Interactive Development using the Dotty Compiler (Tool Paper)

Guillaume Martres
EPFL, Switzerland
guillaume.martres@epfl.ch

Abstract

A programming language is only as good at its tooling. Traditionally, tooling has always been an after-thought of language design since developing good tools take significant development efforts. Dotty is an experimental compiler for what will one day be called Scala 3, and Scala already has established and functional tooling. For Dotty to be seen as a viable alternative to Scala 2, it needs to deliver a developer experience at least as good. In particular, good support for Integrated Development Environments (IDEs) is crucial. In this paper we report our progress on providing IDE support for Dotty.

CCS Concepts • **Software and its engineering → Integrated and visual development environments**; *Incremental compilers*;

Keywords Scala, Dotty, Interactive Development Environment, Language Server Protocol

ACM Reference Format:
Guillaume Martres. 2017. Interactive Development using the Dotty Compiler (Tool Paper). In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3136000.3136012

1 Introduction

Three projects represent the state of the art in Scala IDE support:

- The Scala plugin for IntelliJ IDEA[1], which contains its own implementation of a typechecker for Scala tied to IDEA's internal data structures. The custom typechecker means that the plugin does not always reflect the behavior of the real compiler, but allows the IDE to provide powerful refactoring features based on its understanding of the code.
- The Scala IDE plugin for Eclipse[2] which communicates with the compiler to get semantic information.
- ENSIME[3], which provides both a server that communicates with the compiler and plugins for several editors (Emacs, Vim, Sublime, ...) that communicate with the server using a custom protocol.

Unfortunately, none of these solutions can easily be integrated with Dotty: both the Scala IDE and ENSIME are too tied to the internals of the Scala 2 compiler, and adapting the IDEA plugin would mean reimplementing the semantics of the Dotty typechecker and maintaing them as Dotty continues to evolve. This is why we developed our own solution for IDE support from scratch, starting from the following requirements:

- It should be based on primitives that can be reused to develop other interactive tools, like REPLs.
- It should not be tied to a specific editor.
- It should be as easy to install and use as possible.

This lead us to develop the following components:

- A set of APIs in the compiler designed for interactive usage, described in Section 2.
- The Dotty Language Server, an implementation of the Language Server Protocol[4], described in Section 3.
- A plugin for the Visual Studio Code IDE[5] to launch the Dotty Language Server, as well as a plugin for the sbt build tool[6] to configure and run an IDE, described in Section 4.

2 Using Dotty as an Interactive Compiler

Until now, Dotty has been mainly used as a batch compiler: given a set of files as input it will attempt to compile them and then exit. However, it has been designed from the start for more interactive use-cases where the same compiler instance is reused for multiple compilation attempts. Each compilation attempt is called a *run* and symbols defined in previous runs are accessible in the current run. In

[1] https://github.com/JetBrains/intellij-scala

[2] http://scala-ide.org
[3] http://ensime.org
[4] https://github.com/Microsoft/language-server-protocol
[5] https://code.visualstudio.com
[6] http://www.scala-sbt.org

SCALA'17, October 22–23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5529-2/17/10…$15.00
https://doi.org/10.1145/3136000.3136012

a REPL, a new run would be created everytime the user presses "Enter"; in an IDE, a new run could be created for every keystroke. To expose this functionality, we have developed a set of high-level APIs under the namespace `dotty.tools.dotc.interactive`.

Besides handling the life-cycle of an interactive compiler, the interactive APIs provide convenience methods for implementing typical IDE features such as auto-completion and "go to definition", these methods work by traversing typed Abstract Syntax Trees or *typed trees* for short.

2.1 Querying the Compiler Through Typed Trees

During batch compilation, the source code is parsed to produce untyped trees which are then desugared and typechecked. The typed trees then go through a series of type-preserving transformations that progressively simplifies them until they can be easily translated to a backend-specific format such as JVM bytecode. Interactive compilation reuses the same pipeline but stops the compiler after typechecking. This is enough to report most of the compiler errors and warnings[7] with a minimum amount of latency, but more importantly, this is exactly what we need to implement interactive features: each tree node has a type containing the required semantic information and since the trees have not been simplified yet, it is easy to translate back and forth between a source code position and the corresponding tree node(s).

2.2 TASTY: a Serialization Format for Typed Trees

Using typed trees as the universal interface to ask questions about code opened in the IDE seems convenient since typechecking the user code is required to report errors, but what about files which are not currently open in the IDE? Typechecking the whole project in the background and keeping the corresponding trees in memory does not scale to large projects, and building an index would mean giving up on typed trees as the sole source of truth about the semantics of the code. Instead, we take advantage of another feature of the compiler: TASTY [Odersky et al. 2016] is a serialization format for typed trees. Everytime a class is compiled by Dotty, the trees after typechecking are not discarded but serialized and stored in a dedicated section of the emitted classfile. The original motivation behind TASTY is the desire to solve the binary compatibility problem [Odersky 2014] by providing an interchange format more stable than JVM bytecode and more resilient to compiler changes than source code, but TASTY is also exactly what we need to provide interactive features across one or more projects: instead of re-typechecking everything, we can simply deserialize the stored trees and treat them exactly like source-derived typed trees. In practice, our work on the IDE required changing TASTY to carefully record the positions of all trees, something which wasn't necessary for its original purpose but is critical for interactive features such as symbol renaming to work correctly. From the point of view of a user of the interactive API such as the IDE, the use of TASTY is transparent: the same methods are used to retrieve and query trees whether they come from source code or from TASTY.

[7] The IDE should also run the full compiler in the background to catch errors occuring later in the pipeline. Our implementation does not yet handle that.

3 The Language Server Protocol

3.1 The IDE Portability Problem

While the interactive APIs described in the previous section ease the work needed to implement an IDE, they do not solve the main issue in IDE-compiler integration: traditionally, getting m IDEs to support n programming languages has required the development of $m \times n$ IDE plugins, each taking significant development effort. This has been dubbed the *IDE portability problem* [Keidel et al. 2016]. Microsoft's Language Server Protocol (LSP) is a recent attempt at solving this problem: it defines a protocol for an IDE to communicate with a *language server*.

Language servers implement the language-specific logic necessary to provide IDE features such as auto-completion and go to definition. We only need to develop m IDE plugins to support the LSP and n language server, for a total development effort of $m + n$ implementation artifacts[8]. This also results in a clear separation of work: IDE plugin developers do not need to understand language implementation details and language developers can expose their language to the outside world without knowing anything about IDE internals.

3.2 Basics of the LSP

The LSP is a JSON-RPC based protocol that an IDE and a language server can use to communicate via messages which can either be notifications, requests or responses. The IDE notifies the language server about user actions like file opening and editing. Based on this, the language server can maintain an up-to-date internal representation of the user code and notify the IDE about compilation errors and warnings. The IDE may also send requests like "go to the definition of the symbol at point" or "find all references to the symbol at point" usually triggered by user actions. The protocol is fully asynchronous and multiple requests can be sent before a single response has arrived, requests can also be cancelled.

3.3 Limitations

The LSP is not mature yet:

- Many IDEs do not have LSP implementations yet, see http://langserver.org for a list of implementations.

[8] In practice, a small language-specific plugin is still needed to let the IDE know which file types should be handle by which language server, and how to start the language server. In Section 4 we briefly describe such an extension that we developed for Visual Studio Code.

- The protocol only defines one kind of refactoring: renaming symbols
- Some of its supported queries are not precise enough: when renaming a definition in an object-oriented language, should overriden definitions be renamed too? Currently, the protocol offers no way of signaling this information.

3.4 Implementing a Language Server for Dotty

The Dotty Language Server (DLS) is based on the Eclipse LSP4J library[9] which provides a simple type-safe API that takes care of the low-level details like marshalling and keeping track of the correspondence between requests and responses. When initialized, the DLS parses the configuration file `.dotty-ide.json` at the root of the project that should have been produced by the build tool (as described in Section 4), based on this it creates a compiler instance for each sub-project (different sub-projects can have different dependencies and use different compiler flags, so cannot be handled by the same compiler instance). The implementation of the protocol messages is rather straight-forward since most of the implementation complexity lies in the compiler itself and in the APIs from Section 2. As an example, the following code implements the response to a query asking for the source code position of the definition corresponding to the symbol at a given position, which is exactly the information needed to implement "go to definition":

```
override def definition(params: TextDocumentPositionParams) = // 1.
  computeAsync { cancelToken => // 2. // 3.
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri) // 4.
    implicit val ctx = driver.currentCtx // 5.

    val pos = sourcePosition(driver, uri, params.getPosition) // 6.
    val uriTrees = driver.openedTrees(uri) // 7.
    val sym = Interactive.enclosingSourceSymbol(uriTrees, pos) // 8.

    val classTree =
      SourceTree.fromSymbol(sym.topLevelClass.asClass).toList // 9.
    val defTree = Interactive.definition(classTree, sym) // 10.
    defTree.map(d => location(d.namePos)).asJava // 11.
  }
```

Let us go over this example in details to get a better understanding of the whole system:

1. LSP4J insulates us from the message-passing details, instead we only need to override a few methods like `definition`.
2. `computeAsync` creates a cancellable closure that will be scheduled by LSP4J.
3. `cancelToken` is a callback provided by LSP4J used to interrupt and cancel the current closure if the corresponding request has been cancelled by the IDE. Our implementation currently never calls `cancelToken` because we haven't added safe points for interruption inside the compiler. This means that the language server may take more time than necessary to answer requests (since it could be busy answering old cancelled requests) but since responses are asynchronous this should never block the UI of the IDE.
4. Files are identified by URIs, given such an URI we can retrieve the corresponding compiler instance created during the initialization of the DLS.
5. A compiler instance has a `Context` representing its internal state, we make the current one available implicitly since it will be needed for the semantic operations done below.
6. The current position in the source is translated from its LSP4J representation to its Dotty compiler representation.
7. All typed trees found in the current file are retrieved.
8. The `Interactive` object defines a set of high-level methods to retrieve information from trees, it is part of the APIs we described in Section 2. It is used here to find the symbol `sym` corresponding to the current position.
9. The tree of the class where `sym` is defined is then retrieved. If this class is in a file that is not opened in the IDE, the tree will come from TASTY without the DLS having to handle it specifically.
10. The tree of the class is traversed to find the tree node where `sym` is defined.
11. Finally, the position of the definition tree node is returned after being translated to its LSP4J representation.

4 Tying Everything Together: Build Tool and IDE Integration

No one likes following a complex set of instructions to setup their tools, therefore we strived to make the Dotty IDE integration as easy to install as possible, we ended up reducing the instructions to two steps:

1. Install Visual Studio Code.
2. In your project, run: `sbt launchIDE`.

Compiling Dotty code with sbt requires using the `sbt-dotty` plugin. By adding IDE-specific commands to this plugin, we made it possible for users to use the IDE support without having to install and configure another plugin. The `sbt launchIDE` command does the following:

1. The sbt build is analyzed to find projects that compile with Dotty (the build might contain projects which only compile with Scala 2, those will be ignored).
2. All the Dotty projects are compiled, this is important since the IDE relies on the presence of TASTY which is part of the compiler output.
3. A `.dotty-ide-artifact` file is generated containing the name of the artifact to fetch and run the DLS.

[9] https://github.com/eclipse/lsp4j

4. A `.dotty-ide.json` file is generated containing enough information for the DLS to start a compiler instance for each project and to match each file opened in the IDE with the corresponding compiler instance.
5. The Visual Studio Code extension for the DLS is installed if it's not present
6. Visual Studio Code is started.

At this point, the Visual Studio Code extension should take over. It only has one job: read `.dotty-ide-artifact` and use that to fetch and launch the DLS. The fetching part is not trivial since the DLS has dependencies on other projects that also need to be fetched, the extension delegates this to the artifact fetching tool Coursier [10].

4.1 Support for Other IDEs

`sbt launchIDE` is currently hard-coded to run Visual Studio Code because it features the most complete implementation of the LSP, but we are also interested in supporting other editors. The effort required to do so is minimal: our Emacs extension [11] is only 20 lines of code, it is however not recommended for users currently because the Emacs LSP support is not yet mature.

4.2 Support for Other Build Tools

sbt is the only build tool that can be used to build Dotty projects currently, but others will probably follow. Making them work with the Dotty IDE support should be easy: they only need to generate a `.dotty-ide-artifact` (used by the IDE extension to start the DLS) and a `.dotty-ide.json` (used by the DLS to start compiler instances).

5 Future Work

Our implementation is now usable but work remains to be done to provide a good developer experience:

- **Features** The most important missing feature currently is documentation look up, which we may implement by storing the API documentation text in TASTY. We are also interested in supporting case-splitting and other features allowing for interactive type-driven editing as popularized by dependently-typed languages such as Idris [Brady 2017]
- **Optimizations** Our implementation is suboptimal in many way: it loads too many classes in memory, it cannot interrupt cancelled requests and it does not do targeted typechecking (meaning that every keystroke in a file ends up forcing the recompilation of the whole file). These improvements won't be trivial to implement but should fit in our existing design.
- **Build tool integration** The DLS assumes that all closed source files in the project have a corresponding up-to-date classfile on the classpath, but this is not enforced. If the DLS ran its own instance of sbt, it could continuously do incremental compilations of the whole project in the background, but communicating with an sbt instance is not easy currently. We hope to take advantage of the ongoing work on adding a server mode to sbt [Yokota 2016] allowing communication through a specified protocol.
- **Testing** So far, we have relied on manual testing but real tests will be necessary to avoid regressions, we plan to repurpose some of the testing infrastructure and tests from the Scala 2 compiler.

6 Conclusion

By developing a fully functional IDE experience for Dotty, we have demonstrated the suitability of the compiler design for interactive tooling. The total development effort was reasonable (the complete set of components took a couple of man-months to implement), and thanks to the use of the Language Server Protocol, the result is not tied to a single IDE.

Having working IDE support early in the development of the compiler means that it can drive the compiler evolution: it has already influenced the set of information stored in the TASTY serialization format, but it also offers the opportunity to design language features and the way developers may interact with them hand-in-hand.

[10] https://github.com/coursier/coursier
[11] https://github.com/smarter/emacs-lsp-dotty

References

Edwin Brady. 2017. *Type-driven development with Idris*. Manning Publications Co, Shelter Island, NY.

Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. 2016. The IDE portability problem and its solution in Monto. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 152–162.

Martin Odersky. 2014. The Binary Compatibility problem. https://www.slideshare.net/Odersky/scalax. (2014).

Martin Odersky, Eugene Burmako, and Dmytro Petrashko. 2016. *A TASTY Alternative*. Technical Report. https://infoscience.epfl.ch/record/226194

Eugene Yokota. 2016. sbt server reboot. http://eed3si9n.com/sbt-server-reboot. (March 2016).