The Semantics of Version Control

Wouter Swierstra

With thanks to Andres Löh, Marco Vassena, and Victor Cacciari Miraldo

Robbert

Workshop on Realistic Program Verification

Workshop on Realistic Program Verification

- ~~Programming~~

Workshop on Realistic Program Verification

- ~~Programming~~
- ~~Verification~~

Workshop on Realistic Program Verification

- ~~Programming~~
- ~~Verification~~
- ~~Realistic~~

Workshop on things Robbert likes

- Formalized metatheory in Coq
- Separation logic
- Collaborative open source software development

Who here uses version control?

Who here is happy with the system they use?

Ask any seasoned developer about a long-delayed merge ...
and watch the blood drain out of his or her face.
- Bryan O'Sullivan[1]

[1] Making Sense of Revision-Control Systems, Communications of the ACM, Vol. 52 No. 9, Pages 56-62

Version control systems

Version control systems are like C compilers:[2]

[2] With apologies to Xavier.

Version control systems

Version control systems are like C compilers:[2]

- they solve a hard problem

[2] With apologies to Xavier.

Version control systems

Version control systems are like C compilers:[2]

- they solve a hard problem
- but it's hard to predict their exact behaviour

[2] With apologies to Xavier.

Version control systems

Version control systems are like C compilers:[2]

- they solve a hard problem
- but it's hard to predict their exact behaviour
- their design can be ad-hoc

[2] With apologies to Xavier.

Version control systems

Version control systems are like C compilers:[2]

- they solve a hard problem
- but it's hard to predict their exact behaviour
- their design can be ad-hoc
- and they don't have a formal semantics.

[2] With apologies to Xavier.

Ancient history (circa 2005)

Before git and mercurial were as popular as they are today...

Darcs is a distributed revision control system, written in Haskell.

The darcs manual contains an appendix specifying *The theory of patches* on which it is based.

Andres Löh

How can we describe version control systems more formally?

A principled approach to version control

Submitted in 2007...

Rejection

A fine example of how to write a bad formal methods paper...

1. *Ignore all previous notations and invent your own...*
2. *Make your new notation as misleading as possible...*
3. *Produce results that are mathematically impressive but completely useless...*

...

What do version control systems do?

They manage access to mutable state.

There are logics for reasoning about this!

Terminology

Version control systems manage a repository, consisting of data stored on disk.

This data exists on two levels:

1. The *raw data* stored on disk;
2. The *internal model* of this data, managed by the VCS

These are two different things.

Common models

For example, most VCS have the following internal model:

- text files are a (linked) list of lines;
- binary files are blobs of bits;
- each file has permissions (which are tracked)
- but timestamps are ignored.

Back to programming languages

- A VCS's internal model is a 'heap'
- A *patch* is some change to a repositories internal model, that may shared between repositories.
- Patches modify the 'heap' – we should define their semantics using a suitable logic.

A trivial version control system

Define a version control system that tracks a single binary file.

- What is the internal model?
- What predicates can we formulate that observe properties of the model?
- What operations are there on this model?

Internal model

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

$$M := \epsilon \quad | \quad (F, \mathbf{Bits})$$
$$\mathbf{Bits} := (0|1)^*$$

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 $M \models p$ when M satisfies the predicate p.

Operations

We can define three operations that manipulate the repository as **Hoare triples**:

$$\{\varnothing\} \quad \textbf{create } f \quad \{f \mapsto \epsilon\}$$
$$\{f \mapsto c\} \ \textbf{replace } f\ c\ d \ \{f \mapsto d\}$$
$$\{f \mapsto \epsilon\} \quad \textbf{remove } f \quad \{\varnothing\}$$

Sequential composition

We can now combine patches using the familiar rules for sequential composition of statements:

$$\frac{\{P\}c_1\{Q\} \qquad \{Q\}c_2\{R\}}{\{P\}c_1;c_2\{R\}}$$

Such a sequence of patches records the *history* of a repository.

Conflicts

When applying a patch $\{P\}\ c\ \{Q\}$ to a repository M , for which $M \not\models P$, we say that c causes a **conflict** in the repository M.

This definition does not mention Alice and Bob.

What about multiple files?

- Hoare logic requires the pre- and postconditions to specify the *entire* heap.
- This does not scale to more complex repository models...

Separation logic

Internal model & predicates

Suppose we want to model a repository with multiple binary files.

The internal model is a partial map from filenames to bits:

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

There are two predicates:

$$M \models f \mapsto c \quad \text{iff} \quad M(f) = c \wedge dom(M) = \{f\}$$
$$M \models \varnothing \quad \text{iff} \quad dom(M) = \emptyset$$

Operations

$$\{\varnothing\} \quad \textbf{add } f \quad \{f \mapsto \epsilon\}$$
$$\{f \mapsto \epsilon\} \quad \textbf{remove } f \quad \{\varnothing\}$$
$$\{f \mapsto c\} \quad \textbf{replace } f\ c\ d \quad \{f \mapsto d\}$$

These preconditions refer to the smallest possible footprint.

How can we add files to a non-empty repository?

Separating conjunction

The separating conjunction $M \models P * Q$ holds iff we can partition M into two disjoint parts, M_0 and M_1, such that $M_0 \models P$ and $M_1 \models Q$.

The frame rule

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

Provided c does not modify files mentioned by R.

Of course, we need to prove soundness of the frame rule for our system (and have formalized the proof in Coq).

The frame rule

We can use the frame rule to add new files to non-empty repositories:

$$\frac{\{\varnothing\}\ \mathbf{add}\, f\ \{f \mapsto \epsilon\}}{\{\varnothing * R\}\ \mathbf{add}\, f\ \{f \mapsto \epsilon * R\}}$$

Provided R does not mention f – in other words, we can add a file to any repository not yet containing f.

Independence

Using the frame rule we can specify when two patches are **independent** – that is they modify different parts of the repository.

Lemma: Independent patches commute.

This formalizes the intuition that you can avoid conflicts by working on different files.

Beyond binary files

Of course, restricting ourselves to binary files is unrealistic.

Realistic version control systems must handle text files, built from individual lines.

Can we use the same mathematical structures to model this?

Let's start by restricting ourself to a *single* text file.

A dead end

We could model our file as a finite map from lines of text to their contents:

$$M := n \rightharpoonup \mathbf{ASCII}^*$$

But inserting or deleting lines require modifying all subsequent lines – they need to be shifted up or down.

Such invasive changes are likely to cause unnecessary conflicts.

A better approach

Rather than model the lines as a 'fixed sized array', we want to represent the file as a *linked list*.

Separation logic is specifically designed for reasoning about pointers and complex memory structures.

Lines of text

Given some (abstract) type I representing the labels for every line, we can define a new model for our repository:

$$A ::= I \times \{0, 1\}$$
$$M ::= A \rightharpoonup (ASCII + I)$$

Every model associates with a line labelled by l:

- the line contents at 'heap location' $(l, 0)$
- the next line at 'heap location' $(l, 1)$.

Predicates

As we saw previously, we can choose two basic predicates to describe the internal model of a repository:

$$M \models a \mapsto c \quad \textbf{iff} \quad M(a) = c \wedge dom(M) = \{a\}$$
$$M \models \varnothing \quad \textbf{iff} \quad dom(M) = \emptyset$$

We will sometimes write:

$$M \models l \to l' \quad \textbf{iff} \quad M \models (l, 0) \mapsto l'$$
$$M \models l \mapsto c \quad \textbf{iff} \quad M \models (l, 1) \mapsto c$$

Operations

We can define three operations to manipulate the file:

$$\{l_b \to l_a\}\ \textbf{insertLine}\ l_b\ l\ l_a\ \{(l_b \to l \to l_a) * (l \mapsto \epsilon)\}$$

$$\{l \mapsto c\}\ \textbf{modifyLine}\ l\ c\ d\ \{l \mapsto d\}$$

$$\{(l_b \to l \to l_a) * (l \mapsto \epsilon)\}\ \textbf{deleteLine}\ l_b\ l\ l_a\ \{l_b \to l_a\}$$

Observations

- Once we prove soundness of the frame rule, we can re-use our previous results – independent patches still commute;
- This opens the door to more clever pointer tricks, such as swapping the contents of two lines.

What else?

We can model:

- (nested) directories;
- metadata, such as file permissions;
- using *control flow*, like conditionals, we can mimic branching and merging – even if I'd like a more convincing story here.

What next?

- Does it scale?
- All these semantics have the same structure, can we exploit this to define more realistic systems modularly?
- Can we define an algebraic semantics that is sound with respect to the separation logic semantics?

Beyond lines of text

'All' version control systems are based around traditional Unix tools such as `diff`.

These tools work very well if you're interested in tracking line-based changes – such as changes to C programs.

But this can lead to strange behaviour...

Example: comma-separated-values

```
Name,    Mark
Alice,   8
Bob,     6
Carroll, 7
```

Example: comma-separated-values

Name, Mark, **Date**
Alice, 8, **1/12/2015**
Bob, 6, **1/12/2015**
Carroll, 7, **1/12/2015**

Example: comma-separated-values

Name,	Mark,	**Date**
Alice,	8,	**1/12/2015**
Bob,	6,	**1/12/2015**
Carroll,	**7.5**,	**1/12/2015**

Conflict!

Version control of (semi)structured data

Apply programming technology to this domain:

- A domain specific language for defining file formats
- Generate parser & pretty printer
- Generate diff and merge algorithms

Using *datatype generic programming*!

Closure

As the fruits of programming-language research become more widely understood, programming is going to become a much more mathematical craft.

– John Reynolds

Closure

As the fruits of programming-language research become more widely understood, programming is going to become a much more mathematical craft.

– John Reynolds

We would love the same to be true of software development.

Questions