Trace Debugger: Interactive Execution Trace Debugging for Java and Kotlin

Dmitrii Artiukhov[*], **Bob Brockbernd**[†], **Evgeniia Fedotova**[‡], **Nikita Koval**[†], **Ivan Kylchik**[†], **Evgenii Moiseenko**[‡], **Lev Serebryakov**[†], **Evgeniy Zhelenskiy**[‡], **and Maksim Zuev**[†]

[*]JetBrains, Germany
[†]JetBrains, The Netherlands
[‡]JetBrains, Serbia

ABSTRACT Debugging is a central activity in software development, yet remains challenging and time-consuming, especially when dealing with complex or intermittent failures. Conventional debuggers operate on snapshots of program state at specific breakpoints, lacking a complete view of the execution flow, and often require repeated re-execution to reach the point of interest. Shifting the perspective from breakpoints to execution traces opens new possibilities for debugging.

We present Trace Debugger, a plugin for IntelliJ IDEA that integrates deterministic execution traces into the standard debugging workflow. It allows developers to capture an execution trace of a Java or Kotlin program and replay it within the debugger. The plugin provides convenient tools for trace navigation, search, and filtering. In addition, recorded traces can serve as faithful artifacts of program behavior, supporting new techniques for regression testing.

KEYWORDS Interactive Debugging, Deterministic Replay, Program Analysis, IntelliJ IDEA, JVM.

1. Introduction

Many aspects of software development have seen rapid innovation in recent years, but the user experience of debugging tools has remained mostly unchanged. Conventional debuggers are built around the breakpoint model: the developer sets a point of interest, runs the program, and inspects a snapshot of the state once it is reached. This model provides only a limited view of program behavior, with little support for understanding how the current state was reached. As a result, debugging often involves repeated re-execution and manual stepping, a process that becomes even more challenging when the program behaves non-deterministically.

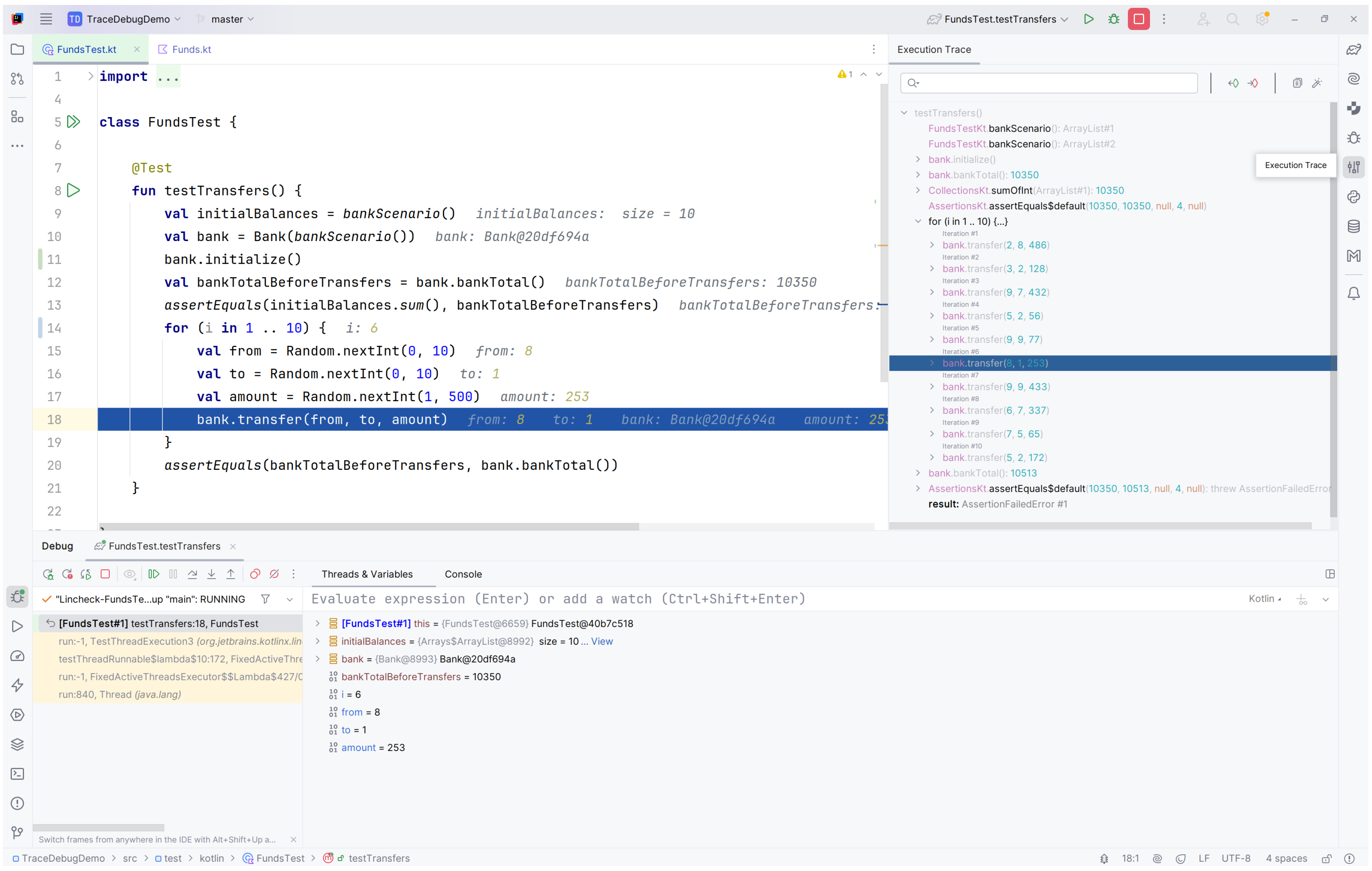
Developers need not just the current state but insight into how the program arrived there — its history, application decisions, and data flow. To obtain this information, many developers fall back on ad hoc methods such as inserting `println(..)` statements. This manual instrumentation is tedious and fragile, yet this practice persists due to a lack of better tooling. But what if, instead of manually deciding where to insert print statements, the developer had access to a structured execution trace and could choose what kinds of events to view, filter, and explore interactively?

Our Contribution. In this work-in-progress report, we present Trace Debugger — a plugin for Intellij IDEA that implements this idea. Trace Debugger allows a developer to capture a deterministic trace of a Java or Kotlin program execution and integrates the trace directly into the familiar debugger interface. Developers can navigate through the recorded trace, filter specific events, and inspect the program state at any execution point using conventional debugger features. By recording runtime events, Trace Debugger enables a new debugging workflow that makes program history observable and explorable.

2. Overview

This section presents an overview of Trace Debugger from the user's point of view. Figure 1 shows a screenshot of Intellij IDEA

JOT reference format:
Dmitrii Artiukhov, Bob Brockbernd, Evgeniia Fedotova, Nikita Koval, Ivan Kylchik, Evgenii Moiseenko, Lev Serebryakov, Evgeniy Zhelenskiy, and Maksim Zuev. *Trace Debugger: Interactive Execution Trace Debugging for Java and Kotlin.* Journal of Object Technology. Vol. 25, No. 1, 2026.
Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2026.25.1.a9



```
import ...

class FundsTest {

    @Test
    fun testTransfers() {
        val initialBalances = bankScenario()  initialBalances:  size = 10
        val bank = Bank(bankScenario())  bank: Bank@20df694a
        bank.initialize()
        val bankTotalBeforeTransfers = bank.bankTotal()  bankTotalBeforeTransfers: 10350
        assertEquals(initialBalances.sum(), bankTotalBeforeTransfers)  bankTotalBeforeTransfers:
        for (i in 1 .. 10) {  i: 6
            val from = Random.nextInt(0, 10)  from: 8
            val to = Random.nextInt(0, 10)  to: 1
            val amount = Random.nextInt(1, 500)  amount: 253
            bank.transfer(from, to, amount)  from: 8  to: 1  bank: Bank@20df694a  amount: 25
        }
        assertEquals(bankTotalBeforeTransfers, bank.bankTotal())
    }
```

```
testTransfers()
    FundsTestKt.bankScenario(): ArrayList#1
    FundsTestKt.bankScenario(): ArrayList#2
  > bank.initialize()
  > bank.bankTotal(): 10350
  > CollectionsKt.sumOfInt(ArrayList#1): 10350
    AssertionsKt.assertEquals$default(10350, 10350, null, 4, null)
  for (i in 1 .. 10) {...}
      Iteration #1
    > bank.transfer(2, 8, 486)
      Iteration #2
    > bank.transfer(3, 2, 128)
      Iteration #3
    > bank.transfer(9, 7, 432)
      Iteration #4
    > bank.transfer(5, 2, 56)
      Iteration #5
    > bank.transfer(9, 9, 77)
      Iteration #6
    > bank.transfer(8, 1, 253)
      Iteration #7
    > bank.transfer(9, 9, 433)
      Iteration #8
    > bank.transfer(6, 7, 337)
      Iteration #9
    > bank.transfer(7, 5, 65)
      Iteration #10
    > bank.transfer(5, 2, 172)
  > bank.bankTotal(): 10513
  > AssertionsKt.assertEquals$default(10350, 10513, null, 4, null): threw AssertionFailedError
    result: AssertionFailedError #1
```

```
"Lincheck-FundsTe...up "main": RUNNING
[FundsTest#1] testTransfers:18, FundsTest
run:-1, TestThreadExecution3 (org.jetbrains.kotlinx.lin
testThreadRunnable$lambda$10:172, FixedActiveThre
run:-1, FixedActiveThreadsExecutor$$Lambda$427/C
run:840, Thread (java.lang)
Switch frames from anywhere in the IDE with Alt+Shift+Up a...
```

```
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
> [FundsTest#1] this = {FundsTest@6659} FundsTest@40b7c518
> initialBalances = {Arrays$ArrayList@8992}  size = 10 ... View
> bank = {Bank@8993} Bank@20df694a
  bankTotalBeforeTransfers = 10350
  i = 6
  from = 8
  to = 1
  amount = 253
```

```
TraceDebugDemo > src > test > kotlin > FundsTest > testTransfers        18:1  LF  UTF-8  4 spaces
```

Figure 1 Trace Debugger user interface in Intellij IDEA.

with the plugin installed. The figure is a reference point throughout this section to explain the plugin's workflow, interface, and user experience.

The typical workflow begins when the user initiates a debug session using the *Debug with trace* button. This launches the program under the Trace Debugger, which executes the program while simultaneously capturing a deterministic execution trace.

Once execution completes, the captured trace becomes available for inspection and replay. The execution trace panel can be seen on the right hand side of the Figure 1. The user is returned to the standard debugger interface, where they can use familiar features such as breakpoints, program stepping, and variable inspection, alongside new capabilities unlocked by trace-based execution: trace navigation, filtering, search, and deterministic replay. The latter, in particular, allows the user to move freely both backward and forward through the trace, or jump directly to any recorded trace point.

The recorded execution trace is presented as a structured list of trace points. Each trace point corresponds to a specific runtime event, such as a method call, field access, *etc.*. For each event, relevant contextual data can be recorded and displayed: for example, method call events show the invoked method name and argument values, and field accesses show the class name, field name, and the value being read or written.

Trace elements are organized hierarchically, reflecting the dynamic structure of the program's control flow. For instance, method call events act as containers and may be expanded or collapsed to show or hide their nested events. This allows the user to focus on high-level behavior while still being able to inspect low-level details when needed.

The trace view is tightly integrated with the source editor and the debugger. Selecting a trace point moves the editor to the corresponding source location and positions the debugger at the matching execution state, enabling intuitive exploration of the execution history.

To provide this functionality, Trace Debugger relies on the ability to replay recorded traces deterministically. During replay, the program state evolves exactly as it did during the original execution, with thread scheduling and other sources of non-determinism captured in the trace. As such, once an erroneous program execution is captured by the Trace Debugger, the bug can be easily reproduced, allowing for a systematic investigation of its root cause.

3. Use Cases

Trace Debugger is designed to enhance the debugging experience by making execution traces available as a first-class artifact within the development workflow. In this section, we highlight several practical scenarios where the tool proves particularly useful.

Debugging failed tests. A core use case for Trace Debugger is helping to investigate failed unit tests by inspecting the recorded trace. For example, by navigating through the trace, the developer can check whether a particular invariant or assumption held throughout execution and identify the moment it was violated. Alternatively, when an assertion fails due to an unexpected value, the trace can be searched backwards to find the last modification to the corresponding object field. These workflows resemble dynamic program slicing techniques (Korel & Laski 1988; Agrawal & Horgan 1990). By exposing the execution trace explicitly and enabling fine-grained navigation, Trace Debugger makes such slicing-like reasoning available through a standard IDE interface.

Regression testing. Execution traces recorded by the tool can serve as behavioral snapshots of program runs. This makes them useful for regression testing during complex refactorings, where the developer wants to ensure that a given test case preserves its observable behavior. By recording a trace before the change and comparing it to the trace after the refactoring, differences in program behavior—such as changes in method call sequences or field values—can be detected even when the test still passes.

AI Agents. Execution traces can also serve as a valuable source of input for AI agents based on Large Language Models (LLMs). While source code alone provides only static information, execution traces offer insight into otherwise missing runtime data— such as control flow, variable values, and method interactions. This additional context can improve an LLM's ability to understand observed failures and suggest meaningful fixes (Zhong et al. 2024; Shi et al. 2024; Hu et al. 2024; Yuan et al. 2025).

Learn new codebase. Last, Trace Debugger can help learn new codebases, providing a way to explore how the system operates in different scenarios with an ability to observe the execution trace and study specific parts in detail to quickly understand the program logic.

4. Implementation

The architecture of Trace Debugger is composed of several components, which are shown in Figure 2. The system consists of two main parts. The first is a trace collector service, based on the Lincheck framework (Koval et al. 2023; Potapov et al. 2024). It includes the instrumentation and runtime libraries for trace capture and replay. The second part is the IntelliJ IDEA plugin, which provides the user interface, performs trace post-processing based on the IntelliJ Program Structure Interface (PSI), and establishes a communication protocol between the runtime and the debugger.

Code instrumentation. The trace collector service attaches a JVM agent to the target program and performs JVM bytecode instrumentation at class load time. It injects monitoring code to track program events such as method calls, local vaiable and field acceses, and others.

Trace collection and replay. The runtime component is responsible for trace collection and deterministic replay. To support replay, it implements a mechanism for intercepting the results and side effects of selected method calls. During the original execution, it records return values and relevant side effects. On replay, these results are substituted to ensure that the behavior matches the original execution. This functionality is implemented through a modular interface. For each kind of API that requires tracking due to non-determinism, a subclass of a generic tracking interface is provided, which defines how values and side effects should be recorded and replayed.

Currently, Trace Debugger supports deterministic replay for common sources of non-determinism in Java and Kotlin standard libraries, such as random number generation and time-related APIs. Support for I/O operations is under active development.

In addition to external APIs, the runtime also accounts for implicit sources of non-determinism within the JVM, such as class loading order and object identity hash codes. The latter arises from the behavior of `System.identityHashCode()`, whose return value depends on the JVM's internal identity mechanism and may vary across executions. To address this, the runtime memoizes the hash code at the point of object allocation and injects the recorded value during replay. This ensures deterministic behavior for code that relies on identity hash codes, preserving consistency across executions.

Finally, the runtime also handles concurrency-related non-determinism. This is achieved by capturing events related to various synchronization APIs, such as thread creation and joining, entering and exiting `synchronized` blocks, invocations of low-level primitives like `park`/`unpark`, and other thread coordination mechanisms. The runtime controls thread scheduling by enforcing a single-thread-at-a-time invariant, re-executing threads in the same order as observed during the original execution trace when replaying the execution.

Trace post-processing. The trace collected by the runtime is based on instrumentation at the level of JVM bytecode. While this enables platform-level compatibility, it also introduces a key limitation: certain high-level source code constructs are not directly represented in the bytecode, which can make the trace appear less readable or harder to relate to the original code.

To address this, the plugin performs a post-processing step that augments trace points with structural information recovered from the source code. Each trace point includes debug metadata available at the bytecode level, such as the originating file name, class name, method name, and line number. The plugin uses this information to link trace points to corresponding PSI elements, which represent the syntactic structure of the program within

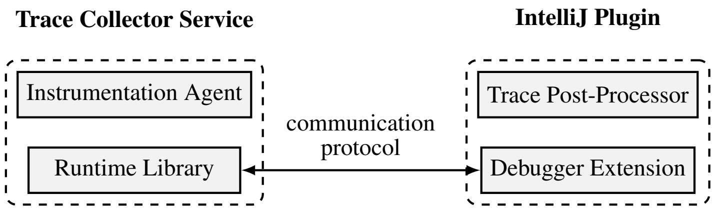


Figure 2 High-level architecture of Trace Debugger.

Intellij IDEA.

As an example of this approach, the plugin detects when multiple consecutive trace points correspond to iterations of the same loop. These entries are grouped together in the UI under a shared loop header and labeled with iteration indices. This improves trace readability and helps users relate trace behavior to familiar source constructs.

Debugger communication protocol. The communication between the debugger and the runtime is implemented via a lightweight protocol that integrates seamlessly with the Intellij debugger infrastructure. This design allows the replay session to reuse all conventional debugging features, including breakpoints, stepping, variable inspection, and expression evaluation, without requiring changes to the underlying debugger implementation.

To implement communication primitives, the protocol leverages existing capabilities of the Intellij debugger, primarily *synthetic breakpoints* and the *force return* mechanism. Whenever the runtime needs to send data to the debugger, it invokes a dedicated method call with the desired payload encoded as arguments. The debugger sets a synthetic breakpoint on this method and intercepts the call, retrieving the runtime-supplied data. To return control or pass information back, the debugger uses the force return feature to substitute a return value without executing the actual method body.

As an example of this communication protocol in action, consider how navigation to a specific trace point requested by the user is implemented. During trace collection, the runtime assigns a unique identifier to every recorded event. At replay time, whenever an event with the requested identifier is encountered, the runtime invokes a special method, `beforeEvent(eventId)`. The debugger sets a synthetic breakpoint on `beforeEvent` and instructs the runtime to enable the corresponding event ID.

5. Related Work

Record-and-replay has been a long-standing area of research, with numerous systems proposed over the years to enable reproducible debugging of non-deterministic program behavior.

The `rr-debugger` (O'Callahan et al. 2017) system is a widely used, open-source solution targeting record-replay of native applications on Linux. It achieves deterministic replay by recording sources of non-determinism such as system calls and context switches, while enforcing single-threaded execution during recording to eliminate data races. While `rr-debugger` is highly effective at the native level, its approach does not directly apply to higher-level languages that run on their own virtual machines, such as the JVM (Schwartz et al. 2024). These runtime environments may introduce additional sources of non-determinism — such as class loading order, JIT compilation, or internal caching. Such behaviors are typically benign, but if not explicitly tracked and controlled at the VM level, they may appear opaque at the system level and clutter the recorded execution trace, affecting reproducibility or performance.

Another notable example is Undo (*Reversible debugging tools for C/C++ on Linux & Android* 2018), a commercial record-replay system designed for industrial use, that integrates with popular CI/CD systems and IDEs. Similarly, to `rr-debugger`, the tool supports debugging only applications running on Linux. Notably, JVM-based languages such as Java and Kotlin are supported. Yet, since the system is proprietary, its internal implementation details are unknown.

Many previously proposed solutions, similar to Trace Debugger, use code instrumentation to achieve deterministic replay (Patil et al. 2010; Huang et al. 2010; Sahin et al. 2019; Schwartz et al. 2024).

Compared to previous work, the main distinguishing feature of Trace Debugger is its focus on exposing the structure of the captured execution trace and enhancing the debugging experience around it. The ability to collect and display various program events— such as method calls with arguments, and field reads and writes— helps developers understand both the control flow and data flow of a program at runtime. Record-replay functionality supplements this workflow by enabling navigation to arbitrary trace points and inspection of the corresponding program state.

6. Conclusion and Future Work

Trace Debugger demonstrates how integrating execution traces into the debugging workflow can unlock new possibilities. Developers gain access to a rich source of runtime information, trace-assisted navigation, and regression tracking. We believe that trace-based debugging has great potential to complement traditional breakpoint-based techniques in many scenarios, especially where understanding control- and data-flow execution history is essential.

At the current stage, we envision Trace Debugger as being primarily intended for debugging unit tests and small-scale programs. By concentrating on this domain, we can tolerate certain performance overheads, such as full replay from the beginning of the trace.

In future work, we aim to mitigate these limitations in two ways. First, by incorporating known optimization techniques such as state snapshotting. Second, by providing fine-grained control over what events are collected, with support for dynamic adjustment of trace resolution. This control could be exercised manually by the developer or guided by AI-based analysis that suggests relevant program points for trace collection based on the information about the error being debugged.

References

Agrawal, H., & Horgan, J. R. (1990). Dynamic program slicing. *ACM SIGPlan Notices*, *25*(6), 246–256.

Hu, X., Kuang, K., Sun, J., Yang, H., & Wu, F. (2024). Leveraging print debugging to improve code generation in large language models. *arXiv preprint arXiv:2401.05319*.

Huang, J., Liu, P., & Zhang, C. (2010). Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth acm sigsoft international symposium on foundations of software engineering* (pp. 207–216).

Korel, B., & Laski, J. (1988). Dynamic program slicing. *Information processing letters*, *29*(3), 155–163.

Koval, N., Fedorov, A., Sokolova, M., Tsitelov, D., & Alistarh, D. (2023). Lincheck: A practical framework for testing concurrent data structures on jvm. In *International conference on computer aided verification* (pp. 156–169).

O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., & Partush, N. (2017). Engineering record and replay for deployability. In *2017 usenix annual technical conference (usenix atc 17)* (pp. 377–389).

Patil, H., Pereira, C., Stallcup, M., Lueck, G., & Cownie, J. (2010). Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual ieee/acm international symposium on code generation and optimization* (pp. 2–11).

Potapov, A., Zuev, M., Moiseenko, E., & Koval, N. (2024). Testing concurrent algorithms on jvm with lincheck and intellij idea. In *Proceedings of the 33rd acm sigsoft international symposium on software testing and analysis* (pp. 1821–1825).

Reversible debugging tools for C/C++ on Linux & Android. (2018). Retrieved from https://undo.io/ (Accessed: 2024-07-01)

Sahin, O., Aliyeva, A., Mathavan, H., Coskun, A., & Egele, M. (2019). Randr: Record and replay for android applications via targeted runtime instrumentation. In *2019 34th ieee/acm international conference on automated software engineering (ase)* (pp. 128–138).

Schwartz, D., Kowshik, A., & Pina, L. (2024). Jmvx: Fast multi-threaded multi-version execution and record-replay for managed languages. *Proceedings of the ACM on Programming Languages*, *8*(OOPSLA2), 1641–1669.

Shi, Y., Wang, S., Wan, C., & Gu, X. (2024). From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.

Yuan, X., Moss, M. M., Feghali, C. E., Singh, C., Moldavskaya, D., MacPhee, D., … others (2025). debug-gym: A text-based environment for interactive debugging. *arXiv preprint arXiv:2503.21557*.

Zhong, L., Wang, Z., & Shang, J. (2024, August). Debug like a human: A large language model debugger via verifying runtime execution step by step. In L.-W. Ku, A. Martins, & V. Srikumar (Eds.), *Findings of the association for computational linguistics: Acl 2024* (pp. 851–870). Bangkok, Thailand: Association for Computational Linguistics. Retrieved from https://aclanthology.org/2024.findings-acl.49/ doi: 10.18653/v1/2024.findings-acl.49

About the authors

Dmitrii Artiukhov Dmitrii Artiukhov is a software developer at Debugging Research Lab at JetBrains Research. You can contact the author at dmitrii.artiukhov@jetbrains.com.

Bob Brockbernd Bob Brockbernd is a software developer at Debugging Research Lab at JetBrains Research. You can contact the author at bob.brockbernd@jetbrains.com.

Evgeniia Fedotova Evgeniia Fedotova is a project manager in Software Engineering Research department at JetBrains Research. You can contact the author at evgeniia.fedotova@jetbrains.com.

Nikita Koval Nikita Koval is a lead of Debugging Research Lab at JetBrains Research. His research interests include program debugging, program analysis, and concurrent programming. You can contact the author at nikita.koval@jetbrains.com.

Ivan Kylchik Ivan Kylchik is a software developer at Kotlin compiler team at JetBrains. You can contact the author at ivan.kylchik@jetbrains.com.

Evgenii Moiseenko Evgenii Moiseenko is a software developer at Debugging Research Lab at JetBrains Research. He is interested in programming languages semantics, formal verification, and novel approaches to program debugging. You can contact the author at evgeniy.moiseenko@jetbrains.com.

Lev Serebryakov Lev Serebryakov is a software developer at Debugging Research lab at JetBrains Research. You can contact the author at lev.serebryakov@jetbrains.com.

Evgeniy Zhelenskiy Evgeniy Zhelenskiy is a software developer at Kotlin compiler team at JetBrains. You can contact the author at evgeniy.zhelenskiy@jetbrains.com.

Maksim Zuev Maksim Zuev is a software developer at Profilers and Debuggers team at JetBrains. You can contact the author at maksim.zuev@jetbrains.com.