Making Dotty more robust

Author: Guillaume Martres
Supervisor: Martin Odersky

January 2017

Introduction

Dotty is a new, experimental compiler for Scala. As with all compilers it will need to be thoroughly used and abused before becoming mature. This will take time, but we can try to speed up the processus by continuously pushing the compiler to its limit. This report describes two approaches we used to achieve this, first we discuss our new compiler bootstrapping mechanism, then we describe how we tested the interactive mode of the compiler by implementing IDE support.

1 sbt-based compiler bootstrap

Compiler bootstrapping is the act of using the compiler to compile itself. For Dotty, this used to be done using the side-effects of unit tests and some classpath hacks but this had several disadvantages:

- The bootstrapping mechanism was extremely fragile, any significant refactoring of the build or the repository broke it silently.
- It was mostly unmaintained and broken for several months before it was removed.
- It was designed to run partest and nothing else, this made it hard to use it for one-off testing, for interactive uses like running the REPL, for practical tests of incremental compilation, etc.
- Eventually, we would have needed to publish it online, this would have involved manually replicating what the `publish` command of sbt already does.

The new bootstrapping mechanism takes advantage of our existing build tool, sbt, instead of repurposing our test suite as build tool. To do this, we simply created two new sbt projects `dotty-library-bootstrapped` and `dotty-compiler-bootstrapped` that contain the same files as `dotty-library` and `dotty-compiler` but compile them using the (non-bootstrapped) Dotty instead of `scalac` (this is possible thanks to previous work we did to get sbt to compile projects using Dotty). This new bootstrap lead to the discovery of several long-standing issues in Dotty because the Dotty test infrastructure had never been compiled by Dotty before, and the JUnit tests had never been run by a bootstrapped Dotty before. Some of these issues were fixed and others were simply worked around since getting a working bootstrap in place was considered important enough. We won't describes all these issues here (see #1896 for the details), but we will focus on one of them as an example.

1.1 Semantics of recursive `lazy val` (#1856)

What should happen when a lazy val is recursively called during its own initialization? According to the specification, any behavior is acceptable:

> Attempting to access a lazy value during its initialization might lead to looping behavior.

In practice, `scalac` allows a lazy val to be called recursively, each recursive call will end up reinitializing the lazy val, which means that multiple values can be observed for the same lazy val. On the other hand, Dotty will return `null` (or `0` for primitives) for any recursive call. Since Dotty itself relied on this behavior, a few lazy vals had to be rewritten as getters-with-cache:

```
// Before
lazy val x: String = computeX()
// After
def x: String = {
  if (myX == null) {
    myX = computeX()
    assert(myX != null)
  }
  myX
}
private[this] var myX: String = _
```

This solved the immediate problem but this issue will need to be revisited in the future if we want non-nullable types to be the default in Dotty. The most likely fix would be to thrown an exception instead of returning `null`, but this requires tracking the initialization status of lazy vals, which may have a significant impact on performance and bytecode size.

1.2 Future work

- While the sbt-based bootstrap now works perfectly well for testing purpose, it is not ready for publication: the published artefacts would have `-bootstrapped` in their name. Fixing this is non-trivial for the same reason that it is non-trivial in the sbt build of scalac, see: https://github.com/sbt/sbt/issues/1872
- It is still possible, though much more unlikely, to accidentally use a non-bootstrapped compiler when we expect a bootstrapped one. More sanity checks are needed to avoid this, for example it shouldn't be too hard to make sure that we never load any Scala2 symbol when compiling Dotty using Dotty.

2 IDE support

So far, Dotty has been mainly used as a batch compiler: given a set of files as input it compiles them, output messages and compiled files, then exit. However, it has been designed from the start for more interactive usecases, where the same compiler instance is reused for multiple "runs" (each run corresponding to a compilation attempt of a set of file), can recover from failures, and can reuse information collected from previous runs in the current run (after making sure that the information is not stale). As part of our

work on improving Dotty robustness we decided to harden its interactive use, and the best way to do this was to implement and test IDE support.

2.1 The Language Server Protocol

Proper IDE support for a programming language usually requires writing a plugin for each and every IDE without reusing much code since they all have very different internals. The Language Server Protocol is an attempt to solve this by standardizing the communication between an IDE and a *language server*. A language server is any program that can respond properly to the JSON-based messages sent by the IDE, quoting the documentation:

> The language server maintains semantic information about a program implemented in a particular language.
>
> - When the user opens a document in the tool, it notifies the language server that the document was opened and that the truth of the document is now maintained by the tool in a memory buffer.
> - When the user edits the document, the server is notified of the changes and updates the program's semantic information.
> - As the user makes changes the language server analyses the document and notifies the tool with any errors and warnings (diagnostics) that it finds.
> - When the user requests to go to the definition of a symbol, the client sends a definition request to the server. The server responds with the URI of the document and a range inside that document. Based on this information the tool opens the corresponding document at the position where the symbol is defined.
> - When the user closes the document, a didClose notification is sent, informing the language server that the truth of the file is now on the file system.

2.1.1 Limitations

The LSP is not mature yet:

- Many IDEs are cannot be used as LSP clients yet, see `http://langserver.org/` for a list of implementations.
- It provides only one kind of refactoring: renaming symbols
- Some of its supported queries are not precise enough: when the definitions corresponding to a symbol are required, it makes sense to return the definition of the symbol as well as the definitions of overriding symbols, but what about overriden symbols? Ideally, IDEs should let the user choose.

- It is under active development, which is a good thing but means that some code needs to be adapted when a new protocol version comes out.

2.2 The Dotty Language Server

Our implementation for Dotty is based on the Eclipse LSP4J library which gives us a simple type-safe API that takes care of the low-level details like marshalling and maintaining queues for incoming and outgoing messages. As an example, here is the code that implements the response to a query asking for the position of the definition(s) corresponding to the symbol at a given position:

```
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    implicit val ctx = driver.ctx

    val trees = driver.trees
    val spos = driver.sourcePosition(
      new URI(params.getTextDocument.getUri), params.getPosition)
    val sym = Interactive.enclosingSymbol(trees, spos)

    Interactive.definitions(trees, sym).map(asLocation).asJava
  }
```

We can understand most of the implementation details of the language server by going over this example in details:

- LSP4J insulates us from the message-passing details, instead we only need to override a few methods like `definition`.
- `computeAsync` create a cancellable closure that will be scheduled by LSP4J.
- `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 some old cancelled requests) but since responses are asynchronous this should never freeze the UI of the IDE.
- Since we will need to call internal compiler APIs, we need an instance of `Context`, it is provided to us by `InteractiveDriver`, a subclass of `Driver` that lets us run the compiler in an interactive mode.
- Definitions will be looked up in trees, this is currently done in a rather crude way: `driver.trees` will return the AST for all top-level classes opened in the IDE (and thus typechecked) and all top-level classes in the classpath who have a TASTY

section, this requires forcing their denotations. A more disciplined approach might be possible, for example by checking if a TASTY section contains the name of the symbol we're looking up before forcing any denotation in that file.

- From the protocol `TextDocumentPositionParams` we can easily get a `SourcePosition` suitable for use with compiler APIs
- The `Interactive` object defines a set of high-level APIs to retrieve information from trees. We first use it to get the symbol corresponding to the queried position, then to get the definition of this symbol and of all symbols that override this symbol. `Interactive` is currently tailored to the feature set of the LSP but could be extended to be used by other tools like Ensime or REPLs. It currently looks like this:

```
case class SourceTree(source: SourceFile, tree: Tree)

def enclosingType(trees: List[SourceTree], spos: SourcePosition)
   (implicit ctx: Context): Type
def enclosingSymbol(trees: List[SourceTree], spos: SourcePosition)
   (implicit ctx: Context): Symbol

def completions(trees: List[SourceTree], spos: SourcePosition)
   (implicit ctx: Context): List[Symbol]
def completions(prefix: Type, boundary: Symbol)
   (implicit ctx: Context): List[Symbol]

def definitions(trees: List[SourceTree], sym: Symbol,
   namePosition: Boolean = true)
   (implicit ctx: Context): List[SourcePosition]
def allDefinitions(trees: List[SourceTree], filter: String = "",
   namePosition: Boolean = true)
   (implicit ctx: Context): List[(Symbol, SourcePosition)]

def references(trees: List[SourceTree], sym: Symbol,
   includeDeclaration: Boolean = true,
   namePosition: Boolean = true)
   (implicit ctx: Context): List[SourcePosition]
def allReferences(trees: List[SourceTree], filter: String = "",
   namePosition: Boolean = true)
   (implicit ctx: Context): List[(Symbol, SourcePosition)]
```

2.3 Evaluting the usage of Dotty as an interactive compiler

We have show by example that Dotty can be used interactively, we have yet to stress-test our IDE support on big projects but so far all the issues in Dotty we encountered have been solvable. However, we want to draw attention to one class of issue that we haven't completely fixed yet: accidentally using a `Context` from a previous run. This usually manifests itself by a `StaleSymbolError` as we try to access a symbol in run n when it has already been brought forward to run $n + 1$. We have identified two root causes:

- In a few places (searching imports in scope, searching implicits in scopes, ...), we traverse the `outer` chain of contexts starting from the current one, this can be fixed by stopping the search when the `runId` changes (except for the import search which also requires the root imports from the very first `runId`).
- In many places we create closures that may capture the current context but won't be executed until a later run, even when these closures have an `implicit ctx =>` parameter the body of the closure might call methods which have their own `Context` in scope. So far, we have been fixing these issues as we discover them, but more could be hidden and they are easy to accidentally introduce, it seems worth investigating whether we could statically prevent accidental `Context` capture.

2.4 Future work

Our implementation is mostly feature-complete but work remains to give a good user experience:

- **Easy startup** Currently, the server has to be started manually by the user before an IDE can connect to it, we should make it easy for a client to start our server automatically.
- **Testing** So far, we have relied on manual testing but real tests will be necessary to avoid regressions, we hope to be able to repurpose some of the testing infrastructure and tests of the scalac presentation compiler and scala-refactoring project.
- **Optimizations** Our implementation is suboptimal in many way: it loads too much classes in memory, it cannot interrupt cancelled requests and it does not do targeted typechecking (meaning that every keystroke in a file ends up forces the recompilation of the whole file), all of those won't be easy to implement but should fit in our existing design.
- **Build tool integration** We assume that all closed source files in the project have a corresponding up-to-date classfile on the classpath, but this is not enforced. Running our own instance of `sbt` would be quite complex, instead we hope to be able to benefit from the ongoing work on adding a client-server mode to sbt, this would also allow us to considerably simplify project setup (we currently rely on the user running the `ensime-sbt` plugin in their build to generate a `.ensime` file ahead of time).

3 Conclusion

To make Dotty more robust we must first break it. The most reliable way we have found to do this is to make the compiler do things it has never done before, like our more extensive bootstrapping in Section 1 or our interactive use of the compiler in Section 2. There are many more ways of stress-testing the compiler that we have not explored yet, like shuffling the order of the files it compiles in tests, or fuzzing its input, which we expect will lead to more important improvements.