Simplifying Kotlin Compile-Time Code Instrumentation

Lorenz Bader
Institute for System Software
Johannes Kepler University Linz, Austria

Markus Weninger
markus.weninger@jku.at
Institute for System Software
Johannes Kepler University Linz, Austria

Abstract

Since Kotlin can be translated for different targets such as the JVM, WebAssembly, or Native, traditional load-time instrumentation would have to be re-implemented for each such compilation target. A target-agnostic solution is to instrument code at the compiler's Intermediate Representation (IR) level. However, the official compiler API for this is low-level and verbose, posing a significant barrier. We present *KIRHelperKit*, a library that simplifies Kotlin compile-time code instrumentation. It drastically reduces the number of lines of code needed to develop custom multi-platform performance profilers that work seamlessly across all Kotlin targets, significantly reducing development effort.

1 Introduction

With Kotlin's rising usage across multiple platforms [4], new challenges for performance instrumentation arise. Traditional run-time instrumentation techniques, such as those based on Java bytecode [6], have demonstrated that expressive and efficient load-time instrumentation is possible for JVM programs. However, these approaches are tied to JVM bytecode and cannot be applied across Kotlin's diverse compilation targets such as JavaScript or WebAssembly.

Compile-time instrumentation via Kotlin's Intermediate Representation (IR) compiler plugins offer a target-agnostic solution. By injecting measurement code during compilation, IR plugins avoid run-time dependencies and support consistent tracing across platforms. However, the Kotlin IR API exposes significant complexity due to its low-level abstractions and limited documentation, rendering even basic tasks such as generating function calls error-prone and difficult to accomplish. These difficulties have been noted in prior research, which identifies the steep learning curve as a key barrier to adoption [5].

To overcome these challenges, this work introduces the *KIRHelperKit*, a utility library simplifying IR plugin development. Our contributions are: (1) a structured and well documented abstraction layer for common IR tasks; (2) a rewritten version of an existing tracing plugin using our DSL, showcasing code reduction; and (3) a small usability study demonstrating improved readability and developer experience.

2 The Kotlin Compiler and its Plugins

This section outlines Kotlin's compilation process and discusses its connection to IR plugin development.

Compilation Process Figure 1 depicts Kotlin's K2 compiler. It transforms source code into a frontend intermediate representation (FIR) primarily used for static code and control flow analysis. FIR is then transformed to backend Intermediate Representation (IR) for code optimization and finally translated to the respective target (e.g., JVM, JavaScript, Native) [2].

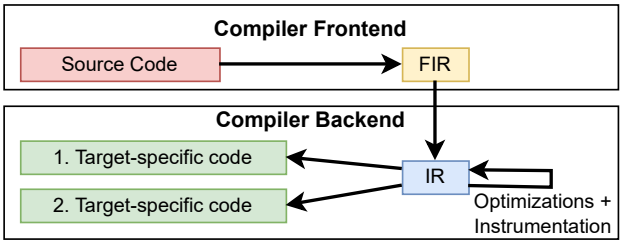

Figure 1: Overview of the Kotlin compilation process.

IR (Backend) Plugins Compiler plugins (either working on the FIR or IR) can be used to perform additional operations at compile-time. However, it is not possible to freely modify code in a FIR plugin without using annotations in the user code [8]. In contrast, (backend) IR plugins act as the last step before target-specific code generation. They have access to the complete, low-level constructs making up the compiled program, making them the perfect tool for target-agnostic instrumentation [9].

IR plugins extend the *IrGenerationExtension* interface, which allows access to all compiling Kotlin files and enables injection of instrumentation code. Norman [3] provides a comprehensive tutorial for getting started with IR plugin development. However, backend compiler plugins still have some limitations. They can only modify Kotlin source code, thus preventing the instrumentation of Java source code or pre-compiled dependencies.

3 KIRHelperKit: IR Instrumentation

Instrumentation is often used to collect performance metrics such as method run times. This section intro-

KIRHelperKit (tagged version of 02.08.2025) can be found at
`https://github.com/NeonMika/k-perf/tree/SSP_2025_KIRHelperKit`

duces our *KIRHelperKit* (Kotlin IR Helper Kit) and outlines how it simplifies common IR instrumentation tasks through high-level abstractions.

Feature Overview Developing our library, we focused on common instrumentation actions, including: (1) Simplified search for classes, functions, properties and constructors; (2) Easier generation of function calls and call chains; (3) *stringBuilder* and string concatenation abstractions for fluent development; (4) Creation of global variables without boilerplate code; (5) Built-in file support to reduce complexity.

Simplified, Unified Class and Function Search Listing 1 illustrates the conventional method for retrieving IR declarations. It looks up a function using `CallableId`, using `FqName` and `Name.identifier` in conjunction with manual argument filtering.

```
val addFn =
  pluginContext.referenceFunctions(CallableId(
    FqName("java.util.concurrent.atomic"),
    FqName("AtomicInteger"),
    Name.identifier("getAndAdd")
)).single { func ->
  func.valueParameters.size == 1 &&
  func.valueParameters[0].type ==
    pluginContext.irBuiltIns.intType }
```

Listing 1: Function search using original IR API.

```
val addFn = pluginContext.findFunction(
  """java/util/concurrent/atomic/
  AtomicInteger.getAndAdd(int)""")
```

Listing 2: Function search using *KIRHelperKit*.

In contrast, Listing 2 demonstrates our approach, which employs a structured search string format for concise and unified declaration resolution:

```
findClass:       my/package/class.inner
findConstructor: my/package/class.inner(params...)
findFunction:    my/package/class.inner.func(params...)
findProperty:    my/package/class.inner.prop
```

Unused parts (such as the inner class part) can be omitted. Generics, wildcards, and nullable function parameters (using `?`) are supported. To avoid unnecessary long search strings, additional extension methods are implemented on variables, classes, etc., enabling localized searches. For example, Listing 3 finds the same `addFn` as Listing 2.

```
val atomicInt = pluginContext.findClass(
  "java/util/concurrent/atomic/AtomicInteger")
val addFn = atomicInt.findFunction(pluginContext,
  "getAndAdd(int)")
```

Listing 3: Local function search using *KIRHelperKit*.

These abstractions avoid the need to differentiate between `CallableId` (functions), `ClassId` (classes), `FqName`, `Name.identifier`, ..., which often lead to confusion in the original IR API.

Simplified Function Calling and Chained Function Calls Listing 4 shows how IR function invocation typically requires nested `irCall` expressions, constructed in reverse order, and careful receiver management. As demonstrated in Listing 5, *KIRHelperKit* enables top-down composition of chained calls using implicit receivers, reducing cognitive overhead associated with nesting and manual receiver configuration.

```
// goal: building "counter.getAndAdd(2).toString()"
val toStringFn = ... // 4 SLOCs additionally needed
val toStringCall = irCall(toStringFn).apply {
  dispatchReceiver = irCall(addFn).apply {
    dispatchReceiver =
      irGetField(null, counter)
    putValueArgument(0, irInt(2))
  }
}
```

Listing 4: Chained calls using original IR API.

```
enableCallDSL(pluginContext) {
  val toStringCall = counter.call(addFn, 2)
                            .call("toString()")
}
```

Listing 5: Chained calls using *KIRHelperKit*.

Various receiver kinds (variables, properties, results of other function calls, etc.) can be used as targets of our `.call()` helper function. These targets are automatically set as `dispatchReceiver` (i.e., the function's `this`) or `extensionReceiver` depending on the kind of function being called. Arguments are automatically converted to `IrExpression` objects (for example, the constant 2 manually had to be converted to an `IrConst` using `irInt()` in Listing 4).

Furthermore, the function to call can not only be provided as a function object (see `addFn` in Listing 5) but also as a function search string (see `"toString()"` in Listing 5). Using a search string automatically looks up the respective function in the receiver.

The DSL can be used in the most common places where call construction takes place, such as function bodies or variable initialization. It is enabled using a lambda with receiver (`enableCallDSL`).

Minifying Common Code Patterns Common patterns such as string concatenation or file operations require considerable boilerplate when using the original IR API. For example, opening and appending to a write-only file in a multi-platform fashion (through so-called *sinks*) approximately takes 30 lines of code if developed using the original IR API.

```
val irFileHandle = IrFileWriter(pluginContext,
  firstFile, "file.txt")
irFileHandle.writeData(myString)
```

Listing 6: File operation using *KIRHelperKit*.

With *KIRHelperKit*, file handling is abstracted and therefore becomes much more expressive, as shown in Listing 6. `IrFileWriter` encapsulates all necessary logic for resolving functions and constructing *sinks*, providing a simple interface for creating textual output to files. Complementarily, `IrFileReader` supports file input operations through a similar abstraction model. Although not shown in this example, we also provide a dedicated `IrStringBuilder`, which greatly simplifies text handling.

Such abstractions reduce code size and complexity, enabling developers to focus on instrumentation logic rather than low-level IR manipulation.

4 Evaluation

We evaluate the impact of *KIRHelperKit* in two ways: (1) by re-implementing an existing compiler plugin [9] using its feature set and (2) by conducted a usability study where participants were asked to implement a new small compiler plugin using *KIRHelperKit*.

Quantifying Code Reduction We compare the code length of three plugin versions: (1) The original plugin [9], (2) a version using the full helper kit, and (3) a version not using the `IRFileWriter` abstraction (to inspect possible code savings for a broader variety of compiler plugins, as much of the plugin logic involves file I/O). We measured the plugins' `generate(...)` method (responsible for instrumentation), excluding complimentary setup code. Table 1 shows that *KIRHelperKit* reduces plugin code size significantly. We achieved 34% less source lines with basic utilities and less than 50% with full integration.

Plugin Version	SLOCs	Red. (%)
(1) Original	993	–
(2) Function + Find Utils	654	34.1%
(3) Full Helper Kit	483	51.4%

Table 1: Comparison of `generate(...)` method size reduction across plugin versions.

Assessing Developer Experience To assess developer experience and usability of *KIRHelperKit*, we conducted a study involving six participants[1]. Three participants had prior experience with Kotlin compiler plugins (incl. one expert), three did not. They were timed at implementing a small compiler plugin and provided feedback in a structured interview.

Participants received an *IntelliJ* project including the *KIRHelperKit*, a demo plugin, and a plugin scaffold for the study. First, an introduction to compiler plugins and IR transformations was given, followed by a presentation of the demo plugin showcasing the helper kit' capabilities. The participants were then asked to develop a similar plugin that (1) creates a global `AtomicInteger`, (2) increments it at the beginning of each method, and (3) prints its content at the end of the `main` method (i.e., printing how many functions have been called). This invovled essential features of the *KIRHelperKit* such as field and call creation, function lookup and expression insertion.

All participants completed the task successfully without asking IR-related questions. Times ranged from 2:17 to 10:30 min (mean $\sim$ 7:19 min). Participants described the *KIRHelperKit* as simple and *considerably* less verbose than native IR APIs. Utilities such as `findFunction` and `createField` were considered self-explanatory. Challenges involved knowing when to turn on the call DSL and somewhat unintuitive search strings (separating packages with slashes). Participants also named the inclusion of examples in the documentation and implementing further abstractions for common code constructs.

This verifies that the *KIRHelperKit* lowers the entry barrier for first time IR users while providing experienced developers with cleaner and faster workflows.

[1] `https://doi.org/10.5281/zenodo.16785498`

5 Related Work

JVM frameworks such as ASM, AspectJ, DiSL, Javassist, or BISM [6] provide powerful run-time instrumentation, while Kotlin tools like KSP [10] and Arrow Meta [7] support annotation-based source transformations. However, neither approach enables cross-platform, compile-time instrumentation, which *KIRHelperKit* delivers by bringing expressive abstractions to Kotlin's IR across JVM, JavaScript, WebAssembly, and native.

6 Summary and Outlook

This paper demonstrated the capabilities of the *KIRHelperKit*, a utility library that simplifies common instrumentation patterns in Kotlin IR compiler plugin development. Our evaluation displayed significant reduction in code size and a reduced entry barrier for IR plugin development, allowing novice developers to implement complex instrumentation tasks.

Nevertheless, feedback revealed room for additional simplifications. Thus, future work will focus on analyzing existing compiler plugins, identifying gaps, and introducing additional transformation capabilities, such as automated injection of instrumentation code in functions matching a specific signature [1].

References

[1] J. Seyster et al. "Aspect-Oriented Instrumentation with GCC". In: *RV*. 2010.

[2] J. Stanier and D. Watson. "Intermediate representations in imperative compilers: A survey". In: *ACM Comput. Surv.* 45.3 (2013).

[3] B. Norman. *Writing Your Second Kotlin Compiler Plugin*. `https://blog.bnorm.dev`. 2020.

[4] R. Nagy. *Simplifying Application Development with Kotlin Multiplatform Mobile*. Packt Publishing, 2022.

[5] D. Pfeffer and M. Weninger. "On the Applicability of Annotation-Based Source Code Modification in Kotlin". In: *MPLR*. 2023.

[6] C. Soueidi, M. Monnier, and Y. Falcone. "Efficient and expressive bytecode-Level instrumentation for Java programs". In: *J. STTT* 25.3 (2023).

[7] The Arrow Authors. *Arrow Meta: Functional Companion to Kotlin's Compiler*. `https://github.com/arrow-kt/arrow-meta/`. 2023.

[8] JetBrains. *FIR Plugins API*. `https://github.com/JetBrains/kotlin/blob/master/docs/fir/fir-plugins.md`. 2024.

[9] M. Weninger. "Tracing Performance Metrics in Kotlin Multiplatform Projects via Compile-Time Code Instrumentation". In: *SSP*. 2024.

[10] JetBrains. *Kotlin Symbol Processing (KSP)*. `https://github.com/google/ksp`. 2025.