Class exercise: Modeling git, continued

COMP 150 - Applied Functional Programming

October 24, 2012

We continue developing semantics for git:

- A language of commands, which is not necessarily a subset of `git` Unix commands
- A denotational semantics that says for each command, how it operates as a function from repositories to repositories
- An algebraic semantics that says how to reason about sequences of commands

In our last session, we agreed on roughly this representation of repositories:[1]

```
type Repo = [Branch]
data Branch = B { name :: String
                , commit :: Commit
                }
data Commit = C { tree :: Tree Blob
                , parent :: Maybe Commit
                }
data Tree a = Leaf a | Tree [Tree a]
type Blob   = String
```

The language of commands

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

Denotational semantics of git

We've stipulated that for the time being we'll try to ignore the index, the working tree, and the representation of git objects as files or *packs*. We'll also try to ignore the hash-consing based on SHA1 hashes.

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

   (a) What functions on repositories do your commands denote?

   (b) Can you show that your semantics is consistent with the exposition in *Git from the Bottom Up*?

[1] There was no agreement on metadata, so I've left it out.

Equational reasoning about git

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

Ideally the algebra is powerful enough to explain what's happening with the index and the working tree, so that our model *accounts* for the index but does not necessarily have a special *representation* for the index.

(4) 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?

(5) 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?

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

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

Questions for the future

(8) Can we account for `merge`?

(9) Can our semantics explain in exactly what ways merge is different from rebase, and in what ways they're the same?

(10) Can we use our semantics to explain exactly what is going on with the "recursive merge" strategy that is described informally in the git manual?

(11) Can we account for multiple repositories? Can we provide a semantics of the `fetch`, `push`, or `pull` commands?

(12) Can we account for what goes wrong when a shared branch is rebased or a rebased branch is shared?

(13) Can we define a new model that makes sharing and rebase compatible?