`") type parameters. In the case of generated final classes (like `KtDatabase`) above, only objects can be inherited. Without such restrictions, innocent statements would lead to type mismatch errors:

```
val x : SomeTypeProvider<[c]> = SomeTypeProvider<[c]>(args)
```

Here, each call produces a new anonymous type, so the type produced by `SomeTypeProvider<[c]>` on the left does not match the type produced by `SomeTypeProvider<[c]>` on the right. Usages satisfying the restrictions above are free from such issues:

```
class Foo(args) : SomeTypeProvider<[c]>(args) {}
val x : Foo = Foo(args)

object Bar : SomeTypeProvider<[c]>(args) {}

val baz = object : SomeTypeProvider<[c]>(args) {}
```

The first two cases use the type provider as a base class for (non-anonymous) types, while the third case sets the right expectations: object expressions are already known to produce anonymous types and can be handled using `SomeTypeProvider<[c]>` in covariant positions only:

```
fun f() : SomeTypeProvider<[c]> {
  return object : SomeTypeProvider<[c]>(args)
}
```

Let us now demonstrate what the signature of embedded `SQL`-methods looks like:

```
abstract class DbBase(const val dbSchema : DbSchema, args) {
  public fun table(const name : String) : Table<[name]>

  internal init class ExprCtx.Table(name : String)
   : this.NewClass<this@DbBase.Table> {…}

  abstract inner class Table : View
  abstract inner class View {
    public fun View.select(vararg const cols : this@DbBase.ColSpec) : View<[cols]>

    internal init class ExprCtx.View(cols : this.ColSpec)
     : this.NewClass<this@DbBase.View>  {…}
    ...
  }
  ...
}
```

Type provider capabilities can be improved if we were to support abstract type members (`abstract inner class`) mixins (including type provider generated ones) modeled after Scala:

```
mixin class Foo : Bar {…}

object Baz : Foo(args) with Bar
```

3 Structural type providers

Structural type providers are compile-time generated type aliases:

```
typealias ExprCtx.ByName(name : String) : this.InvariantTypeExpr
 = when(name) {
  "Int" -> Int::class
  "String" -> String::class
  else -> Any::class
}

val x : ByName("Int") = 5
```

These functions are assumed to be pure and return the same result for the same parameters, so they cannot generate new classes; they only compute valid type expressions, which makes them quite different from other type providers in both purpose and nature. Providing type-safe signatures for `printf`-like functions is the most relevant use case for structural type providers, but there we need a special form, argument type providers:

```
vararg typealias PlaceholderTypes(fmt : String) : Sequence<ArgDecl> = {…}

fun (const String).format(vararg args : *PlaceholderTypes<[this]>) {…}
// ("It costs %4.2f, %s").format(price, username)
```

The function `PlaceholderTypes` determines the required types for the placeholders in the format string `this`, and produces matching argument declarations, which can include argument names:

```
("It costs %{price}4.2f, %{name}s").format(price = 5.99, name = username)
```

To allow position-only and name-only arguments as in Python, we can use

```
data class ExprCtx.UnnamedArgument(val type : this.InvariantTypeExpr,
                                   val optional : Boolean = false) : this.ArgDecl

data class ExprCtx.NamedArgument(val name : SimpleIdentifier,
                                 val type : this.InvariantTypeExpr,
                                 val optional : Boolean = false,
                                 val nameOnly : Boolean = false) : this.ArgDecl
```

For dependent signatures, say `fun f(cv : Canvas, p : cv.Point)`, argument types must be allowed to depend on previous arguments.

4 Conclusion and outlook

We have proposed a number of far-reaching enhancements that allow for significant boilerplate reduction and type safety improvements with compile-time metaprogramming. By introducing the other major compile-time metaprogramming mechanism of metaclasses and decorators modeled after Python, we would provide all-around metaprogramming capabilities unparalleled in compiled programming languages to the best of the author's knowledge.