Class exercise: Semantics of git

COMP 150 - Applied Functional Programming

October 17, 2012

The in the filesystem paper we saw two kinds of semantics:

- A denotational semantics that says for each command, how it operates as a function from filesystems to filesystems
- An algebraic semantics that says how to reason about sequences of commands

Our plan for today is to develop similar semantics for git.

Important note: I've listed questions in three separate categories: language, denotational semantics, and algebraic semantics. But each of these is connected to the others. *You will need to iterate.* In particular, as you develop your semantics, you will probably want to change your language. And of course any change in language must be properly accounted for by corresponding changes in the semantics.

The language of git

(1) What is the language of git commands for you will give a semantics?

Denotational semantics of git

In the denotational semantics, let's not model the index. Let's stick to what's happening in the repository.

Git implements what is sometimes called "hash-consing" or "content-addressable memory"—any two objects with the same SHA1 hash share storage. This technique is important for performance, but it's not the only specialized storage technique git uses. For example, git also merges multiple objects into a single Unix file called a "pack." For today, *let's create a model that ignores the details of* how *things are stored.*

(2) Git's low-level semantics, as described in *Git from the Bottom Up*, seems quite imperative in places. How would we make a more functional model?

(3) Develop a denotational model of git operations. Stealing ideas, notation, and equations from things you have read is definitely OK.

Equational reasoning about git

Let's see if we can use algebraic ideas to explain what job the index is doing, so that our model *accounts* for the index but does not necessarily have a special *representation* for the index.

(4) Develop algebraic laws that could be used in equational reasoning about git commands.

(5) Use your algebra to say precisely and formally what is meant by a "base commit"? (*Git from the Bottom Up*, page 15 and page 16).

Perhaps there are functions you could write that would help?

(6) On page 17, *Git from the Bottom Up* says "when W was based on A, it contained only the changes needed to transform A into W." Does git store "changes?" Where? What is actually going on here?

(7) Does *Git from the Bottom Up* explain how rebase works, from the bottom up?

(8) Use your algebra to account for what happens when two parallel lines of development are combined with *rebase*.

Questions to avoid for today

One of the central facts about rebase is that it should never be used on any branch that has been shared. A good model of rebase should be able to account for what goes wrong when rebase is combined with sharing. But I think that's beyond what we should aim for in two hours.

We lack any notion of push, fetch, pull, or any other operation involving multiple repositories. These operations are very important, but for today, it may be best to avoid them.