On type providers for Kotlin

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

We propose introducing a safe form of type providers – compile-time functions that synthesize interfaces and type aliases – to greatly improve the type safety of libraries and APIs, enable very sophisticated 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 Nominal type providers

When importing or exporting data, it is often necessary to describe the data format 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 a constant source of unnecessary hassle: you have to constantly make sure that the formats on both sides are synchronized. Reflection in Kotlin allows us to use annotated data classes to generate the respective JSON schemas, thus avoiding redundancy. But what if we find it more convenient 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 source files at compile-time on demand. How might this look in practice?

First, we need the ability to initialize constants not only with literals, but also with expressions that are evaluated during compilation, which might look like this:

```
const val APP_ENV = CompileTime.getenv("APP_ENV") ?: "DEV"
```

For convenience, it is useful to allow non-primitive data types to be used as constants:

```
value class JsonSchema(val source: String) {
  init {/* perform validation */}
  companion object {
    const fun fromResource(name: String) = JsonSchema(CompileTime.readResource(name))
  }
}

const val SCHEMA = JsonSchema.fromResource("schema.json")
const val DB_SCHEMA = DbSchema("jdbc:sqlite:./resources/prototypeDb")
```

Here is what the declaration of synthesized interface synthesized might look like:

```
interface JsonObject<SCHEMA : const JsonSchema> by {/* compile-time expression */}

// Usage:
fun <SCHEMA> parseJson(json : String) : JsonObject<SCHEMA> {…}

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 groupBy. 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)`.

To avoid inconsistencies, it is crucial to only allow synthesizing interfaces, not classes. Synthesized interfaces may, however, contain synthesized member/nested classes and objects, as well as a companion object which may be used to provide fake constructors. Besides, we can also allow structural type providers.

2 Structural type providers

Structural type providers are pure functions generating type expressions. Being pure functions, they must return the same result for the same parameters, so they do not synthesize new types.

```
typealias PrintfArgType(fmt : String) = when(fmt) {
  "i", "d" -> Int::class
  "e", "f", "g" -> Float::class
  else -> Any::class
}

val x : PrintfArgType("i") = 5  // Usable as types when applied to compile-time constants
```

We also propose introducing structural type providers that synthesize not just one type, but a list of argument declarations, allowing to make `printf`-like functions type-safe:

```
fun printf(const template : String, vararg args : *PrintfArgs(template)) {…}
vararg typealias PrintfArgs(template : String) = {…}
```

The function `PrintfArgs` parses the `template` and produces matching argument declarations, which can even include argument names if desired:

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

3 Conclusion and outlook

Type providers facilitate significant boilerplate reduction and type safety improvements. 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.