JOHANNES KEPLER
UNIVERSITÄT LINZ

Bachelor's Thesis

A DSL for Fluent Kotlin Compiler IR Generation

Dipl.-Ing. Dr. Markus Weninger, BSc
Institute for System Software
T +43-732-2468-4361
markus.weninger@jku.at

Student: Lorenz Bader
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: March 2025

Kotlin's growing popularity as a versatile programming language has led to the need for advanced compiler plugins that can extend its capabilities. These plugins are essential for code manipulation, allowing developers to introduce new features and optimize existing ones. However, the current process of generating Kotlin Intermediate Representation (IR) code within these plugins is often complex and repetitive, posing challenges for developers.

The complexity of the existing Kotlin compiler plugin development process can lead to a higher incidence of errors and a steeper learning curve for new developers. The verbosity of the current constructs used for IR generation hampers productivity and can obscure the underlying logic of the code generation, making maintenance and updates more difficult.

The bachelor's thesis aims to refine the Kotlin compiler plugin by creating a domain-specific language (DSL) that facilitates the generation of Kotlin IR code. The DSL will introduce helper functions and extension methods that encapsulate the complexity of the IR generation process, thereby enhancing code readability and maintainability. The student will:

- Acquaint themselves with the Kotlin compiler internals and the existing plugin's architecture.
- Develop helper functions for more intuitive referencing of compiler elements.
- Implement methods for simplified function calls on different objects and simplified string concatenation on different objects.
- Apply, test, and document the developed functions and methods within the plugin.

This thesis will benefit Kotlin developers and the broader software development community by providing a more streamlined approach to compiler plugin development. The DSL will enable faster and more reliable code generation, which can lead to the creation of more sophisticated and efficient Kotlin compiler plugins.

The student should have a foundational understanding of compiler construction, Kotlin programming, and software development principles. Through this thesis, they will further develop their skills in DSL design, Kotlin IR, and Kotlin compiler plugin development.

Modalities:
The progress of the project should be discussed at least every four weeks with the advisor(s). A time schedule and a milestone plan must be set up within the first 2 weeks and discussed with the advisor(s). It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.08.2025.

Appendix:

- Familiarize yourself with the existing plugin and Kotlin compiler internals_ such as *IrClass*, *IrFunction*, *IrProperty*, *IrField*, *IrCall*, their respective *XYZSymbol* classes and the *plugin context* object.
- Development of *Helper Functions for Element Referencing*: Find and refactor *.referenceXYZ* calls (such as *referenceFunction(…)*) with less complex functions (such as *findFunction("kotlin/io/println(String)"), findFunction("kotlin/io/StringBuilder.append(Int)")* or *findClass("kotlin/io/MyClass")*). Also create extension functions for class IR nodes in a similar fashion (such as *stringBuilderClass.findFunction("append(Int)")*, where *stringBuilderClass* was previously found using *findClass*). Replace all *referenceXYZ* calls in the existing plugin with the newly introduced (and easier to use) *findXYZ* calls.
- When working with types, you must differentiate between built-in and non-built-in classes and manage nullability in types (e.g., *Int* vs. *Int?*, *my.package.Person* vs. *my.package.Person?*).

  Guidance (Pseudo Code):

```
when (typeStr) {
  "Int" -> pluginContext.irBuiltIns.intType
  "Long" -> pluginContext.irBuiltIns.longType
  else -> findClass(typeStr) // typeStr could be "my.package.Person?"
}
```

- Handling varargs of various kinds of elements (String literals, fields, properties, function calls, ...) requires you to create the correct corresponding IR nodes based on the element type.

  Guidance (Pseudo Code):

```
fun concat(vararg parameters: Any): IrExpression {
  return irConcat().apply {
    for (parameter in parameters) {
      when (parameter) {
        is String -> addArgument(irString(parameter))
        is IrField -> addArgument(irGetField(null, parameter)) // assuming field contains a string,
        otherwise a .toString() call would also have to be generated, see else branch
        is IrProperty -> addArgument(irCall(parameter.getter!!)) // same as above
        is IrFunction -> addArgument(irCall(parameter)) // same as above
        else -> addArgument(irToString(parameter))
      }
    }
  }
}
```

- *Facilitating Function Calls on Various Objects*: Implement methods to call functions on objects stored in different locations such as fields or properties. Also enable fluent calling of functions on objects returned from other function calls. For this, develop extension methods for *IrProperty*, *IrField*, *IrCall*, etc., to simplify function invocation. For example, have a look at how the _bufferedTraceFileSink field is generated in the compiler plugin. The IR code there generates the code

```
_bufferedTraceFileSink  = SystemFileSystem.sink(Path(bufferedTraceFileName)).buffered()
```