Functionnal Programming project
IT & Management Department

Creation of a Git-like tool in Scala

Author
Thomas Falcone

October 20, 2019

Contents

Every link of this document is clickable if you are reading this in a digital format

1 Instructions

You can find the source code on Github through the following link : Source code
https://github.com/ThomasF34/sgit

In order to install **sgit** you can either use the Github release TODO or build the binary file from the source code. *Be aware that you must have installed sbt before trying to compile the source code. You will find more information about sbt on this site https://www.scala-sbt.org/*

To compile from source code use :

```
sbt assembly
```

Then add the file *target/scala-2.13/sgit* to your PATH.

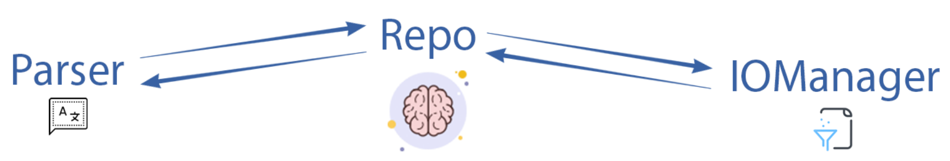
To install via the Github release, simply download the binary file and add it to your PATH.

*You are now able to use sgit. You can find the usage with **sgit –help***

2 Architecture

2.1 Presentation

The architecture of sgit is quite explicit and can be seen as a simple flow.

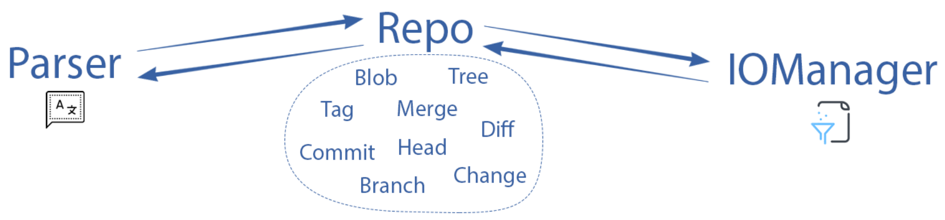


Everything begins within the **Parser** that will take the given command, initiate the **Repo** instance and ask it to execute the wanted action. It will then load all the needed the content from the **IOManager**, the HQ of File management and compute the result.
In order to achieve this goal, the Repo uses an abstraction of the objects it manipulates. Hence, it exists several *case classes* that will help readability and separate each other's concerns.

Here is a list of existing classes :

- Blob
- Tree
- Branch
- Tag
- Commit
- Head
- Merge
- Diff
- Change

Theses objects are using High Order Functions in order to gain access to the content they need. For example we give a function called *commitContent* to the Commit object so it can access some commit's informations. Hence, every function that needs to have a commit information respects Referential Transparency with this function injection. The Repo instance is responsible for this function injection. It is using IOManager's function to let other object access file's content, while hiding its origin to the aforementioned object.



Also, I used currying in order to specialize theses functions we are giving to other classes. Let's see how I used currying with an example :

```scala
def getContent(dir: String)(filename: String): String = {
  val file = new File(s"${dir}$filename")
  if (file.exists() && file.isFile()) {
    Source.fromFile(file).mkString
  } else ""
}
```

Listing 1: Function in IOManager

```scala
lazy val blobContent = ioManager.getContent(blobsPath)(_)
lazy val tagContent = ioManager.getContent(tagsPath)(_)
```

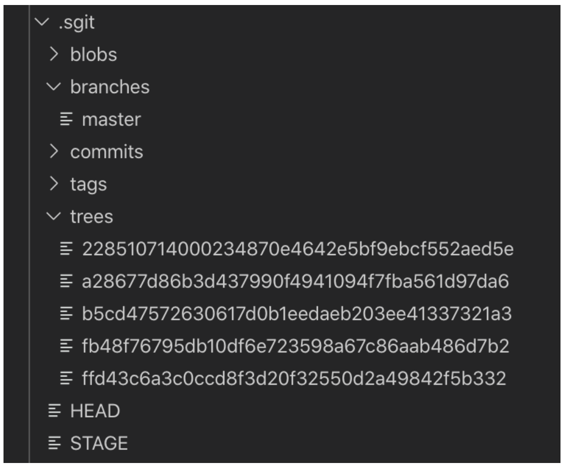
Listing 2: Functions to be injected

With these functions, the Blob instance (to which we are gonna inject *blobContent*) will only be able to access blob content while Tag instance will only be able to access tag content.

2.2 File organization

For the sake of readability, I decided to fill a *.sgit* directory with all the repository informations. All objects are sorted in a sub directory corresponding to their type.

Here is the organization of this file :



```
∨ .sgit
  > blobs
  ∨ branches
    master
  > commits
  > tags
  ∨ trees
    228510714000234870e4642e5bf9ebcf552aed5e
    a28677d86b3d437990f4941094f7fba561d97da6
    b5cd47572630617d0b1eedaeb203ee41337321a3
    fb48f76795db10df6e723598a67c86aab486d7b2
    ffd43c6a3c0ccd8f3d20f32550d2a49842f5b332
  HEAD
  STAGE
```

Figure 1: File organization in *.sgit* directory

2.3 Git copycat

In order to reproduce most of Git function I had to implement some interesting algorithms, firstly for the *diff* functions and then for the *merge* one.

2.3.1 Diff algorithm

In order to obtain the diff between two files I implemented an algorithm based on the Longest Common Sequence (LCS) between two collections (here two collection of strings, where each element represents a line of the compared file)

The idea behind this algorithm is to fill a matrix based on the content of both list. Then, once the matrix is filled, we start from the last element of the matrix and go backward to return at the beginning. The way we are using to come back to the first element will indicate wether the content had been added or removed (or is common between both list). You can retrieve the algorithm implementation in the *Diff.scala* file [1]

2.3.2 Merge algorithm

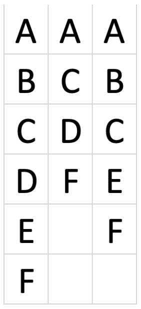
To compute the merge algorithm I had to obtain the LCS between three files. The first and the second one are the files on each branch we want to merge and the last one is the file on the common ancestor commit.

[1] https://github.com/ThomasF34/sgit/blob/master/src/main/scala/igpolytech/Diff.scala

I implemented an algorithm to compute the LCS matrix '*in 3D*' and even if it sounds more complex, it is not really the case. In fact, for the merge algorithm we don't need to know if each modified line had been inserted or deleted.

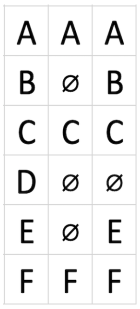
The complex part of this algorithm, though, is *the align* step. I had to align all three file based on their LCS to then compare line by line and apply changes.

Here is an imaged example of what the align step will do. (You can retrieve the algorithm implementation in the *Merge.scala* file [2]



A	A	A
B	C	B
C	D	C
D	F	E
E		F
F		

(a) Before alignment



A	A	A
B	∅	B
C	C	C
D	∅	∅
E	∅	E
F	F	F

(b) We can now compare line by line

2.4 Pros

Though this architecture has some drawbacks, it also have major advantages !

First, all the **IO is grouped** in Repo and IOManager (and Parser for interface displaying) znd the usage of HOF makes all the other classes (Commit, Blob, Tree, etc.) **pure and respecting RT**. Secondly, the currying usage is great to **separate each classes' concern and reduce coupling**. Then, for all the getters functions, the usage of *lazy val* is useful to **increase performance**. Lastly, **readability is highly increased** with the existence of abstract data structures (Commit, Tree, Blob...) and of *.sgit* directory organization (coupled with XML format)

2.5 Cons

Obviously these choices cannot be 100% perfect so they come with some drawbacks.

First, Repo is the brain of all actions and it results in a **complex object** (around 500 lines of code). Secondly, in order to limit objects' actions, I had to **declare a lot of variables** (at the top of *Repo.scala* file [3]) - which doesn't not improve my first point... Lastly, usage of HOF **increase the number of parameters** given to object's function and, though it helps reducing coupling and limiting object's scope of action, it can **become a weird enumeration when calling an object's function**

[2] https://github.com/ThomasF34/sgit/blob/master/src/main/scala/igpolytech/Merge.scala
[3] https://github.com/ThomasF34/sgit/blob/master/src/main/scala/igpolytech/Repo.scala

3 Tests

3.1 Naïve approach

At the beginning of the project I've been mainly **focused on integration tests** to see if all the use cases were correctly made and all users actions would lead to a successful answer to their needs. But I quickly saw that it was not precise enough because it only gives an overview of the problem when one occurs. Gradually, I began to see that the strategy of focusing on use cases and integration tests was not ideal.

After a great refactoring at the end of the project, I tried to change my point of view on tests and tried to **focus more on individuals, unitary tests**

3.2 I-understood-the-mistake approach

After my refactoring step, I coded a lot of unitary tests to test each functionality in object. Here you can find an example of test for the *getLastCommit* function that returns an Option of Commit from a Branch.

```
it("should return None if no last commit") {
  val branch = Branch("master", "")
  val fakeCommitRepoDir = mutable.Map[String, Node]()
  val mockCommitContent = (hash: String) => fakeCommitRepoDir(hash)

  val res = branch.getLastCommit(mockCommitContent)

  res shouldBe None
}
```

Listing 3: Unitary test of *getLastCommit*

I also kept the integration tests that I previously wrote, though I did not get enough time to transform them with a mock of *IOManager*

4 Post-mortem

If some say that *with great powers come great responsibilities*, I would say that *with great projects come great lessons*

4.1 Teamwork has a lot of benefits

Though this project was individual, I spent a lot of time in groups talking about architecture, algorithms, data structures and all. This dynamics created by teamwork is really important and help us, individually, keep a constant fresh eye on our work. Plus, it can really increase the quality of the work we produced - the dynamic we created with Yannick Mayeur and Lucas Gonçalves helped a lot for our respective projects, though the aim of it was trying to find a counter-example to prove the weakness in each others works

4.2 Testability is important

This project also changed the vision I had on testability. In fact, it was really important to test but I had placed tests at the end of a project's workflow. I realized it was really something to think about from the very beginning of a project. During the late refactoring I did on the code, I changed IOManager to make it mockable and it changed a lot of thing, opened new possibilities of tests, that would have increased the quality of my project if thought from the beginning.

4.3 Dev tools can be a great guiding light

Lastly, the great lesson I learned during this project is that the time spent setting up dev tools (SonarQube, Travis etc.) is 100% worth it. During all the living cycle of the project I realized it was a great and mostly trustful guiding light that helped me to keep in mind that the quality of a project is a constant and meticulous work