Type providers

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 interfaces and type aliases. Type providers enable DSLs like embedded `SQL`:

```
val users = db.table("users")
users.select(name, age, NewColumn("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. 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 Nominal type providers

Nominal type providers synthesize interface declarations using compile-time parameters:

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

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

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

To introduce type providers we'll need to introduce the concept of compile-time expressions, and allow instantiating compile-time constants by any compile-time expressions. It is precisely such constants that are typically used as parameters of type providers:

```
const val APP_ENV = CompileTime.getenv("APP_ENV") ?: "DEV"
const val DB_SCHEMA = DbSchema("jdbc:sqlite:./resources/prototypeDb")
val db = DbConnection<DB_SCHEMA>(Config[APP_ENV].dbConnString)
```

To avoid inconsistencies, it is crucial to only allow synthesizing interfaces, but no non-inner classes. Synthesizing mixins would also be feasible, should they ever be introduced to Kotlin.

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 cannot synthesize new types.

```
typealias ByName(typeName : String) : TypeExpr = when(typeName) {
  "Int" -> Int::class; "Float" -> Float::class; else -> Any::class
}

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

We also propose introducing structural type providers that synthesize argument declarations:

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

This way we can make `printf`-like functions type-safe. The function `PrintfArgs` parses the `template` and produces matching argument declarations, which can include argument names:

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

3 Implementation of SQL DSL

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

```
interface DbConnection<const val SCHEMA> {

  fun table(const name : String) : View<SCHEMA[name]>

  val users = table("users")
  ...synthesized

  inner interface View<const COLS : Array<Col>> {

    infix fun <COLS2> join(other : View<COLS2>) : View<COLS + COLS2>
    fun where(predicate : Entry.() -> Boolean) : View<COLS>
    fun orderBy(vararg const cols : Ordering) : View<COLS>
    fun groupBy(vararg const cols : Column) : View<cols + fused(COLS sans cols)>
    fun select(vararg const cols : Column) : View<cols>
    // prefixing columns for joins:
    infix fun as(const prefix : String) : View<COLS.map { prepend(prefix) }>

    data class Entry(val col1 : T1, val col2 : T2, ...synthesized)

    enum class NamedColumn(property : KProperty1<Entry, *>) : Column {
      col1(Entry::col1), col2(Entry::col2), ...synthesized

      val name = property.name
      val type = property.returnType
    }

    sealed interface Column : Col, Ordering
    sealed interface Ordering
    class Desc(col : Column) : Ordering

    class NewColumn<reified T>(val name : String,
                               val body : Entry.()-> T) : Column {
       val type = T::class
    }

    infix fun NamedColumn.as(newName : String) = NewColumn(name) {
      (this@NamedColumn).property.get(this)
    }
  }
  interface Col {val name : String, val type : KType}
}
```

4 Conclusion and outlook

We have proposed a number of far-reaching enhancements that allow for significant boilerplate reduction and type safety improvements using type providers