The semantics of version control

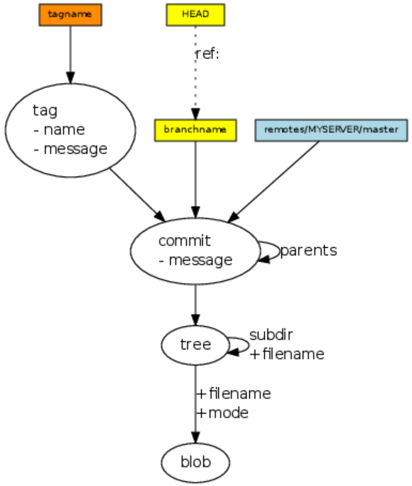
Wouter Swierstra

March 31, 2014

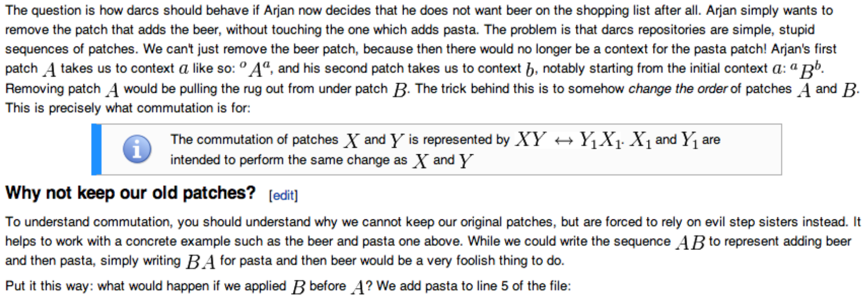
Version control is hard

- There is a variety of complex version control systems:
  - Dropbox
  - CVS
  - Subversion
  - Git
  - Mercurial
  - Darcs
  - …
- How do these systems work?

GIT FOR COMPUTER SCIENTISTS



DARCS PATCH THEORY



The question is how darcs should behave if Arjan now decides that he does not want beer on the shopping list after all. Arjan simply wants to remove the patch that adds the beer, without touching the one which adds pasta. The problem is that darcs repositories are simple, stupid sequences of patches. We can't just remove the beer patch, because then there would no longer be a context for the pasta patch! Arjan's first patch A takes us to context a like so: ${}^{o}A^{a}$, and his second patch takes us to context b, notably starting from the initial context a: ${}^{a}B^{b}$. Removing patch A would be pulling the rug out from under patch B. The trick behind this is to somehow *change the order* of patches A and B. This is precisely what commutation is for:

The commutation of patches X and Y is represented by $XY \leftrightarrow Y_1 X_1$. X_1 and Y_1 are intended to perform the same change as X and Y

Why not keep our old patches? [edit]

To understand commutation, you should understand why we cannot keep our original patches, but are forced to rely on evil step sisters instead. It helps to work with a concrete example such as the beer and pasta one above. While we could write the sequence AB to represent adding beer and then pasta, simply writing BA for pasta and then beer would be a very foolish thing to do.

Put it this way: what would happen if we applied B before A? We add pasta to line 5 of the file:

A question from a colleague

Why work on the semantics of ***version control****? There are plenty of existing* ***tools*** *out there that work just fine.*

A question from a colleague

Why work on the semantics of ***programming languages****? There are plenty of existing* ***languages*** *out there that work just fine.*

Version control and programming languages

- Version control systems manage shared mutable state;
- We have plenty of logics for reasoning about (programs using) shared mutable state:
  - Hoare logic;
  - Separation logic;
  - Refinement calculus;
  - …
- Version control systems should be designed with such a logic in mind.

Terminology

- Version control systems manage a **repository**, consisting of some data stored on disk;
- This data exists on two levels:
  - the *raw data* stored on disk;
  - the *internal model* of this data that is managed by the version control system;
- For example, many version control systems have the following internal model:
  - text files are a (linked) list of lines;
  - binary files are a blob of bits.
  - but timestamps are ignored.

More Terminology

- Users can edit the raw data stored in a repository. The version control system may *observe* these changes to the raw data.
- The corresponding change to the internal model will be called a *patch*.
- A version control system maintains a *history*, the sequence of patches that have lead to the current state.
- Patches may be communicated across repositories – but this does not always work.

A trivial version control system

Suppose we want to define the semantics of a trivial version control system managing a single binary file.

- Define the internal model – how to represent the file?
- Define predicates observing properties of this model – what can we observe?
- Define operations on this model as Hoare triples:
    - initialize the repository;
    - modify the file;
    - delete the file.

INTERNAL MODEL

We define the type of our internal model M, assuming some valid set of file names F:

$$\begin{aligned} \mathsf{M} &:= \varepsilon \mid (\mathsf{F}, \mathsf{Bits}) \\ \mathsf{Bits} &:= (0 \mid 1)^* \end{aligned}$$

Predicates

- If (the internal model of) the repository is (f, c) then we say the predicate $f \mapsto c$ holds;
- If (the internal model of) the repository is ε then we say the predicate $\varnothing$ holds.

We will write $M \vDash p$ when the predicate p holds in the repository M.

Operations

$$\begin{array}{rcl}
\{\varnothing\} & \textsf{create}\,f & \{f \mapsto \varepsilon\} \\
\{f \mapsto c\} & \textsf{replace}\,f\;c\;d & \{f \mapsto d\} \\
\{f \mapsto \varepsilon\} & \textsf{remove}\,f & \{\varnothing\}
\end{array}$$

Sequential composition

We can reason about the sequential composition of two patches c_0 and c_1 using the usual rule from Hoare logic:

$$\frac{\{P\}\, c_0\, \{Q\} \qquad \{Q\}\, c_1\, \{R\}}{\{P\}\, c_0; c_1\, \{R\}}$$

Example:

$$\{\varnothing\}\ \mathsf{create}\, f; \mathsf{replace}\, f\ \varepsilon\ 010\ \{f \mapsto 010\}$$

Example

Another example:

$$\{\varnothing\}\ \mathsf{create}\,f;\mathsf{replace}\,f\ \varepsilon\ 010;\mathsf{replace}\,f\ 010\ \varepsilon;\mathsf{remove}\,f\ \{\varnothing\}$$

Is this the same as *skip*?

CONFLICTS

- Suppose we start with a repository satisfying $f \mapsto 00$;
- Programmer A modifies (his local copy of) f, pushes his patch so the repository now satifies $f \mapsto 10$;
- Programmer B modifies (his local copy of) f to 01. What happens when he tries to push his patch, `replace` 00 01?
- The precondition of his patch is not satisfied – we have a *conflict*.

Handling conflicts

- Programmer B can discard his patch;
- Programmer B can remove Programmer A's patch from their shared repository;
- Programmer B can update his patch to `replace` 10 01;
- or update his patch to `replace` 10 11.

Beyond Hoare logic

- What if we want to have multiple files in our repositories?
- Patches now refer to the *complete* state of the repository…
- which means that you can only share changes if you agree on *everything*…

MULTIPLE FILES IN A SINGLE DIRECTORY

- Define the internal model – how to represent the files?
- Define predicates observing properties of this model – what can we observe?
- Define operations on this model as Hoare triples:
  - add an empty file;
  - modify a file;
  - delete a file.

Internal model

We can model the repository as a partial map from file names to their contents:

$$\mathsf{M} := \mathsf{F} \rightharpoonup \mathsf{Bits}$$

Now we define a pair of predicates on such finite maps:

$$\begin{array}{lll} M \vDash f \mapsto c & \text{iff} & M(f) = c \\ M \vDash f \not\mapsto & \text{iff} & f \notin dom\,(M) \end{array}$$

Operations

$$\begin{array}{rcl} \{f \not\mapsto\} & \textsf{add}\, f & \{f \mapsto \varepsilon\} \\ \{f \mapsto \varepsilon\} & \textsf{remove}\, f & \{f \not\mapsto\} \\ \{f \mapsto c\} & \textsf{replace}\, f\ c\ d & \{f \mapsto d\} \end{array}$$

Frame rule

The predicate $P * Q$ holds if we can split a repository into two disjoint parts that satisfy P and Q respectively.

The *frame rule* from separation logic now states:

$$\frac{\{P\}\, c\, \{Q\}}{\{P * R\}\, c\, \{Q * R\}}$$

Provided $mod(c) \cap fv(R) = \emptyset$

Local reasoning

Suppose one programmer modifies a file f and wants to commit his patch. In the meantime, a second programmer has committed a new empty file g. Do we have a conflict?

$$\frac{\{f \mapsto 1\}\ \mathsf{replace}\, f\ 1\ 0\ \{f \mapsto 0\}}{\{f \mapsto 1 * g \mapsto \varepsilon\}\ \mathsf{replace}\, f\ 1\ 0\ \{f \mapsto 0 * g \mapsto \varepsilon\}}$$

‘Free variables’

We can now define the ‘free variables’ mentioned by our predicates:

$$\begin{aligned} fv\,(f \not\mapsto) &= \{f\} \\ fv\,(f \mapsto c) &= \{f\} \end{aligned}$$

Variables is a bad choice of name. In practice, these will not be variables in the programming language sense, but rather concrete addresses – such as filenames, directory paths, or linenumbers.

MODIFIED ‘VARIABLES’

Next, we define the sets of files modified by the various patches:

$$\begin{aligned}
\mathit{mod}\ (\mathsf{add}\, f) &= \{f\} \\
\mathit{mod}\ (\mathsf{remove}\, f) &= \{f\} \\
\mathit{mod}\ (\mathsf{replace}\, f\ c\ d) &= \{f\} \\
\mathit{mod}\ (c_0; c_1) &= \mathit{mod}\ (c_0) \cup \mathit{mod}\ (c_1)
\end{aligned}$$

Derived operations

We may want to group a series of related patches into one atomic patch:

$$\frac{\{P\}\ c\ \{Q\}}{\{P\}\ \mathsf{atomic}\ c\ \{Q\}}$$

This way we can avoid rolling back into any intermediate state.

Using such an operation we can define new operations:

$$\begin{aligned}&\mathsf{rename}\ f\ g\ c\\ &\quad = \mathsf{atomic}\ (\mathsf{add}\ g; \mathsf{replace}\ g\ \varepsilon\ c; \mathsf{replace}\ f\ c\ \varepsilon; \mathsf{remove}\ f)\end{aligned}$$

and calculate their associated semantics.

What else?

- In the same style we can model:
  - text files as linked lists of lines;
  - (nested) directories;
  - structured data, such as CSV.
- Using *control structures*, we can model the branching and merging of repositories.

Further work

- What is the relationship with bidirectional transformations?
- What are the metatheorems that we can (or should) prove?
- How does this relate to existing revision control systems?
- …