Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

July 5, 2012

Contents

1 Introduction

1.1 Git

A version control system is a tool that records changes on your files over time. The main idea is to keep track of the changes on your files and to retrieve them later. There are mainly three kinds of version control system, local, centralized and distributed. On a local VCS it is not possible to collaborate with other users and the structure that keeps track of the files is kept only locally. The need for collaborations with other users brought the centralized VCS. On the centralized version control systems there exists a server that keeps track of the changes on the files and each user pulls and checks out files from this server. This approach has mainly two problems. The first is to have a single point of failure, if a server goes down during a certain time, nobody can check out or pull updates. The second problem is when you are not connected to the network, you cannot pull or commit your changes. To solve this problems, the distributed VCS appear. Here each user keeps a mirror of the repository and there is not a central server. An user can always commit and when connected can pull/checkout the changes.

Git is a distributed VCS. It was created in 2005 by Linus Torvalds, for the Linux kernel development. The main difference when comparing to others distributed VCS is that *git* keeps snapshots of how your system looks likes on a certain moment in time instead of keeping track of changes.

1.2 Motivation

Nowadays *git* is having great success, it is efficient, fast and in the begin it looks easy to understand and to use. When the users start diving into git, they are faced with a behavior that nobody can explain very well. There are not any formal or even informal specification of the operations, like what are the pre-requisites and what is the result of performing an operation. So normally the users when fronted with something they do not understand, they avoid it.

The purpose of this project is to formally model the *git* core using a tool called Alloy, analyse and model some *git* operations and then check which properties the model does (not) guarantee. This manual presents to the reader semi-formal description of *git* internals, the description of some operations and it ends with some properties that the model guarantee.

Alloy is a declarative specification language, used for describing structures in a semi-formal way. After modeling the structures, it is possible to check for some properties, look for counter examples and to visualize instances of the model.

So, the difference of this manual from the others, is that, before writing it, we tried to understand and model the *git* concepts from a semi-formal perspective using Alloy. As result we think that our understanding on some key parts of *git*, is more precise and rigorous, than most known manuals. Thus, we try with this manual to pass the knowledge obtained.

1.3 Manual Structure

This reports starts by explaining the organization of *git*, how it is structured and then we cover each component of *git*. Working directory, index and repository. For matters of time and complexity we focus our model in the index and in the repository more precisely in the object model. After the presentation of the structure we give the semi-formal specification of some operations. We finish the report with the analysis of some properties.

2 The Git structure

Git, as it was said before, is a distributed version control system. It means that each user as a mirror of the repository (local repository) and in the case of *git*, there is no need for a central server, even that, each user is able to fetch or push updates from other user's repository (remote repository). *Git* is divided mainly in three components. The working directory, the index and the repository. The connection between these components can be seen in Figure 1. Local and remote repository have the same internal structure. What is a local repository for an user is a remote repository for another user.

3 The Repository

Git repository is like a database of a project along construction time. It is the place where the snapshots taking during the project construction are

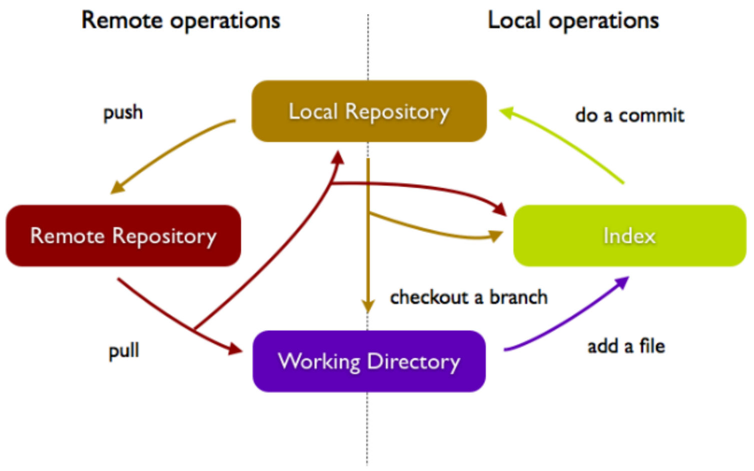


Figure 1: Git WorkFlow

kept, as well as, all the necessary information to allow the users to go through all the snapshots. The snapshot's structure and snapshots' relations are kept using, what is called, *git objects*. The information needed to go through the project is kept in, what is called, *git references*. In the next subsections we present both of them.

3.1 Git Objects

As we have presented before *git* keeps snapshots of how your system looked like on a certain moment in time. This moment in time is represented in *git* by a commit. Each commit points to a tree (corresponding to a directory), that represents the structure of your project on that moment. So a tree contains others trees and/or blobs. In *git* the files are not kept. What *git* keeps is a object called blob that represents the content of files. The relation between the path of a file and the content of that path is kept on the trees corresponding to that path.

In Figure 2 is possible to see an example of a snapshot. We have a commit that points to a tree. This first tree corresponds to the root directory of the project. All the other trees (and blobs) corresponds to some directory (or file) preserving the relation between them.

In the next sections a detailed description about each object is given, but for now something that is useful to know is that each object is identified by a 40-character string. This string is calculated by taking the SHA I hash of the contents of the object. This approach has many advantages, being the two more important, in our opinion, the fact that it is possible to compare two objects only by the name and if two objects have the same content then, only one of them is kept.

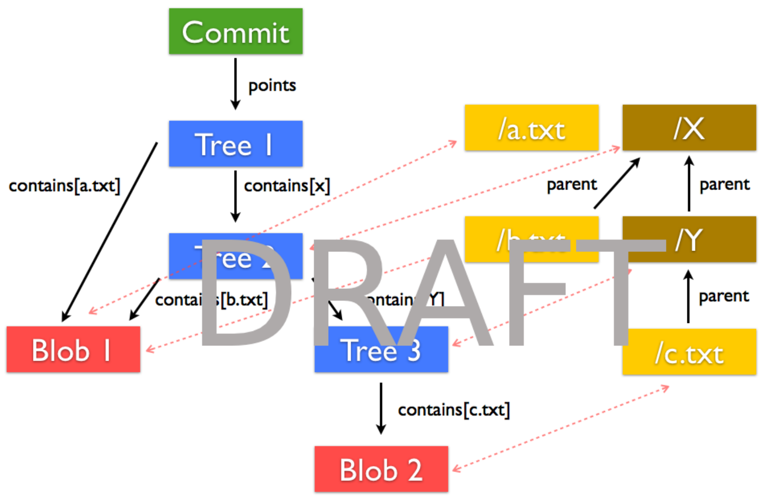


Figure 2: A Snapshot

3.1.1 Blob

A blob object, as it was said before, represents the content of a file and the blob name is calculated from the blob content. So if we have two files exactly with the same content, only one blob will be kept, even if, they have different names . This happens because the blob is not directly associated to a file. The relation between a path (of a file) and a blob is kept in the tree object.

3.1.2 Tree

A tree is nothing else than a map from names to blobs and other trees. This map represents the contains relation. Also here, a tree is not associated to any specific directory. If two directories have exactly the same content they will be represented by the same tree object. Here have the same content means that it should have exactly the same relations, or in other words, we have exactly the same names in both trees and each name corresponds exactly to the same objects in both trees.

3.1.3 Commit

The commit object, is like a snapshot of the project on a certain moment in time. A commit object has more variety of information than the two objects addressed before. Looking a commit, it is possible to find out the following:

- Author - The person responsible for the change on the project.
- Committer - The person which actually created the commit. It is possible that the committer is different from the author. An author

can create a patch, send it to the committer, which will create the commit.

- Parent - The previous commit, or, the commit from which this one was created. It is possible for this field to be empty, when the commit is a Root Commit (the first commit to be done). It is also possible to have two parents, when the commit is a result from a merge operation.
- Comment - It is possible, when creating a commit, write a comment about the commit. This comments can contain details about the changes that were done in the project, as well as, something that the Committer wants to write.
- Tree - It is a pointer for a tree, or in other words, it is a pointer for how the project looked like on a certain moment in time.

In our model, we concentrated our efforts in the Parent and in the Tree fields. We just care about the structure of the object model, so, we removed everything else that does not influences such structure. One property that we observed when modeling, is that, we cannot have cycles in the parent relation. It means that if we are on commit C1 and we go through the parent relation, we will never reach C1.

3.1.4 Tag

At last we have the tag object. The tag object is just a pointer to a commit with some more information. It can be used to mark a special commit, like a new version of the project. The structure is quite similar to the commit object. It has a tagger (the person who created the tag), a date in which the tag was created, a message (some comment from the tagger) and it points to a commit object. In our model we did not model this object, because when abstracting it looks like a branch, that we will see in next section.

3.2 Git References

Basically a reference is just a pointer to a commit. There is much more about references, but in this manual, lets keep it simple and concentrate on the object model. In this manual, we just speak about Branches and the special reference HEAD.

3.2.1 Branches

One of the trump cards of *Git* is its definition of branches. In other VCS each time a branch is created a new copy of the entire repository has to be done. Thus, branches operations in those VCS are slow and hard to use. In *git*, when a new branch is created the only thing that *git* does, is to create a new pointer to the current commit. The current commit is marked by a special reference called HEAD, that we present next.

3.2.2 HEAD

HEAD is maybe the most important reference in the repository. It indicates where you are situated in the repository. When a commit is done, *git* has to know which commit is going to be the parent. It does so, looking into the HEAD. HEAD normally is a pointer to a branch, so when the commit is done, the HEAD is kept, but the branch points to the new commit. HEAD can point directly to a commit, but lets keep it simple.

4 Working Directory

The working directory is basically a subset of a file system that contains the files of the project you are currently working on. This files can be the current files, files retrieved from an old snapshot or even files that are not being tracked. When retrieving an older snapshot of the project, the working directory is updated to reflect the project in that state. The untracked files in the working directory are just ignored, unless there is a conflict when retrieving files from an older snapshot.

When a user starts a repository, all the files are untracked. When a new file is created it will be untracked. So, how does *git* know which files are tracked or not? There exists the index which we present next.

5 Index

The index is something in between the working directory and the repository. When a file is created if the user wants that file to be on the next commit, it has to be on index. Even if the user just modifies a file and the user wants that change to be reflected on the next commit, it must be added to index, otherwise, what will be committed is the older version of the file. So, basically the index contains all the files that will be in the next commit.

6 Git operations

In this section we explore some of the git operations. We try to do so, in a different way, if comparing with other manuals. Normally the manuals we see around just explain how to perform an operation. We try to go further, so in this manual, we have divided our operations in three parts. In the first part, we give a description about the operation. In the second part, we give the pre-conditions, or in other words, we explain in which conditions the operations can be performed. Finally, in the last part of each operation, we show what is the result of perform such operation.

6.1 Add and Remove

The operations add and remove are used to add and remove content from index. As we have said before the index contains all the files and contents to be added on the next commit.

Git add does not refer directly to the file. Instead it is like a map from file to content. If we have a file on the index and we modify it on the working directory, for this modification to be visible on the next commit, the file has to be added again to the index.

The remove operation removes a file from index, so it will not be present if we perform a commit exactly after the remove. The removed file besides of being removed from index, it will also be removed from the working directory (if it is still exists there).

6.1.1 Pre requisites

There are a few cases where Git prevents you from removing a file, and they are:

- The file doesn't exist in the index.
- Removing a file, when the file with it's current content doesn't exist in last commit.

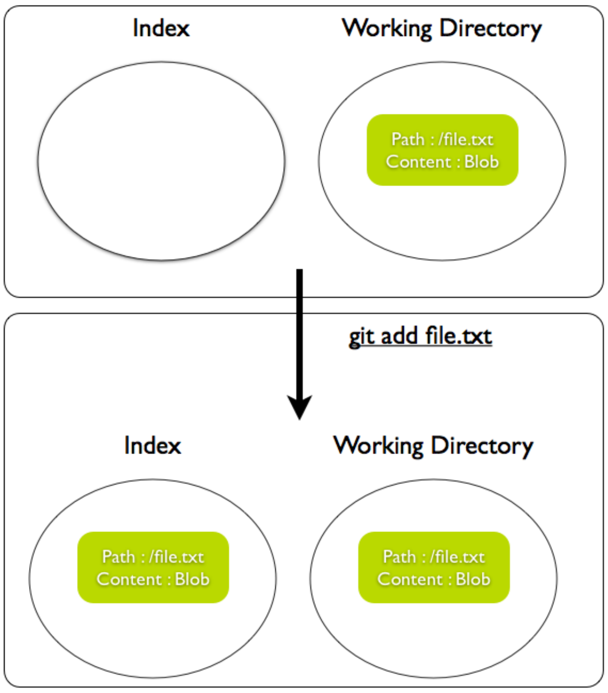
The last restriction exists, to avoid the accidental deletion of files. Imagine the case where you have a file 'x' with 10k lines of code. You add it to the index, but, before committing you accidentally use 'git rm x'. Because Git erases from the index and the working directory you would lose it permanently.

6.1.2 Result

6.1.3 Examples

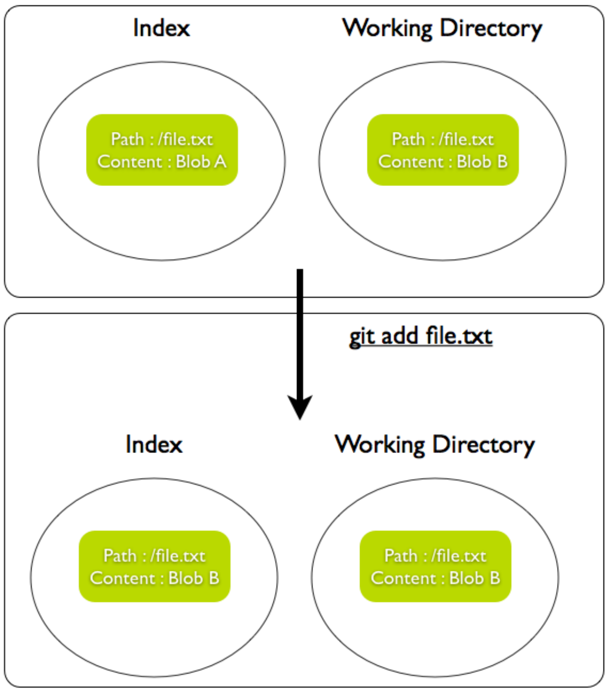
In these figures, it is shown the process of adding a file to the index. Note that both files showing represent the same file, but with different content at a given time. So imagine that, after adding "File1", "File0" is added. "File1" would be removed from the index (because there cannot be a file with different contents in the index at any given state) and "File0" would be added. What was just described was the representation of the update of the content of a file in the index.

Figure 3: Add operation



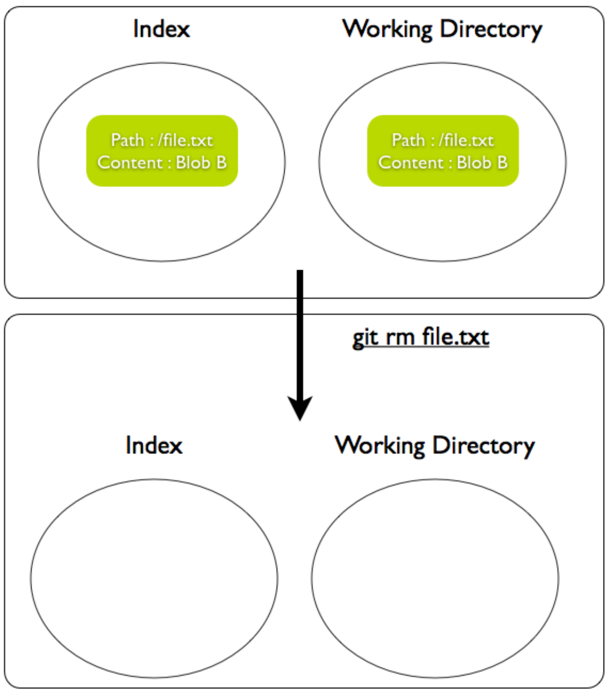
pagebreak

Figure 4: Add operation (update)



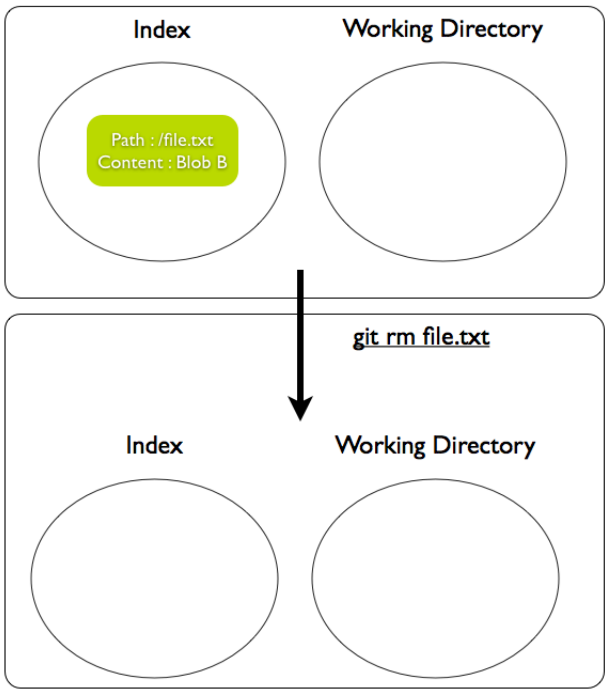
pagebreak

Figure 5: Rm operation



pagebreak

Figure 6: Rm operation (when the file doesn't exist in the Working Directory)



6.2 Commit

The 'git commit' operation, stores the current content of the index in a new commit.

When the first commit is created, a branch called "Master" will be created and will be marked as the current branch. Also, the first commit of a repository is called a RootCommit.

6.2.1 Pre requisites

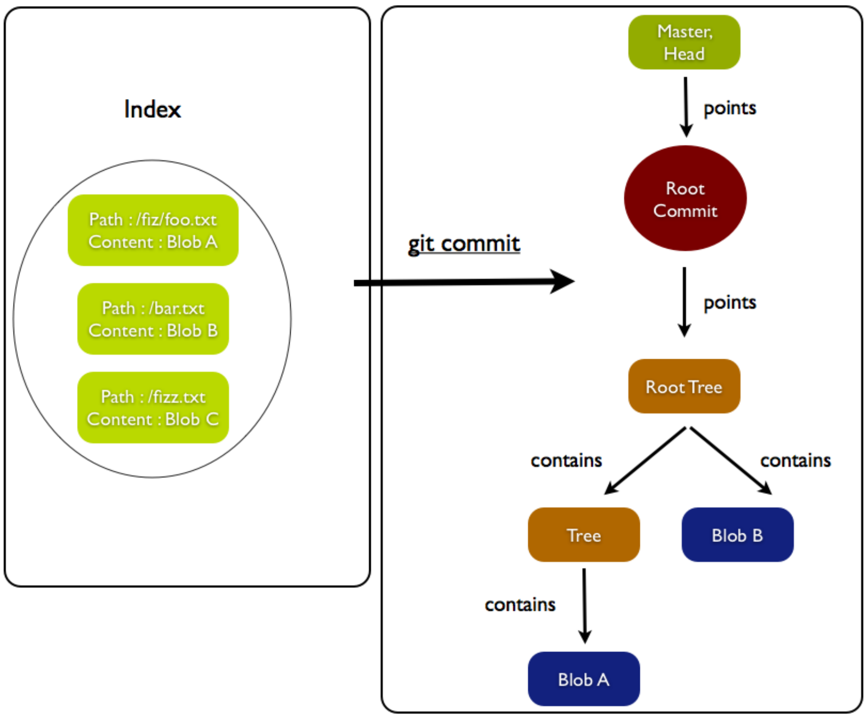
The only restriction that Git has about this operation it's that your new commit must have something different from the previous commit, or, if it's the first commit, then there must exist some content in current index.

6.2.2 Result

6.2.3 Examples

pagebreak

Figure 7: A first commit



pagebreak

The figure 8 represents the case of a normal commit, where the newest commit comes after the RootCommit. The line of thought is the same as in the last image. We can see that the difference between these two commits in the figure, is that the only file was moved to a folder named "Name".

We can also see the efficiency of Git, about the sharing property of objects in the commits, thus minimizing the space occupied in the repository.

6.3 Branch

6.3.1 Pre requisites

6.3.2 Result

6.3.3 Examples

Git branch operation has several forms, that to different things on git. The ones which we care are the forms that create and delete branches.

When a new branch is created it will point to the current HEAD by default.

The man git-branch says that when deleting a branch, it must be fully merged in its upstream branch, or in HEAD if no upstream was set.

6.4 Checkout

6.4.1 Pre requisites

6.4.2 Result

6.4.3 Examples

From man git-checkout : "Switches branches by updating the index, working tree, and HEAD to reflect the specified branch or commit".

6.5 Merge

When you do a 'git merge <branch>', a merge will happen between the current branch and the branch selected on the command. The result will be a new merge commit, or a partially merged index.

Git knows three types of merge :

- A fast-forward merge, that will happen when the current commit from the other branch, is more recent than the current commit from the current branch.

  This type is not a true merge, in the sense that the only thing that changes it's the pointer of the current branch is moved to the newest commit.

- A 2-way merge, that happens when the conditions for a fast-forward don't exist, and there isn't a common ancestor for both commits.
- A 3-way merge, happens when the conditions for a fast-forward don't exist, but, there is a common ancestor for both commits.

  The difference between these two last types, is that the last is much more automatic having much less conflicts than the previous type.

Knowing these 3 types of merge, it's important to know that Git tries to make the merge process as automatic as possible. However conflicts do exist, and when they happen it leaves to the user the process of resolving those conflicts.

When the Git/user, finishes the merging of files in the index, he can commit the resulting merge, in a merge commit.

Note that while there are unmerged files in the index, a commit cannot be made.

6.5.1 Pre requisites

- A merge can't be made, if the current commit is more recent then the other commit.
- A merge can't be made, while there are unmerged files.
- In the case of a fast-forward merge, the index can't have uncommitted files that can be changed by the merge operation.
- In the case of a 2-way or 3-way merge operation there can not be uncommitted files.

6.5.2 Result

6.5.3 Examples

6.5.4 Git Pull and Git Push

When working with remote repositories, these two operations will be vastly used. They are in fact a sequence of operations, and are used to get content from a remote repository and to place your content on the remote repository.

If a 'git pull' is done, the branch from the remote repository will be fetched, and the local branch and the remote branch will be merged.

If a 'git push' is done, the remote branch will be uploaded to the remote repository and will be merged there.

However, a 'git push' can only be done when the resulting merge will be a fast-forward, simply because there is no one on the other side to resolve the merge conflicts. For the case of 'git pull' it's pre requisites are the same of the merge operation.

References

Figure 8: A second commit

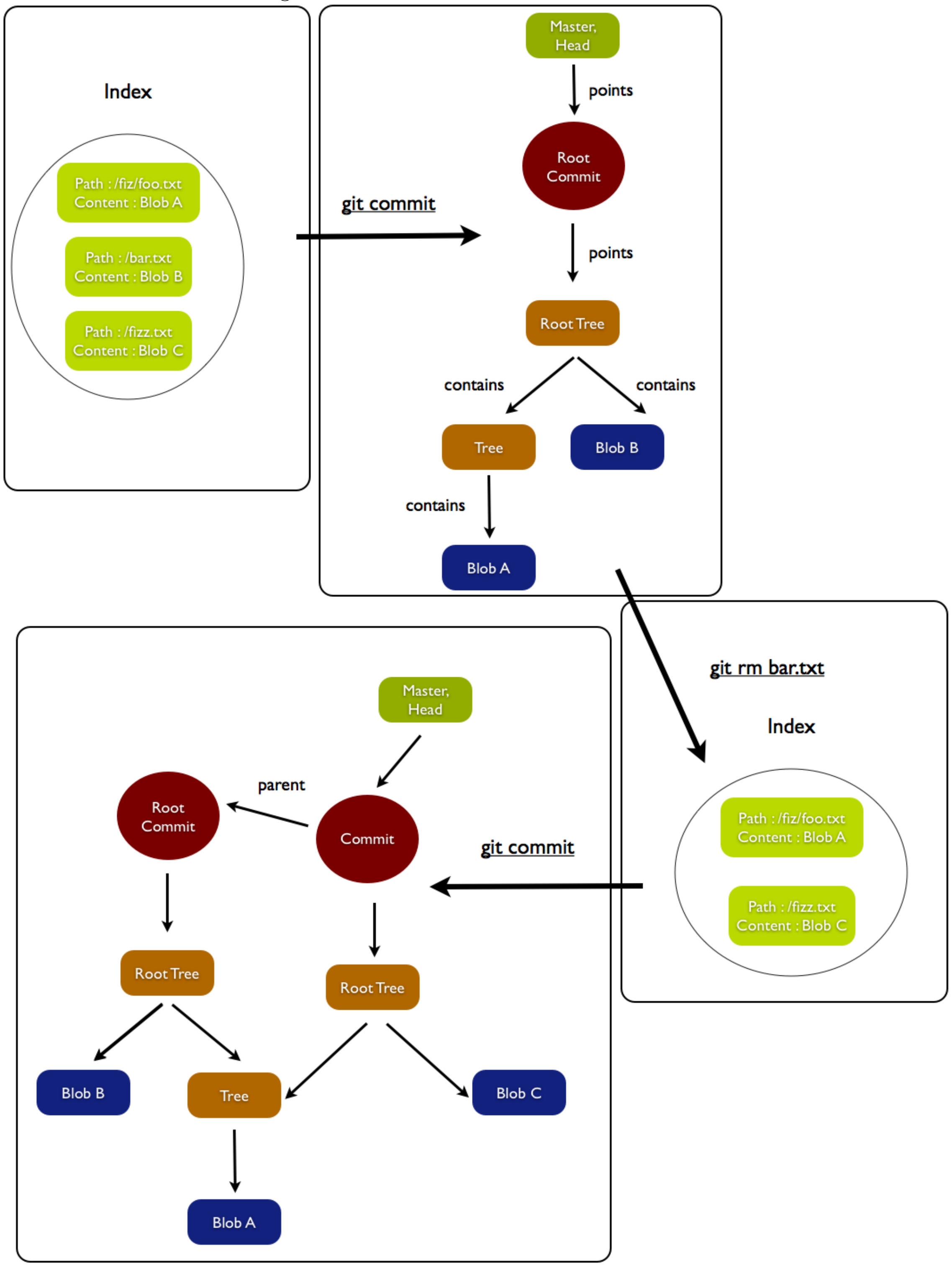


Figure 9: A typical case of a fast-forward commit

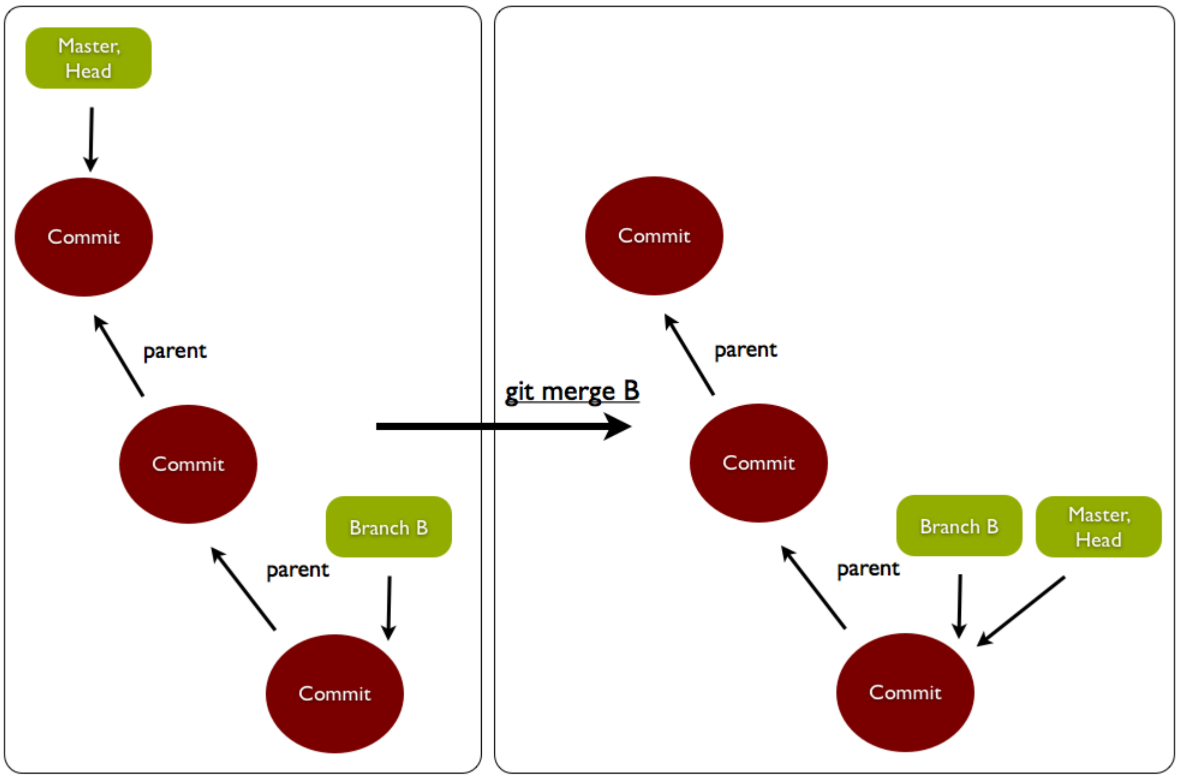


Figure 10: Normal case of a 3-way merge

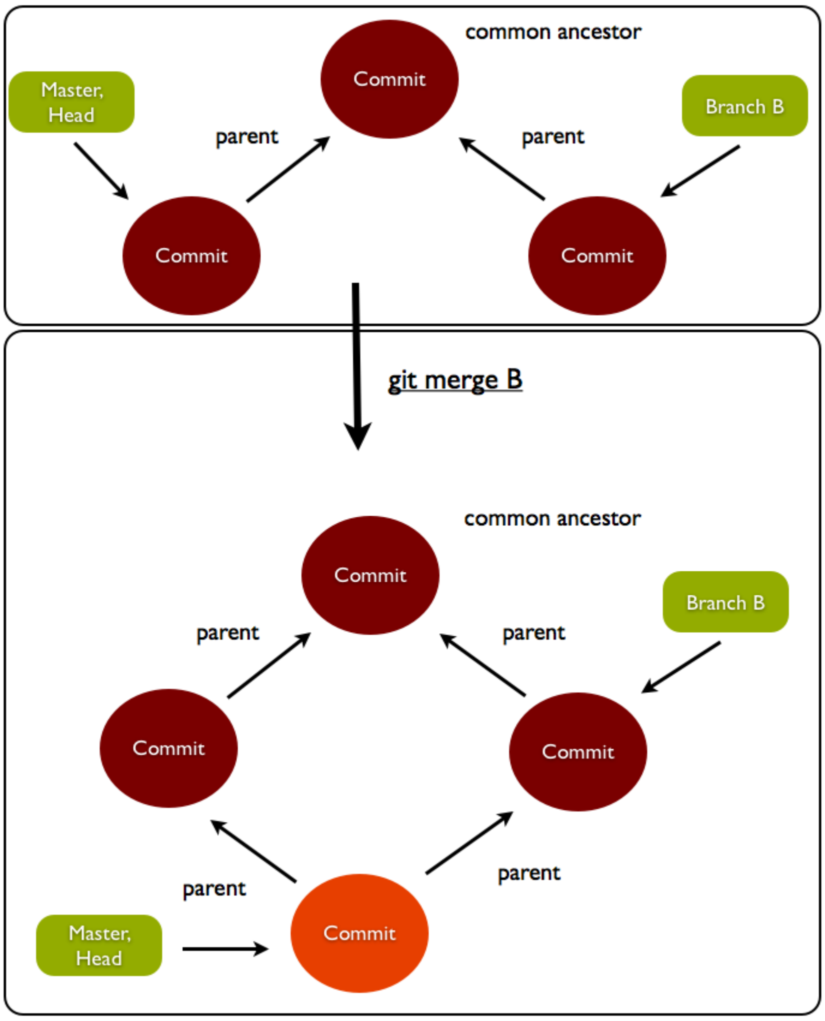
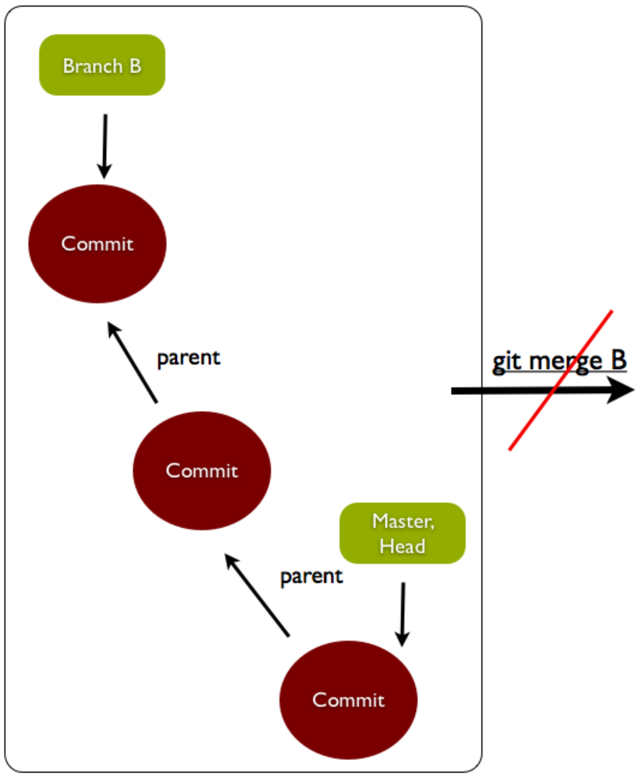


Figure 11: A case where, the merge operation can't be made


Branch B
Commit
parent
Commit
Master, Head
parent
Commit
git merge B