Integration of Dotty in Jupyter notebooks

Author: Romain Gehrig
Supervisor: Guillaume Martres
LAMP – EPFL

Master semester project – January 2019

1 Abstract

The popularity of Jupyter notebook for data analysis is undeniable. Its uses have now reached beyond the field and brought a powerful education tool with good literate programming capacities. The transformation from what was previously called IPython notebook to Jupyter marked the new possibility of creating Jupyter kernels for languages different from Python[1]. For this project, we've created a Jupyter kernel to support Dotty, the experimental compiler that will become Scala 3.0. Then, to generalize the newly added features, we've extended the Language Server Protocol to support REPLs. In this report, we'll describe what has become a big lesson in plumbing.

2 Introduction

2.1 Presentation of Jupyter

Jupyter[2]'s philosophy is "to support interactive data science and scientific computing across all programming languages"[3]. The Juypter notebooks are composed of cells that can either be code or Markdown. The code cells can be executed like simple blocks of code. Each notebook has a single programming language associated to it and the code cells will be evaluated using the kernel corresponding to the language. As presented in the abstract, it's now possible to make Jupyter kernels to support notebooks in any language.

2.2 Goals:

For this project, some goals were set during the implementation and were all met:

- **Jupyter kernel**: The first and major goal was to have a working Jupyter kernel so that Dotty notebooks could be created and used.

[1] See this article from The Atlantic, 2018, about the potential uses in research: `https://www.theatlantic.com/science/archive/2018/04/the-scientific-paper-is-obsolete/556676/`
[2] Jupyter's website: `https://jupyter.org/`
[3] According to Wikipedia: `https://en.wikipedia.org/wiki/Project_Jupyter#Philosophy`

- **General protocol**: Once a simple kernel was working, we searched to generalize the protocol so it could be implemented on the compiler-side and be also useful to other people wanting to build Jupyter-like applications. We decided to extend the Language Server Protocol (LSP) that already had a working implementation for Dotty in order to add REPL capacities. The goal was to find and match API calls similar to both LSP and Jupyter to reuse the maximum amount of code.
- **REPL Proof-of-Concept**: We needed a proof-of-concept that the new protocol could support other uses. The new LSP functions enabled the creation of a LSP REPL client: the REPL client delegates the Evaluation part to the server and only does the Read - Print - Loop.

2.3 Jupyter is a REPL

It may be surprising at first, but a Jupyter notebook acts exactly like a REPL: the unit of evaluation is a cell, which is equivalent to REPL with multiline support, cells are interpreted sequentially[4] and state is updated by running a cell. Its Read simply reads the content of the cell, Eval is running the cell, Print is printing the result and loop is choosing the next cell to eval.[5]

3 Presentation of the parts

This project depends on different support libraries to achieve its goals. Without them, a lot of time would have been invested to handle the different protocols. I would like to thank their respective authors for the time they invested in developing them.

3.1 Libraries used

3.1.1 Almond[6]: developed by Alex Archambault.

Almond is a library that abstracts away the communication protocol of Jupyter. It enables the creation of a kernel in Scala by only implementing some of the methods of the Interpreter interface.

The library also provides a working implementation of a Jupyter kernel for Scala. One reason for not reuse this implementation is that the underlying REPL used (Ammonite) is currently non-compatible with Dotty.

3.1.2 LSP4J[7]: developed under the Eclipse foundation aegis

LSP4J is to LSP what Almond is to the Jupyter protocol: it provides a layer of abstraction on top of it. In this model, each remote procedure call to a server returns a (Java) Completable-

[4] Sequential with respect to time, not from top to bottom as the top cell can be evaluated after the cell below it.

[5] It is interesting to notice the similarities in API between the `Interpreter` interface to implement for Almond and the `ReplDriver`.

[6] Almond's website: `https://almond-sh.github.io/almond/stable/docs/intro`

[7] LSP4J's github: `https://github.com/eclipse/lsp4j`

Future. As supported by the LSP, each request can be canceled before receiving a reply. In LSP4J, it translates to a the receiver getting a cancel token to check if the future has been canceled.

LSP4J also enables the addition of new messages types and server/client capacities, ie. remote procedures that are supported.

3.1.3 Coursier[8]: also developed by Alex Archambault.

Coursier enables easy packaging of Scala applications when using its `bootstrap` command and an easy launch of dependencies using the `launch` command.

3.2 Workflow and use:

The workflow for this project is a bit tricky as there are its share of moving parts. It uses SBT's `publishLocal` to publish the artifacts locally so they can be used by other parts of the project. For instance the ReplServer and the Jupyter LSP programs are different SBT targets: `repl-server` for the first and `jupyter-lsp` for the second. The `jupyter-lsp` target is important to publish locally: we then have to package it using by Coursier as a single binary. This binary is then executed with the `--install` flag to indicate that we want to install it where Jupyter stores all the kernel (most likely on `~/.local/share/jupyter/kernels` on Linux). Almond's `installOrError` method copies that binary and creates a `kernel.json` file in this location. This file contains the information for Jupyter to run the kernel with the right parameters, for instance `java -jar ...`.

```
# Publish the artifact
sbt jupyter-lsp/publishLocal

# Creates the kernel binary from the artifact
coursier bootstrap ch.epfl.lamp:jupyter-lsp_0.11:0.11.0-bin-SNAPSHOT -f -o kernel

# Install the kernel in Jupyter's dir
./kernel --install
```

Listing 1: How to install the kernel

Then, when we want to use a notebook for Dotty, we first need to launch a REPL server listening on a particular port (12555 for the moment) so the Dotty kernel can send commands to this REPL server. The server can be runned in two different ways: one is simply using SBT in the Dotty project and then `repl-server/run 12555` and the other is using Coursier once the artifact has been published locally:

```
coursier launch ch.epfl.lamp:repl-server_0.11:0.11.0-bin-SNAPSHOT -- 12555
```

3.3 Possible concurrent actions with the Python kernel

By curiosity, it was interesting to test which actions were permitted concurrently in a Jupyter notebook when using its Python kernel (which is the reference kernel after all). For instance,

[8] Coursier's website: `https://get-coursier.io/`

```
# Run the REPL server in a terminal
sbt "repl-server/run 12555"

# Launch Jupyter in another
juypter notebook
```

Listing 2: How to use the Jupyter kernel for Dotty

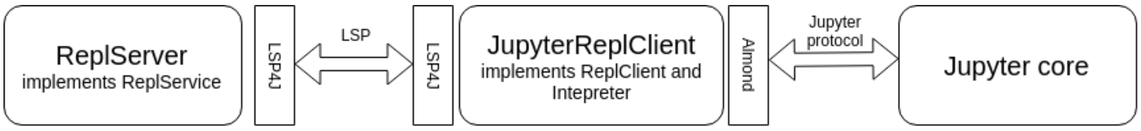

Figure 1: How pieces fit together

notebook users will quickly learn that it's impossible to run two cells at the same time. On the other hand, there are other limitations that perhaps a bit more surprising: it is for instance impossible to ask for completion or function documentation when a cell is running. At first, we could think it's the case because the Python kernel doesn't support it, but in reality we can see by running our own kernel that the request for completion is not even sent by Jupyter !

4 Implementation details[9]

4.1 How to interpret code remotely ?

At first, the message remote interpretation call was sending the code to `interpret`, waiting for the end result and printing it. However, as we are now in a remote setting, every intermediate prints of the REPL to its output stream would only be visible to the client at the end of the execution. That is not great for interaction or even debugging as we don't see anything until the end of the execution.

The problem had to be solved by requesting for new (partial) output until the execution was finished. However, in LSP each request can only get one response so we can't just send the request once and get the output streamed back.

The solution on the server-side was to reply directly after the initializing the REPL thread and to add a boolean to the reply to indicate if the REPL was still executing the code. In addition, a new LSP procedure (`interpretResult`) was created to query for new output. This method waits until the REPL has flushed its output stream so we could get the new output. Then the client sees that the REPL is still running so it calls the procedure again. This repeats until the execution of the code stops – either normally or by an exception. This way, we can have incremental updates for the client.

[9] The repository of the implementation can be found here: `https://github.com/RomainGehrig/dotty/`

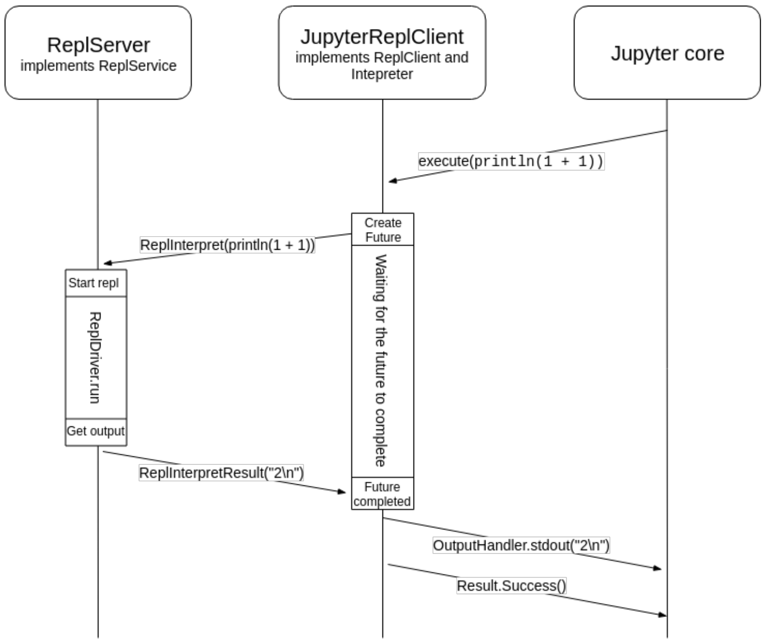

Figure 2: First implementation: the result is visible only after the REPL is done.

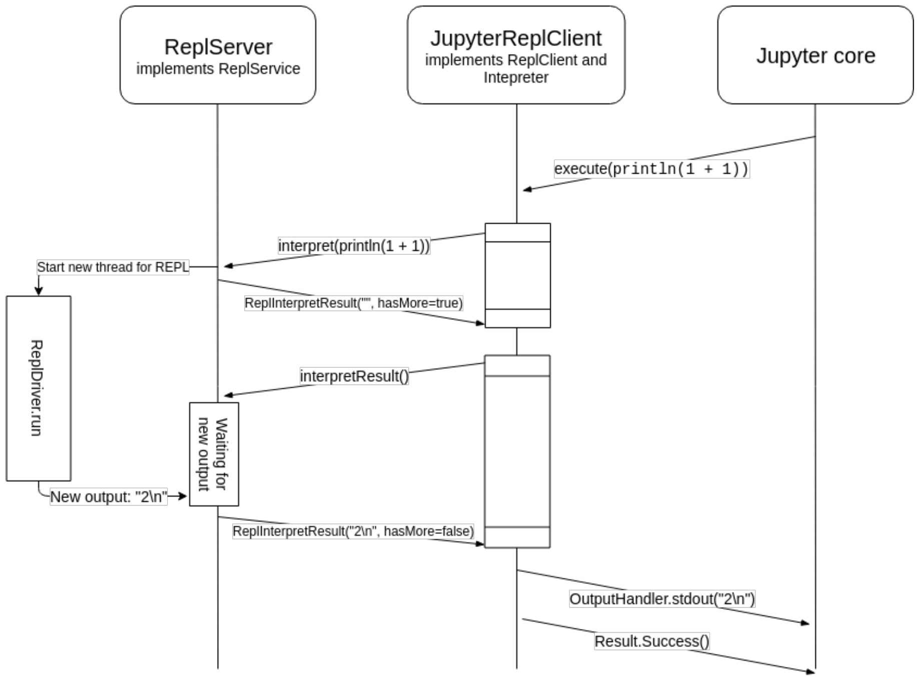
ReplServer
implements ReplService
JupyterReplClient
implements ReplClient and Intepreter
Jupyter core
execute(println(1 + 1))
interpret(println(1 + 1))
Start new thread for REPL
ReplInterpretResult("", hasMore=true)
interpretResult()
ReplDriver.run
Waiting for new output
New output: "2\n"
ReplInterpretResult("2\n", hasMore=false)
OutputHandler.stdout("2\n")
Result.Success()

Figure 3: Separated futures: we ask for the output separately from asking to run the code. Not illustrated here but the output can now appear incrementally for the client.

4.2 Interruption styles and their implementations:

It is important to be able to interrupt the execution of the program at all time as otherwise an infinite loop (or a long running command) can render the kernel and/or the client unusable. Using the previous protocol of asking for answers, we will show that canceling Futures doesn't work well in all cases and that we have to add a new mechanism for interrupting the execution.

4.2.1 First implementation: just cancel pending future[10]

For the first implementation, interruptions were implemented by canceling the pending Future output. The LSP server detected that the output request was canceled and interrupted the corresponding REPL thread. There were two problems with this approach:

1. We didn't get a stack trace (can be important for debugging)
2. The interruption failed in the precise case where the client cancels a request that the server just responded to.

It can be illustrated in the case below:

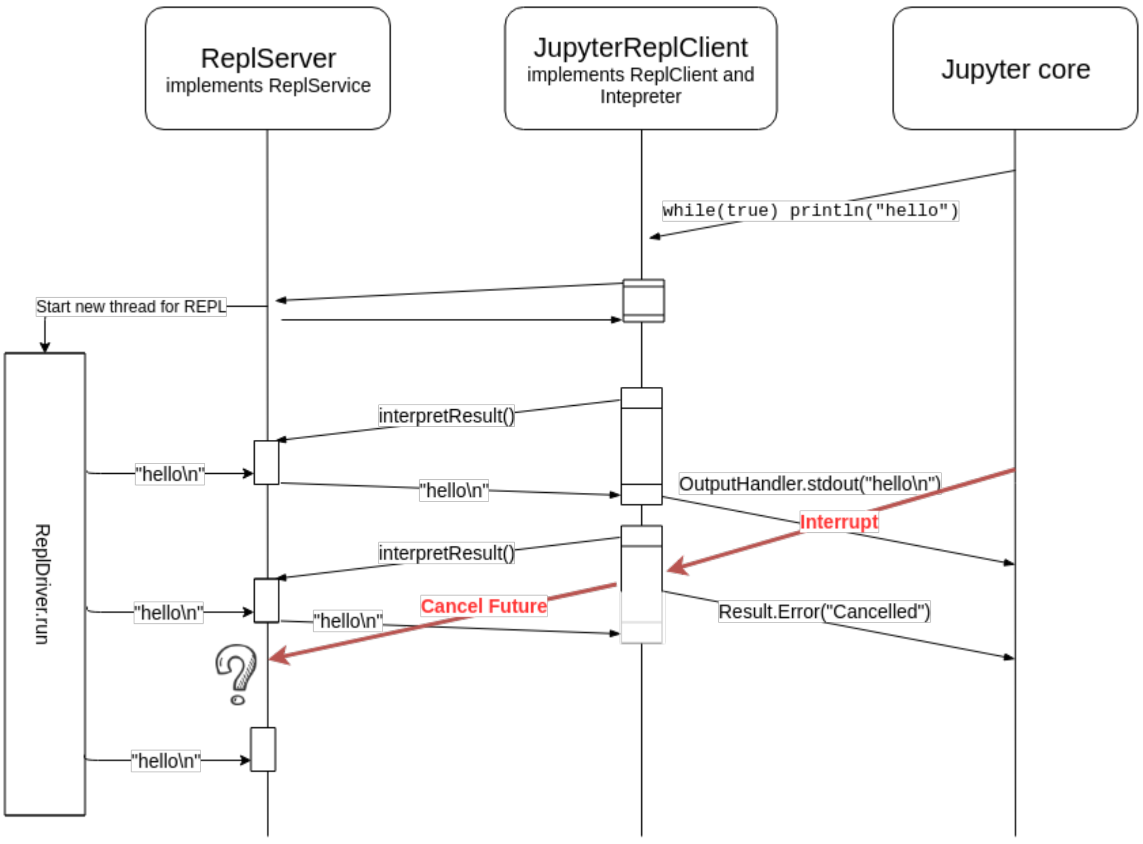

Figure 4: The canceled future doesn't run anymore on the server so it cannot react to the cancellation. Result: the REPL thread still runs.

[10] The version was implemented in `https://github.com/RomainGehrig/dotty/commit/7bc2f294aa9f40bebfaf0a184bdb7d5bda17841a`

4.2.2 Second implementation: explicit interruption

To counter this, a new LSP method was implemented on the server: `interruptRun`, taking the runId as the parameter. Because it is a new message, it cannot miss and it can also have a response back. In the response, the stacktrace was included as the main message for convenience.

4.3 Methods and messages created

Most of the methods and messages created where already explained before. We have explained the need for a separate method for interruption but not why there is a different method for completion. It comes down to the differences of how LSP works with IDEs and how Jupyter (and REPLs) don't work in the same way. LSP makes the assumption that both the client and the server have access to the files currently edited. When an IDE (LSP client) asks the completion for a symbol, the LSP server can fetch the file to look at the context. If the file had modifications, each modification was submitted to the LSP server via the method `textDocument/didChange` so it has the same knowledge of the file as the client. In the REPL case however, the server is only made aware of changes when the code is submitted. For completion, – and that's where the similarity between Jupyter and a REPL really strikes – only the current line is submitted as well as the position within that line. For a REPL, it's not important to know the general context, what matters is the current state of the REPL and what will influence the next state: the current line of code.

```scala
@JsonSegment("repl")
trait ReplService {
  @JsonRequest
  def interpret(params: ReplInterpretParams): CompletableFuture[ReplInterpretResult]
  @JsonRequest
  def interpretResults(params: ReplRunIdentifier):
  ↪ CompletableFuture[ReplInterpretResult]
  @JsonRequest
  def replCompletion(params: ReplCompletionParams):
  ↪ CompletableFuture[JEither[java.util.List[CompletionItem],CompletionList]]
  @JsonRequest
  def interruptRun(params: ReplRunIdentifier):
  ↪ CompletableFuture[ReplInterruptResult]
}
```

Listing 3: List of the methods added to support a REPL

```scala
case class ReplInterpretParams(code: String)
case class ReplRunIdentifier(runId: Int)
case class ReplInterpretResult(runId: Int, output: String, hasMore: Boolean)
case class ReplCompletionParams(code: String, position: Int)
case class ReplInterruptResult(runId: Int, stacktrace: String)
```

Listing 4: List of the messages added to support a REPL

4.4 Note: the two paths of the Kernel

There is a finer point in the `LspKernel.scala` to address: there are actually two completely different paths in the Main. One is used when the flag `--install` is present. The other path is taken when the kernel is started by Jupyter with the command specified in `kernel.json`. This path has to setup the Interpreter by creating the threads used to communicate with Jupyter. Then, by calling `Kernel.create()` with an instance of the `Interpreter`, we are actually finally launching the kernel.

4.5 LSP REPL[11]

As stated, a REPL was created to show that the new LSP capacities were enough. Here, the REPL is very simple. It only reads lines and handle completion by adding a question mark at the end of the line. Interruptions were handled by capturing the SIGINT and sending a `interruptRun`.[12]

5 Reflexions, future work and improvements

5.1 ANSI escape and Jupyter cells

Interestingly ANSI escape code can still be used to modify the behaviour of the REPL's stdout, but they are completely ignored by Juypter[13]. What Jupyter supports however is updating the output cell using their protocol. In Almond, it is possible to update the display on the kernel side using `DisplayData.*` functions. For the clients however, they have to create a function taking an implicit parameter `almond.api.JupyterApi` that will be provided to them[14] to accomplish that. Having the REPL separated from the Jupyter kernel most likely makes the same trick very hard to realize.

5.2 Managing classpaths

Managing dependencies is still difficult in Scala, as classpaths are the only way to tell the compiler where to search for dependencies. Where Jupyter differs a bit from a REPL is that its command-line arguments are set in the `kernel.json` file and are not easily updatable by the end user. As such, it's not possible to easily add a dependency. A better way of setting dependencies would be to handle special commands written directly in a Jupyter cell like `%add_classpath '/path/to/new/jar'` – or better, with Coursier capacities: `%import 'ch.lamp.epfl:dotty:2.14.10'`. These would be handled specially by the Jupyter kernel and forwarded as a new type of LSP message to the ReplServer that would then update the ReplDriver's classpath.

[11] Repository of the implementation: `https://github.com/RomainGehrig/LSP-REPL-PoC`

[12] To see an abuse of Futures, head to: `https://github.com/RomainGehrig/LSP-REPL-PoC/blob/master/src/main/scala/LspReplClient.scala#L77-L82`

[13] Try with `println("Hello"); println("\033[1AWorld")`.

[14] From Almond's documentation: `https://almond-sh.github.io/almond/stable/docs/api-access-instances`

5.3 For clients to launch their own REPL server: problems with streams

As we don't support multiple concurrent clients on the server (the compiler isn't thread safe yet), the best next thing is that each client runs its own instance of the server instead of having two different processes and having to specify a socket to communicate. After some unsuccessful attempts of running the server as a child process of the client, it was thought that the efforts would be better placed somewhere else. The handling the input/output of each process and knowing to what each stream was linked were the main difficulties encountered - each LSP client/server and the ReplDriver must have their own IO streams. Combined with the need of regenerating the different artifacts each time changes were made and having SBT behaving with weirdly with the IO, that particular task was put aside.

5.4 Handling of the input stream (eg `scala.io.StdIn.readLine`)

For this to work, a new message needs to be added to the protocol. In the same vein as the ReplInterpretResult enabling a remote access to the output stream of the ReplServer, we would need a mean to stream characters and (maybe keys) pressed by the client for a complete remote terminal experience.

5.5 Reimplementation of `interpret`:

After implementing the interruption request, an alternative implementation of the interpretation process using notifications emerged back as a potentially cleaner approach. During the search for solutions to have incremental output, it was put on the side by lack of knowledge on how the LSP notifications worked and their properties/guarantees (ordering, deliverability, . . .).

Here is a gist of the implementation using notifications:

- Have a long running `Future[Result]` as in the first implementation.
- Use notifications with partial output sent by the server to the client (with IDs for the client to coordinate the prints).
- An interruption can simply cancel the Future with no chance of "miss-timing" like talked about previously.

5.6 Possible uses of the REPL protocol

If this protocol was adopted like LSP, we could have a lot of different REPL clients to choose from depending on our taste. We could have a single REPL to pair with the language we want (if there is a LSP REPL server for it). We could also imagine connecting to a debugging server while having a full-fledged REPL customized to our needs.

6 Evaluation

The goals of having a functional Jupyter kernel for Dotty was met. There is still work to do to support functionalities like handling input streams and the general process to get a Dotty notebook running is still rough and unwieldy. But overall the project went well.

7 Conclusion

We have seen that there exist fundamental differences between REPLs and IDEs. They may no seem like much but we have to consider them when creating a unifying protocol to embrace both.