On type providers for Kotlin

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

Compile-time functions that synthesize classes and interfaces on demand are commonly known as type providers. They can be used to optimize performance by precomputing some functions, improve type safety in APIs and libraries, and enable more efficient, concise, and type-safe domain-specific languages (DSLs) such as embedded `SQL`:

```
db.users.select(name, age, NewCol("login") { account.name })
        .where { age > 18 and login ≠ null }
```

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

1 Motivation and prospective syntax

Interaction with external services or components often requires describing formats twice:

- as a data class on the Kotlin side and as a JSON schema on the external side, or
- as a set of table classes on the Kotlin side and a database schema on the external side.

Redundancy of this kind is not only inconvenient, but also error-prone: the formats on both sides must be meticulously kept synchronized. Wouldn't it be better to use schemas as a single source of truth? For example, when we are dealing with databases, we usually have a prototype database for unit tests, and its schema can be used as a reference. For that, we need generative metaprogramming, a mechanism for synthesizing sources at compile time on demand.

For a simple example, let us consider the `format` function which is presently essentially untyped:

```
inline fun String.format(vararg args: Any?): String
```

A better, type-provider based definition might look as follows:

```
val String.format by Formatter<this>()
class Formatter<TEMPLATE : const String> by {/* compile-time expression */}
```

where the compile-time expression synthesizes the source code on demand, e.g.

```
class Formatter<"Total of %d items weight %4.2f kg"> {
  inline fun invoke(arg1: Int, arg2: Float): String = …
}
```

Now `format` is a well-typed precomputed inline function and can support named arguments:

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

For JSON schemas, we can define a wrapping value class to perform schema validation, a type provider generating classes from schemas, and a parser that generates objects of such classes:

```
value class JsonSchema(val source: String) { init {/* perform validation */} }
interface JsonObject<SCHEMA : const JsonSchema> by {/* compile-time expression */}
fun <SCHEMA> parseJson(json : String) : JsonObject<SCHEMA> {…}
```

Intended usage looks like this:

```
const val CONFIG_SCHEMA = JsonSchema(Resource("data/example.json").readText())

val config = parseJson<CONFIG_SCHEMA>(configFile.readText())
```

For databases, synthesis from a schema allows generating both table objects and row classes:

```
object users : IdTable("users") {
  val name    = varchar<users>("name", 32)
  val age     = integer<users>("age")
  val account = reference<users>("account", accounts)

  data class Row(val name: String, val age: Int, val account: accounts.Row)
  data class NewCol<T>(val name: String, val block: Row.()-> T) : Col<users>
}
```

The syntax above is based on the Exposed library, with the addition that we parametrize the column type by its host table to enable context-sensitive resolution:
`users.select(name, NewCol(…))` instead of `users.select(users.name, users.NewCol(…))`.

Besides this, another significant advantage is the ability to synthesize views/flows (temporary tables) on demand, which arise when using joins, selects, and group-by/having clauses.[1] In the example above, the select generates a view with an additional login column, and in the next line, this column is used as a variable inside `where(block: ResultRow.()→ Boolean)`. That is how we achieve syntax as concise as

```
db.users.select(name, age, NewCol("login") { account.name })
        .where { age > 18 and login ≠ null }
```

To avoid type identity issues, it is crucial to only use synthesized interfaces in covariant positions: as return types, types being extended or implemented, and covariant type parameters.

2 Conclusion and outlook

Type providers facilitate significant boilerplate reduction and type safety improvements, as well as enable precomputed inlineable methods for optimal performance.

In theory, it would also be consistent to allow synthesizing mixins, decorators, and even metaclasses, but we don't think it's advisable due to extreme abuse potential.

[1]We want to specially thank Oleg Babichev from Exposed team for valuable comments.