The Semantics of Version Control

Wouter Swierstra Andres Löh
Universiteit Utrecht Well-Typed LLP
w.s.swierstra@uu.nl andres@well-typed.com

Abstract

As software becomes increasingly complex, software configuration management is becoming ever more important. This paper shows how logics for reasoning about mutable state, such as separation logic, can also be used to give semantics for version control systems. By applying these ideas from the programming language research community, developers may reason formally about the broader software development process.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Version control

1. Introduction

Version control is hard. Today's version control systems are expected to manage huge source trees with multiple branches, shared by hundreds of developers spread all over the world. In his overview of article in the *Communications of the ACM* [1], Bryan O'Sullivan wrote:

> Much work could be done on [version control systems'] formal foundations, which could lead to more powerful and safer ways for developers to work together.

This paper addresses this challenge. More concretely, there is one key point that this paper makes:

> *Version control systems manage the access to (shared) mutable state. Therefore they should be designed using a logic for reasoning about such state.*

After all, version control systems maintain a history of changes that lead to the current state of the files on disk. You can think of this sequence of changes as an imperative program that 'computes' the current state of the files under version control. Reasoning about the behaviour of version control systems is closely related to reasoning about imperative programs, for which a great many program logics have been developed. This paper explores this correspondence, making the following novel contributions:

- After giving a bit of background on the history of version control (Section 2), we need to fix certain terminology (Section 3).
- To illustrate the version control problem, we give a mathematical description of a trivial version control system and define the behaviour of this system using *Hoare logic* [2, 3] (Section 4). This trivial version control system can only manage a single binary file. While impractical, studying this example enables us to define abstract notions such as *conflicts* precisely and unambiguously.
- To describe how version control systems manage multiple binary files (Section 5), text files (Section 6), and directories (Section 7), we need to use a richer *separation logic* [4, 5]. We demonstrate how the *frame rule* from separation logic is the key ingredient to reasoning about the interaction between different patches.
- Finally, we show how we can describe different *branches* in a version control system using the same logic to reason about statements affecting a program's control flow (Section 8). We provide a *generic condition* that is sufficient to guarantee the absence of *merge conflicts* between branches.

In each of these sections we will use established technology from the programming language community. The most important contribution this paper makes is implicit throughout: ideas from the programming language community may also have important applications in other areas such as software engineering and version control.

2. Background

One of the earliest version control systems is the Source Code Control System (SCCS) developed at Bell Labs in the early 1970s [6]. At the time, SCCS was considered 'a radical departure from conventional methods for controlling source code.' SCCS maintains a series of *deltas*, the line-based changes associated with a particular file. The files un-

Onward! 2014, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3210-1/14/10…$15.00.
http://dx.doi.org/10.1145/2661136.2661137

der version control are stored on a central mainframe, which each developer accesses from a workstation. When a developer needs to change a file, that file is locked until the developer is finished. The theory behind SCCS was published in a research paper [6], carefully documenting how deltas are created, chained together, and applied.

In the early 1980s, the Revision Control System (RCS), based on the popular Unix *diff* utility, gained more traction. Later the Concurrent Versions System (CVS) used RCS to manage a *repository*, recording the history of several files [7–9]. Until the turn of the century, CVS was the most popular version control system. Using CVS developers 'check out' the current state of a repository, stored on some central server, copying the repository to their machine. This local copy of the repository is sometimes called the *working copy*. After locally making modifications to the source files in the working copy, developers can 'check in' their changes, sending them back to the central server, thereby sharing their changes with other developers. This was a major advance compared to SCCS. Different developers can work on the local copy of the same file, instead of having to acquire a global lock as is the case in SCCS. Nowadays CVS has been outdated by several more modern revision control systems. Notably, the Subversion version control system [10–12] has been specifically developed to replace CVS.

More recently, *distributed revision control systems* are becoming increasingly popular. In contrast to CVS and Subversion, distributed revision control systems treat every working copy as a repository in its own right. Instead of communicating through a single, centralized server, each repository may share changes with any other repository. Around 2005 several distributed revision control systems saw the light of day, including GNU arch, darcs [13], mercurial [14] and Git [15]. Of these systems, mercurial and Git have developed into two of the most widely used revision control systems in the open source community as witnessed by the popularity of websites such as GitHub and Bitbucket.

3. Terminology

In the remainder of this paper, we will show how to develop a formal semantics for a series of increasingly complex version control systems. The aim of this paper is *not* to formalize any particular existing revision control system, but instead to try and find a suitable formalism for the general problem. Before we do so, however, we need to fix some terminology and make several deliberate simplifications.

A *repository* manages (some collection of) data stored on disk, varying from a single file to a complete source tree. This data exists on two levels: the *raw data*, consisting of files and directories on your hard drive, and the *internal model* of this information that the version control system maintains. For example, many version control systems model text files as a sequence of lines and binary files as a blob of bits. Similarly, most version control systems do record changes to file permissions, but ignore modification timestamps.

There is a close relationship between the raw data and the version control system's internal model. When a user changes the raw data, that change must somehow percolate into the model—a process we shall refer to as *observation*. Typically a user will modify the files stored in a repository, before recording these changes in (the internal model of) the version control system. We will refer to such a change to the model as a *patch*. The *history* maintained by a version control system consists of the sequence of patches that led to the current state of the internal model.

Patches can be moved between repositories. Applying a patch to a repository consists of extending the history, updating the internal model of the repository, and *interpreting* the patch as a change to the raw data. Adding a patch may lead to a *conflict*, when there is some inconsistency between the internal model of the repository and the model that is assumed by the patch. For example, the patch may try to remove a file that does not exist. Usually there is some form of user interaction necessary to resolve a conflict and return the repository to a consistent state.

In this paper, we will not discuss the raw data stored in a repository, observation, or interpretation. Instead, we focus on internal models and patches—that is where the key design of a version control system is expressed.

Many other notions usually found in version control systems are not part of our core model. We do not distinguish between centralized and distributed version control systems. In principle, any patch may be moved freely between any two repositories; heedlessly doing so will often result in conflicts. Centralized version control systems may be modeled as a special case where all repositories communicate via a single centralized server. Some version control systems may maintain additional information, such as the permissions of users or a certain relation between repositories. Such policies are specific to a version control system's implementation and we will not consider them further.

4. Managing a single file

In this section, we will describe the model associated with a very simple version control system and the operations it supports. Let us start by considering a version control system only capable of managing a single binary file. In later sections, we will extend this incrementally into a full-fledged system handling text files, directory structure and file renames. It is important to emphasize that we are not advocating any particular version control system, but rather exploring suitable formalisms for defining the semantics of version control systems.

Internal model

We assume there is a set F of all valid file names. For this very first example, the internal model of a repository is either empty or a pair of the name and binary contents of the file under version control. We give the following more formal definition:

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

Next we introduce two predicates on these models. If M is the repository (f, c), the predicate $M \vDash f \mapsto c$ holds, that is, the contents of the file f in the repository consists of the sequence of bits c. Similarly, the predicate $M \vDash \varnothing$ holds if and only if the repository M is the empty repository ε.

Note that we write the type of the internal model of our repository in a sans-serif font, M, while inhabitants of this type are written in italics M. We will adhere to this convention throughout this paper.

Operations

Now we can begin to define operations on repositories. In particular, we consider the following three operations on repositories:

- creating a new repository storing an empty file,
- removing an empty file from the repository,
- and updating the contents of a file in the repository.

We will define how these operations affect the model of the repository by giving their semantics as *Hoare triples*. We can choose the following pre- and postconditions for these three operations:

$$\begin{array}{rcl}\{\varnothing\} & \mathsf{add}\, f & \{f \mapsto \varepsilon\}\\ \{f \mapsto \varepsilon\} & \mathsf{remove}\, f & \{\varnothing\}\\ \{f \mapsto c\} & \mathsf{replace}\, f\ c\ d & \{f \mapsto d\}\end{array}$$

When the file f does not exist, the command $\mathsf{add}\, f$ creates f in the repository with an initially empty contents. Dually, $\mathsf{remove}\, f$ removes f from the repository, provided that the empty file f exists in the repository. Finally, the command $\mathsf{replace}\, f\ c\ d$ changes the contents of a file from c to d.

Usually, the pre- and postconditions in such Hoare triples are predicates on (a mathematical model of) the computer's memory. The pre- and postconditions we have given here are predicates on the internal model of a repository.

History

The *history* stored by a version control system is a sequence of patches. We can assign semantics to the complete history using the usual law for sequential composition in Hoare logic:

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

One important *invariant* we will enforce is that the history of a repository can always be assigned a sensible semantics. This is not the case for all sequences of patches. For example, the sequence of patches $\mathsf{replace}\, f\ 0\ 1; \mathsf{replace}\, f\ 0\ 1$ is not a valid history, given the rules we have seen so far.

Repository

We have still not given a formal definition for a repository. For the purpose of this paper, it suffices to identify a repository with its history, i.e., the sequence of patches that have been applied so far. In addition to the history, it may be useful to cache the current state of the internal model of a repository. Although this can always be recomputed from its history, caching the current state has obvious performance benefits. Realistic systems may want to track other information, for instance regarding permissions or preferences of users, but we will not try to model such information in this paper.

Discussion

The following sequence of patches creates a repository storing the file f, and sets its contents to the bit sequence 010:

$$\mathsf{add}\, f; \mathsf{replace}\, f\ \varepsilon\ 010;$$

It is easy to check that the semantics of this sequence is given by the following Hoare triple:

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

If we choose to apply the patch $\mathsf{replace}\, f\ 010\ \varepsilon; \mathsf{remove}\, f$, the repository now consists of the following history:

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

Although the internal model of the repository is empty both before and after executing this sequence of patches, it is important that the repository stores the entire history to allow users to *rollback* to previous states. If a user decides that the latest patch was undesirable, it can be removed from the history, resulting in a repository satisfying $f \mapsto 010$.

Although we only allow the creation and deletion of empty files, there is no fundamental reason to do so. We could just as well have permitted alternative operations:

$$\begin{array}{rcl}\{\varnothing\} & \mathsf{add}\, f\ c & \{f \mapsto c\}\\ \{f \mapsto c\} & \mathsf{remove}\, f\ c & \{\varnothing\}\\ \{f \mapsto c\} & \mathsf{replace}\, f\ c\ d & \{f \mapsto d\}\end{array}$$

In the version control system presented previously, we favour the simplicity of primitive patches over their expressivity. Many alternative choices certainly exist. This paper, however, focuses on the semantic framework for creating and comparing such designs, rather than the individual merits of a particular version control system.

Conflicts

Using this abstract model, we can also reason about *conflicts* that may arise from users editing the same file. For instance, suppose we consider a repository storing the sequence 00 shared by two users Alice and Bob. Suppose Alice commits a patch corresponding to the operation $\mathsf{replace}$ $\mathsf{README.md}$ 00 10. What will happen if Bob tries to submit a patch to the same repository flipping the other bit, that is, corresponding to the operation replace README.md 00 01? In the system described here, this is not allowed. The precondition of Bob's operation is not satisfied—and hence it cannot be added to the repository's history. There is a *conflict*. There are several ways to resolve this conflict:

- Bob could discard his change, rolling back to the previous state of his repository after Alice's patch;
- Bob could decide that he wants to override Alice's change. In that case, he would change his patch to correspond to the operation replace README.md 10 01.
- Bob could remove Alice's patch from the shared repository and push his own patch;
- Bob could resolve the conflict manually, by changing his patch. For example, the patch replace README.md 10 11 incorporates both changes to the individual bits.

Each of these solutions has its own advantages and disadvantages. The point we wish to make here, however, is that we can the following *semantic characterization* of what a conflict is:

Definition 1. *When applying a patch $\{P\}\ c\ \{Q\}$ to a repository M, for which $M \nvDash P$, we say that c causes a* conflict *in the repository M.*

Note that we can give this definition independently of how the repositories involved communicate, the merging algorithm used, or even the raw data stored in the repository.

Furthermore, we have not said anything about how these patches are constructed from the changes to the raw data stored in a repository. Nor have we discussed how these patches are communicated between repositories. We have also not discussed whether the version control system is distributed or centralized. This is a good thing! The semantics we present is completely independent of these design decisions, as it should be.

This version control system is far too simple to be of any practical use. Yet even in this simple scenario, there are plenty of design choices. Should the patches be invertible? Are there any other operations necessary? Is there really a conflict if two separate bits are modified? We will revisit many of these questions in the coming section.

This example does illustrate the central idea of this paper that program logics may be used to reason about version control systems. To be of any real use, a version control system must store multiple files. To do so, we need to use a richer logic.

5. Separation logic

The previous version control system is extremely limited. Suppose we would like to extend this initial system to handle multiple files. Recall that the pre- and postconditions are predicates on (some internal model of) a repository. To create a new patch, you need to refer to the full state of the repository—even if this patch only changes a single file. This limits how repositories can share information: in order to validate that a patch can be incorporated in a repository, the patch must start from precisely the same initial repository.

This may sound like a familiar problem: separation logic was introduced to address precisely this shortcoming in traditional Hoare logic. In this section, we will develop a marginally more realistic system, capable of storing multiple binary files.

Internal model

Let us revise the simple repository model we defined previously. Our new internal model of a repository consists of a (finite) partial map from filenames to their contents:

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

Just as before, we will define a pair of predicates on repositories. We write $M \vDash p$ when some internal model of a repository M satisfies the predicate p:

$$\begin{array}{lll} M \vDash f \mapsto c & \text{iff} & M(f) = c \wedge dom(M) = \{f\} \\ M \vDash \varnothing & \text{iff} & dom(M) = \emptyset \end{array}$$

Intuitively, the predicate $f \mapsto c$ holds on the repository modeled by M when the repository only tracks the file f with contents c. Similarly, the predicate $\varnothing$ holds on the repository modeled by M when the repository M is empty. In addition to these atomic predicates, we define a compound predicate $P * Q$, inspired by the separating conjunction from separation logic:

$$\begin{array}{lll} M \vDash P * Q & \text{iff} & \exists M_0, M_1 .\, M = M_0 \# M_1 \\ & & \qquad \wedge M_0 \vDash P \wedge M_1 \vDash Q \end{array}$$

Here we write $M_0 \# M_1$ for the union of two disjoint finite maps. Put in words, the predicate $P * Q$ holds on the repository M precisely when we can split M into two disjoint maps M_0 and M_1 such that M_0 satisfies P and M_1 satisfies Q. Note that the separating conjunction operator is commutative and associative.

Operations

Just as before, we define three operations on repositories:

$$\begin{array}{ccc} \{\varnothing\} & \mathsf{add}\, f & \{f \mapsto \varepsilon\} \\ \{f \mapsto \varepsilon\} & \mathsf{remove}\, f & \{\varnothing\} \\ \{f \mapsto c\} & \mathsf{replace}\, f\ c\ d & \{f \mapsto d\} \end{array}$$

The operation add f adds the file to the repository, provided it does not exist. The remove operation removes a file from the repository. Finally, the replace operation modifies the contents of a file in the repository. Once again, we would like to stress that different choices for these operations exist. For instance, the add operation could be allowed to overwrite the file if it exists already. We do not claim that this is the best way to manage a single repository containing binary files, but rather aim to illustrate that this is a good way to define such a version control system's semantics.

Separation logic extends Hoare logic with the *frame rule*:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ mod(c) \cap addr(R) = \emptyset$$

Here $addr(R)$ denotes the files mentioned by the predicate R and the *mod* function computes the set of modified files associated with every command. As we are only interested in the files being modified, we define the *addr* function as follows:

$$\begin{aligned} &addr\ (\varnothing) &&= \emptyset \\ &addr\ (f \mapsto c) &&= \{f\} \\ &addr\ (P * Q) &&= addr\ (P) \cup addr\ (Q) \end{aligned}$$

Traditionally, the side condition of the frame rule refers to the free variables of the predicate, but this name is a bit misleading in this context. Version control systems, in contrast to programming languages, typically do not have a notion of 'variable.' Instead, they refer to files by their name. An analogy might be to refer to program variables by their memory address. For that reason, we use the *addr* function computing the set of 'addresses' mentioned by a predicate, rather than the traditional *fv* function computing the free variables. In this example, these addresses correspond to file names. In later examples, they may also refer to the individual lines of a file, for instance.

The *mod* function computes the set of file names modified by a series of operations and is defined as follows:

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

The frame rule is the central rule of separation logic: it 'codifies a notion of local behaviour' [5], which allows us to reason about the effects of a single patch on different repositories.

We still need to show that the frame rule, as formulated above, is *sound*. By modifying a standard denotational model of mutable state in type theory [16, 17], we have developed a denotational model for these semantics in the Coq proof assistant [18]. That is, we have defined a type M, representing the internal models of a repository. Furthermore, we define a type of patches p. The denotational semantics of a patch p corresponds to a function of type $[\![c]\!] : M \to M$, modifying a repository. We have shown that our semantics and the frame rule are sound with respect to this model:

Proposition 1 (soundness). *For all patches $\{P\}\ c\ \{Q\}$ and any internal model of a repository M, if $M \vDash P$ then $[\![c]\!]\ (M) \vDash Q$.*

Proposition 2 (soundness of frame rule). *For all patches $\{P\}\ c\ \{Q\}$, internal model of a repository M, and for any predicate R if $M \vDash P * R$ then $[\![c]\!]\ (M) \vDash Q * R$, provided R and c satisfy the condition:*

$$mod\ (c) \cap addr\ (R) = \emptyset$$

associated with the frame rule.

Discussion

To illustrate how you may reason with these rules, we will cover a few examples in some more detail. Firstly, it may seem from the rules we have given above that you may only ever add files to the empty repository. Using the frame rule, however, we can provide the following derivation:

$$\frac{\{\varnothing\}\ \mathsf{add}\, f\ \{f \mapsto \varepsilon\}}{\{\varnothing * R\}\ \mathsf{add}\, f\ \{f \mapsto \varepsilon * R\}}\ f \notin addr(R)$$

We may always add a file f to any repository that does not yet contain f.

Let us consider at another example. Suppose the repository R holds a single file f with contents 1. Once again Alice and Bob apply patches to this repository. First, Alice applies the patch $\mathsf{add}\ g$. Suppose Bob would like to apply the patch $\mathsf{replace}\, f\ 1\ 0$. Is there now a conflict? Using the frame rule we can derive:

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

Hence applying Bob's patch will not yield a conflict, but results in a repository satisfying the predicate $f \mapsto 0 * g \mapsto \varepsilon$.

The patches that Alice and Bob wrote also commute: they can be applied in either order to produce the same repository. This is not always the case of course, for example, if they both tried to modify the same file their patches would not commute. More generally, we call such patches *independent*.

Definition 2. *We call two patches $\{P_1\}\ c_1\ \{Q_1\}$ and $\{P_2\}\ c_2\ \{Q_2\}$* independent *iff*

$$\begin{gathered} mod(c_1) \cap addr(P_2 \wedge Q_2) = \emptyset \\ \text{and } mod(c_2) \cap addr(P_1 \wedge Q_1) = \emptyset \end{gathered}$$

Using the frame rule, we can prove that for all pairs of independent patches c_1 and c_2, the composite patches $c_1; c_2$ and $c_2; c_1$ exist and commute. We give one half of the derivation below:

$$\frac{\dfrac{\{P_1\}\ c_1\ \{Q_1\}}{\{P_1 * P_2\}\ c_1\ \{Q_1 * P_2\}} \qquad \dfrac{\{P_2\}\ c_2\ \{Q_2\}}{\{Q_1 * P_2\}\ c_2\ \{Q_1 * Q_2\}}}{\{P_1 * P_2\}\ c_1; c_2\ \{Q_1 * Q_2\}}$$

Having a formal semantics makes it possible to *prove* such properties. We will return to this point in Section 8 where we consider branching and merging more generally.

With this basic system in place, it is easy to define new operations on repositories. For example, you may want to facilitate the renaming of files. To do so, we can define a new operation rename with the semantics below.

$$\{f \mapsto c * g \not\mapsto\}\ \mathsf{rename}\, f\ g\ c\ \{f \not\mapsto * g \mapsto c\}$$

Although we could add rename as a new primitive operation, we can also define it using the primitive operations we have seen so far:

$$\mathsf{add}\ g; \mathsf{replace}\ g\ \varepsilon\ c; \mathsf{replace}\, f\ c\ \varepsilon; \mathsf{remove}\, f$$

Using the semantics we have presented, we can check that this definition has the desired pre- and postconditions.

This version control system is still very much a toy example. In the next sections, we will show how to extend these ideas further to a more realistic system.

6. Beyond binary files

Most version control systems record line-based changes on text files, in addition to the monolithic changes to binary files we have seen so far. In this section, we will show how to model another simple version control system that stores (the lines of text of) a single file. For the moment, we leave the name of the file unspecified and focus exclusively on handling the lines of text. It is important to stress that there is no fundamental limitation that prevents us from handling multiple files, or even multiple directories as we will see shortly. We will consider only a single file for the sake of presentation.

Internal model

We begin by describing the internal model of this system. Once again, there are many different design choices possible. We will represent the file under version control as a singly linked list of lines. Other alternatives exist, but we defer further discussion to the end of this section.

Let l be some set of labels, that we use to identify uniquely the lines in our file. We will assume the existence of two distinguished elements of l, init and eof, corresponding to the first and last line of the file under version control. From the set of line labels l, we construct the set $\mathsf{l} \times \{0,1\}$, which we use as our address space A. We now define the internal model of our repository M as follows:

$$\begin{array}{ll} \mathsf{M} & ::= \mathsf{A} \rightharpoonup (\mathsf{ASCII} + \mathsf{l}) \\ \mathsf{A} & ::= \mathsf{l} \times \{0,1\} \\ \mathsf{ASCII} & ::= (\texttt{'a'} \mid \texttt{'b'} \mid \ldots)^* \end{array}$$

The internal model consists of a map describing the contents of the file, that maps addresses in A to either the subsequent label or the line contents, represented by the sum type $\mathsf{ASCII} + \mathsf{l}$. Just as other models of linked lists [4], we will maintain the invariant that for every $i \in \mathsf{l}$, $M\ (i,0)$ maps to the text on the line i and $M\ (i,1)$ maps to the address of the next line. Using a pair of finite maps or dependent types [19], we could enforce this invariant, but we refrain from doing so here for the sake of simplicity. We will sometimes write $contents(i)$ and $next(i)$ rather than $(i,0)$ and $(i,1)$ respectively.

Using this internal model we now define the following two predicates:

$$\begin{array}{lll} M \vDash a \mapsto c & \text{iff} & M(a) = c \wedge dom(M) = \{a\} \\ M \vDash \varnothing & \text{iff} & dom(M) = \emptyset \end{array}$$

We will sometimes write:

$$\begin{array}{l} M \vDash l \to l' \text{ iff } M \vDash next\ (l) \mapsto l' \\ M \vDash l \mapsto c \text{ iff } M \vDash contents\ (l) \mapsto c \end{array}$$

Here we overload the notation $x \mapsto c$, where x may be either in l or A. It will always be possible to infer the necessary definition from the context. Furthermore, we will sometimes use the shorthand $l_0 \to l_1 \to l_2$ for the predicate $(l_0 \to l_1) * (l_1 \to l_2)$.

An initial repository storing an empty file satisfies the predicate init $\to$ eof, but leaves the contents of both init and eof empty. This does not require that all files necessarily contain an empty line. It is up to the version control systems' interpretation function to map between text files and this representation. If the intepretation function traverses the linked list and concatenate all the strings stored therein, the empty repository is mapped to an empty file. If it explicitly inserts newline characters between the elements of the list, the empty repository is mapped to a file with a single empty line.

Operations

We now introduce three operations for inserting new lines, modifying existing lines, and deleting empty lines.

$$\begin{array}{rll} \{l_b \to l_a\} & \mathsf{insertLine}\ l_b\ l\ l_a & \{(l_b \to l \to l_a) \\ & & \quad * (l \mapsto \varepsilon)\} \\ \{l \mapsto c\} & \mathsf{modifyLine}\ l\ c\ d & \{l \mapsto d\} \\ \{(l_b \to l \to l_a) & \mathsf{deleteLine}\ l_b\ l\ l_a & \{l_b \to l_a\} \\ * (l \mapsto \varepsilon)\} & & \end{array}$$

Where l_b, l, and l_a are distinct labels. Each of these operations has the obvious interpretation: calling insertLine $l_b\ l\ l_a$ inserts a new empty line between the subsequent lines l_b and l_a; secondly, modifyLine $l\ c\ d$ modifies the contents of the line labeled l from c to d; and finally, deleteLine $l_b\ l\ l_a$ deletes the empty line labeled l, situated between lines l_b and l_a.

To complete the definition of our version control system, we still need to define three new clauses for the *mod* function:

$$\begin{array}{ll} mod\ (\mathsf{insertLine}\ l_b\ l\ l_a) & = \{next(l_b), next(l), contents(l)\} \\ mod\ (\mathsf{modifyLine}\ l\ c\ d) & = \{contents(l)\} \\ mod\ (\mathsf{deleteLine}\ l_b\ l\ l_a) & = \{next(l_b), next(l), contents(l)\} \end{array}$$

The definition of the *addr* function is unchanged. Once again, we have developed a denotational model in Coq and proven soundness of our semantics and the frame rule.

Discussion

The entire text of the original file can be recovered by concatenating all the strings in the list, starting from the location init until we hit eof. Note that the mathematical model does not mention *lines* of text anywhere—it could model words, sentences, or paragraphs equally well.

There is a great deal of freedom in the design of the primitive operations. We have tried to follow the operations that current *diff*-based version control systems support closely. In principle, however, we can allow other operations, such as swapping the contents of two lines. One possible definition of such an operator could be:

$$\{\, l_0 \mapsto c_0 * l_1 \mapsto c_1 \,\}\ \mathsf{swap}\ l_0\ l_1\ c_0\ c_1\ \{\, l_0 \mapsto c_1 * l_1 \mapsto c_0 \,\}$$

This operator modifies the contents of both lines. This patch records the contents of both of lines. If another user has changed the contents of the lines involved, this patch will cause a conflict. On the other hand, it *only* mentions the lines l_0 and l_1. It is completely independent of the surrounding lines.

Alternatively, we could also choose the following pre- and postconditions:

$$\begin{array}{l}\{\, l_0 \to l_1 \to l_2 * k_0 \to k_1 \to k_2 \,\}\\ \quad \mathsf{swap}\ l_0\ l_1\ l_2\ k_0\ k_1\ k_2\\ \{\, l_0 \to k_1 \to l_2 * k_0 \to l_1 \to k_2 \,\}\end{array}$$

This second choice is interesting: it works regardless of the contents of the lines. If other patches change the contents of the lines, there will not be a conflict. The drawback of this choice, however, is that it mentions the surrounding lines. If these surrounding lines are themselves swapped or deleted, the precondition of this swap operation may fail to hold, causing a conflict. This is similar in spirit to darcs *replace* patches, that express a limited form of lexical substitution, independent of the exact contents of a file.

Once again, having a formal semantics makes it possible to compare the tradeoffs between these two design choices. Without a precise semantics, it is hard to even talk about such issues. The purpose of this paper is not to claim that either of these two alternatives is superior, but rather demonstrating that there is an important design choice to be made here. Version control systems could even offer both alternatives, leaving it to users to specify which operation they wish to apply.

There are, of course, alternative ways to model lines of text. One obvious choice would be to model the lines of text as an array. This choice has several serious drawbacks. Most text based version control systems record line insertions and deletions, using tools such as Unix's *diff*. Naively implementing such insertions and deletions on arrays typically requires the copying of data to free a new location upon insertion or the overwrite an empty location upon deletion. Moving around large portions of the file in this manner greatly increases the set of modified addresses associated with each operation, which in turn, makes it harder to apply the frame rule and increases the chance of conflicts. For that reason, we have chosen to represent the lines as a linked list rather than an array. There may be situations however where a model using arrays may be preferable, for example, if it is important to track absolute line numbers.

7. Directories

Any model of a realistic version control system should be able to handle multiple files and directories. Here we outline how to model such a system, once again, by defining the internal model of the repository, predicates over this model, and primitive operations as Hoare triples. Although we choose to model binary files only, it is certainly feasible to extend this model to include text files as described in the previous section.

Repository

Once again, we will assume a set of names F that is used for the names of both files and directories. We model a repository as a finite map from the names of files and directories to their contents, C. The contents of a directory is a new partial map from filenames to their contents. The contents of a binary file is simply the raw bits that it stores.

$$\begin{array}{l}\mathsf{M} ::= \mathsf{F} \rightharpoonup \mathsf{C}\\ \mathsf{C} ::= \mathsf{Bits} + \mathsf{M}\end{array}$$

Next, we define several useful predicates on repositories. As the model of our repository is richer, the predicates become a bit more complex. It is tempting to reuse the previous definition of $f \mapsto c$, that we have seen in Section 5:

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

This definition is suitable for top-level files, but cannot be used to refer to files stored in nested directories. To do so, we introduce the notion of *path* as follows:

$$\mathsf{Path} ::= \mathsf{F}^* \times \mathsf{F}$$

A path consists of a possibly empty list of directory names and a file name. In accordance with most operating systems, we will write paths as a file name, such as f above, or $d\,/\,p$, where d is the outermost directory and p is the remainder of the path. When we wish to extend a path with a new file or directory, we will write (p,f) or (p,d).

While the above definition handles the case that the list of directories is empty, we still need to define the non-empty case. We complete the definition of the predicate with the following clause:

$$\begin{array}{ll}M \vDash (d\,/\,p) \mapsto c \quad \text{iff} & M(d) = M' \wedge dom(M) = \{d\}\\ & \wedge\, M' \vDash p \mapsto c\end{array}$$

Besides the $p \mapsto c$ predicate, we also need a predicate that asserts the directory referred to by a list of filenames is empty. Once again, we define this predicate recursively over the list of directory names.

$$\begin{array}{lll} M \models \mathit{isEmpty}\ (\varepsilon) & \text{iff} & \mathit{dom}(M) = \emptyset \\ M \models \mathit{isEmpty}\ (d\,/\,p) & \text{iff} & M(d) \models \mathit{isEmpty}(p) \\ & & \wedge\ \mathit{dom}(M) = \{\, d \,\} \end{array}$$

We leave the assumption implicit that $M(d)$ maps to a directory, rather than a file.

Operations

Now we can define operations to manipulate such repositories:

$$\begin{array}{rcl} \{\, \mathit{isEmpty}(p) \,\} & \mathsf{addDir}\ p\ d & \{\, \mathit{isEmpty}(p,d) \,\} \\ \{\, \mathit{isEmpty}(p) \,\} & \mathsf{addFile}\ p\,f & \{\, (p,f) \mapsto \varepsilon \,\} \\ \{\, (p,f) \mapsto c \,\} & \mathsf{modifyFile}\ p\,f\ c\ c' & \{\, (p,f) \mapsto c' \,\} \\ \{\, (p,f) \mapsto \varepsilon \,\} & \mathsf{deleteFile}\ p\,f & \{\, \mathit{isEmpty}(p) \,\} \\ \{\, \mathit{isEmpty}(p,d) \,\} & \mathsf{deleteDir}\ p\ d & \{\, \mathit{isEmpty}(p) \,\} \end{array}$$

The first two operations, addDir and addFile, create files and directories respectively. The precondition of both these operations does require the enclosing directory to exist, even if it is empty. The operation modifyFile updates the contents of a file in the repository. Finally, the deleteFile and deleteDir operations remove an empty file and empty directory from the repository respectively.

In the above, we have developed a richer internal model for repositories than the usual 'flat' heaps described using separation logic. In particular, the model of our repository may contain nested heaps, used to model directories. To complete the definition of our version control system, we define the *mod* and *addr* functions as in the obvious fashion. Note that both these function compute a set of *paths*:

$$\begin{array}{ll} \mathit{mod}\ (\mathsf{addDir}\ p\ d) & = \{\, (p,d) \,\} \\ \mathit{mod}\ (\mathsf{addFile}\ p\,f) & = \{\, (p,f) \,\} \\ \mathit{mod}\ (\mathsf{modifyFile}\ p\,f\ c\ c') & = \{\, (p,f) \,\} \\ \mathit{mod}\ (\mathsf{deleteFile}\ p\,f) & = \{\, (p,f) \,\} \\ \mathit{mod}\ (\mathsf{deleteDir}\ d) & = \{\, (p,d) \,\} \\ \mathit{addr}\ (\mathit{isEmpty}(d)) & = \emptyset \\ \mathit{addr}\ (p \mapsto c) & = \{ p \} \\ \mathit{addr}\ (P * Q) & = \mathit{addr}\ P \cup \mathit{addr}\ Q \end{array}$$

The nested directory structure introduces very little complexity. Why not? We have judiciously chosen 'address space' to reflect the structure of our heaps: both the internal model of a repository and paths are defined recursively. Furthermore, the primitive operations we have defined have as tight a footprint as possible. If we had tried to define an operation that deletes non-empty directories, computing the set of modified variables would be much more complex. In practice, a version control system may choose to expose such a command to its users, even if it is implemented using the primitive operations we have described here.

Although we have not given a complete specification of a realistic version control system, we have seen many crucial ingredients: binary files, text files, and directories. There are several features we have not yet covered, such as symbolic links or file permissions. For the sake of exposition, we have not tried to define a single system combining all these features in this paper. Doing so will involve serious technical overhead, but seems to be within grasp.

8. Branching and merging

The operations we have seen so far capture the changes a user makes to the raw data that is stored in the repository. These changes are represented as patches; the history of the repository consists of the sequence of patches that lead to the current state. These are not the only kind of commands that a version control system must process.

Version control systems also support commands to manipulate a repository's history, such as pulling patches from or pushing patches to another repository. Things become more interesting, however, when we admit operations that introduce branching structure to the repository's history.

Many version control systems support some notion of *branches* within a repository. Each branch is a repository in its own right. Changes made in one branch of the repository are typically kept separate from all other branches. Many modern version control systems allow new branches to be created cheaply, without copying files on disk. Users can easily switch between different branches, making it possible to maintain different versions of the repository at the same time.

There are many practical scenarios where branching can be useful [20]. Consider a developer working on a new feature, when she is notified of a critical bug she needs to fix right away. Without branching, she has a few options:

- She could check out a fresh copy of the repository in which to fix the bug. For large repositories with long histories, this can be an expensive operation. In particular when the build and deployment process is complex, duplicating the entire repository may be undesirable;
- She could discard the work on the new feature she has done so far. Although rolling back to a previous state of the repository is usually an inexpensive operation, discarding code is clearly undesirable.
- She could develop the bugfix in her repository and push both the fix and the unfinished new feature. Once again, this is a bad choice as it publishes untested or incomplete code in the main repository.

By introducing a new branch for the bugfix, developers can work on different projects within the same repository, without having to store more than one copy of the repository on disk.

The question remains how we can model branches in framework for version control systems we have seen so far. We will illustrate the general principle using an example and notation drawn from the Git literature [15].Typically, branching is illustrated using acyclic directed graphs to model the changes made to a repository over time. The nodes of the graph correspond to the states of the repository; the edges correspond to the patches. For example, a repository's history consisting of the patches $c_1; c_2;$ could be described graphically as follows:

$$\begin{tikzcd}
s_1 \arrow[r, "c_1"] & s_2 \arrow[r, "c_2"] & s_3
\end{tikzcd}$$

Version control systems sometimes allow users to label certain states. For example, in Git the main branch is usually labeled the *master*. We will draw such labels using rectangles.

$$\begin{tikzcd}
 & & \boxed{master} \arrow[d] \\
s_1 \arrow[r, "c_1"] & s_2 \arrow[r, "c_2"] & s_3
\end{tikzcd}$$

Creating a new branch does not change the state of the current repository. For example, to creating a new branch called *issue53* to resolve a bug report yields the following situation:

$$\begin{tikzcd}
 & & \boxed{master} \arrow[d] \\
s_1 \arrow[r, "c_1"] & s_2 \arrow[r, "c_2"] & s_3 \\
 & & \boxed{issue53} \arrow[u]
\end{tikzcd}$$

Things become more interesting once these two branches diverge. Suppose a developer commits a bugfix, c_3, to the *issue53* branch. We can model this graphically as follows:

$$\begin{tikzcd}
 & & \boxed{master} \arrow[d] & \\
s_1 \arrow[r, "c_1"] & s_2 \arrow[r, "c_2"] & s_3 \arrow[r, "c_3"] & s_4 \\
 & & & \boxed{issue53} \arrow[u]
\end{tikzcd}$$

How do these branches affect the 'program' stored in the repository's history? Modeling choice involves introducing conditional statements:

$c_1; c_2;$ **if** *issue53* **then** c_3 **else** *skip*;

If development on the master branch continues, the development branches fork further:

$$\begin{tikzcd}
 & & & s_5 & \boxed{master} \arrow[l] \\
s_1 \arrow[r, "c_1"] & s_2 \arrow[r, "c_2"] & s_3 \arrow[ur, "c_4"] \arrow[dr, "c_3"'] & & \\
 & & & s_4 & \boxed{issue53} \arrow[l]
\end{tikzcd}$$

Unsurprisingly, this corresponds to the program:

$c_1; c_2;$ **if** *issue53* **then** c_3 **else** c_4;

Things become more interesting once branches *merge*. When merging the *issue53* and *master* branches, the version control system must compute patches m_1 and m_2 that can be applied to both s_5 and s_6 to yield some new common state s_6.

$$\begin{tikzcd}
 & & & s_5 \arrow[dr, "m_2"] & & \boxed{master} \arrow[dl] \\
s_1 \arrow[r, "c_1"] & s_2 \arrow[r, "c_2"] & s_3 \arrow[ur, "c_4"] \arrow[dr, "c_3"'] & & s_6 & \\
 & & & s_4 \arrow[ur, "m_1"'] & & \boxed{issue53} \arrow[ul]
\end{tikzcd}$$

The resulting history of the repository then becomes:

$c_1; c_2;$ **if** *issue53* **then** $\{c_3; m_1\}$ **else** $\{c_4; m_2\}$;

Note that this does not specify *how* to compute m_1 and m_2. Typically, this is done using a three-way merge algorithm between the states s_3, s_4, and s_5. Computing the patches m_1 and m_2 may not always be possible. If both branches modify the same lines of the same file in a different fashion, for example, most version control systems have no way of guessing which modification to prefer. This situation is sometimes referred to as a *merge conflict*.

Predicting when merge conflicts occur is not easy. This depends heavily on the exact merge algorithm used. Using the framework we have developed so far, however, we *can* give a sufficient condition for the absence of merge conflicts. Consider the definition of *independent* patches from Section 5. A crucial property of two independent patches is that they *commute*. Hence if the patches c_1 and c_2 in two separate branches are independent, there is an obvious choice for the merging patches: c_2 and c_1 respectively.

Typically when the version control cannot find merging patches, i.e., when the patches in both branches are not independent, the user is required to resolve the conflict manually. To do so, the user applies patches to either (or both) branches until the changes can be reconciled.

The independence condition is not *necessary* for merging patches to exist. For example, suppose c_1 changes the contents of a binary file f and then later reverts this change to f. According to the above characterization of independence, c_1 is not independent of a patch c_2, which also modifies the file f. Yet it may still be possible to find merging patches. In this example, removing the patches from c_1 that modify f produces a new patch with a smaller footprint that will compose with c_2.

The problem lies in our definition of *mod* function: it computes the set of modified variables regardless of which addresses actually change. For example, a command such as replace $f\ c\ d$ from Section 5 is said to modify f, even if c and d are equal. A more extensional view of modification would be to compute those addresses that have changed in the repository after applying a series of patches.

There are advantages and disadvantages of such a more extensional viewpoint. On the one hand, it becomes easier to commute patches as the set of modified addresses may be more precise. The drawback, however, is that the set of modified variables is harder to compute and may be *dependent* on the context: setting the contents of a file f to c may or may not modify f, depending on the current contents of f and its complete content history. This makes it harder to reason about patches independent of the repository to which they are applied. Both choices are sound, but further practical experience is necessary to decide which is preferable.

In this section, we have limited the discussion to simple branching and merging. In principle, we can also describe more complex merging operations, such as the behaviour of the Git rebase operation. We have restricted our example to the single operation of introducing a binary branching operator, corresponding to the familiar if-then-else construct. Many git repositories use a variety of branches to track different versions of the same repository. We postulate that simple branching control flow should be enough to model such more complex branching behaviour also.

9. Discussion

Why bother writing semantics for version control systems? After all, there are plenty of widely used *version control systems* without a solid mathematical foundation. There are also plenty of widely used *programming languages* without a clearly defined semantics, yet this does not stop us from continuing to research programming language semantics.

There are several ways we envision a clearly defined semantics for version control systems, based on logic for mutable state, can make a difference. First and foremost, such semantics can help *designers* of version control systems make difficult design decisions. Mailing lists and internet fora are full of questions about version control systems asking whether some unexpected behaviour is a bug. Such threads can spark all kinds of debate, even amongst the system's developers, about what the correct behaviour is. A good example is the recent work by Lourenço et al. [21] that models (a fragment of) Git in the Alloy modeling language [22]. By doing so, it becomes possible to check certain sanity properties of the specific git commands for manipulating repositories. Their work uncovered corner cases where developers could not initially agree if it was a bug or not. Such discussions are much harder to conduct without a clear specification. These results, however, are only applicable to git. The framework we present here is much more general.

Not only the developers of version control systems could profit from a clearly defined semantics. The users of version control systems also have much to gain. A clear semantics gives more understanding of *how* a version control system is expected to operate. This makes it easier to predict what the outcome of a complex operation, such as merging two branches, will be *without* having to execute this command. When collaborating across different developers, it is common to distribute programming tasks to minimize the chance of having conflicts, e.g., if one programmer only modifies file X and the other modifies file Y, they should be able to merge their work later. With a precise mathematical semantics it becomes possible to reason more abstractly about this process: what *invariants* do we need to incorporate in our development process to *guarantee* the absence of merge conflicts? Answering this kind of question is only possible with an unambiguous semantics.

Related work

There are several other proposals for the semantics of version control systems. The early work on the semantics of SCCS [6] and CVS [7] is obviously closely related, as are other early models of version control [23, 24]. More recently, there have been several proposals for a semantics for version control, for example, using acyclic directed graphs [25], calculus [26], category theory [27], or homotopy type theory [28]. No existing work makes the connection between the semantics of imperative programs and version control systems as we do in this paper.

The darcs version control system is one of the few systems that strives to have a formally defined semantics. The theory of patches [29–32] tries to give an algebraic characterization of version control. The darcs model differs in several crucial aspects to the semantics we have presented here.

Firstly, darcs maintains an (unordered) set of patches, stored in no particular order; we have chosen for an ordered sequence of patches. Working with sets increases the algorithmic complexity of some of the algorithms that darcs uses to add or remove patches, as these operations are allowed to make fewer assumptions about the order in which patches are applied.

Furthermore, darcs has a special mechanism for commuting patches. Where we have identified conditions under which two patches can commute, darcs allows all patches to commute in principle. When darcs commutes two patches $c_1;c_2$, it computes two new patches $d_2;d_1$, where d_1 and d_2 'do the same as' c_1 and c_2 respectively. Associating a precise meaning to this specification is one of the hardest problems in darcs's theory of patches.

Finally, all patches are invertible by design in darcs. We have chosen to enforce this in the version control systems presented in this paper, even if it is not strictly necessary.

The existing research on operational transformations and bidirectional transformations is a bit further afield. Operational transformations [33] focus on simultaneous edits on a single text document. Revision control, on the other hand, never needs to deal with simultaneous changes to the same repository. Bidirectional transformations [34] focus on the view-update problem: how can we reconcile changes on a *view* on a piece of data with the original data. Bidirectional transformations are not concerned with *versioning* or being able to roll back to previous states. Furthermore, bidirectional transformations attempt to consolidate changes between *different* data structures. The work we present here is concerned with consolidating different changes on a *single* data structure, namely the internal model of the repository.

Future work

The semantics presented in this paper do not model a realistic version control system, but focus on the many pieces of the puzzle. Clearly, it would be worthwhile to write a realistic model of an actual version control system in this style. While there are no obvious technical obstacles to such a model, it would provide further evidence of the applicability of these ideas.

In this paper we have shown how to model both binary files and text files. In principle, however, the same ideas can be used to model structured data. A good example would be files containing comma-separated values. Although these may be represented as monolithic binary files or lines of text, there is two dimensional structure worth exploiting. If you do represent such files as lines, adding a new column will conflict with every other change to the same file—even if there is no modification to the same cell by these two patches. Applying techniques from data type generic programming [35, 36], we hope to develop a general theory of version control of structured data.

The focus of this paper has been on the design of internal models and the operations that manipulate them. Although we have briefly touched upon branching and merging (Section 8), there are many other operations on the repositories history worth supporting. Users may want to reorder patches, add new patches at specific locations in the linear history, or delete patches besides the most recent one. All of these are easy to support in principle, but finding an expressive, minimal set of commands may be difficult in practice.

Closure

In his textbook on the semantics of programming languages [37], John Reynolds wrote:

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

With the research presented in this paper, we hope that the same will become true not only of programming, but of the broader software development process.

Acknowledgments

We would like to thank Stevan Andjelkovic, Pierre-Evariste Dagand, Stephan Eggermont, Sebastian Fischer, Eric Kow, Ian Lynagh, Ganesh Sittampalam, Owen Stephens, our anonymous reviewers, and the members of IFIP Working Group 2.1 and the Software Technology Reading Club in Utrecht for their valuable feedback.

References

[1] O'Sullivan, B.: Making sense of revision-control systems. Communications of the ACM **52**(9) (2009) 56–62

[2] Floyd, R.W.: Assigning meaning to programs. In: Proceedings of the Symposium on Applied Maths. Volume 19., AMS (1967) 19–32

[3] Hoare, C.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1967) 576–583

[4] Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, IEEE (2002) 55–74

[5] O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Computer Science Logic, Springer (2001) 1–19

[6] Marc, J.: The source code control system. IEEE Transactions on Software Engineering. SE-1 (4) (1975) 364–470

[7] Grune, D.: Concurrent versions system, a method for independent cooperation. Unpublished note (1986)

[8] Cederqvist, P.: Version Management with CVS. Network Theory Ltd. (2002)

[9] Vesperman, J.: Essential CVS. O'Reilly Media (2003)

[10] Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly (2004)

[11] Mason, M.: Pragmatic Version Control: Using Subversion. Pragmatic Bookshelf (2006)

[12] Rooney, G.: Practical Subversion. Apress (2006)

[13] Roundy, D., et al.: Darcs `http://www.darcs.net`.

[14] O'Sullivan, B.: Mercurial: The Definitive Guide. O'Reilly Media (2009)

[15] Chacon, S.: Pro Git. Apress (2009)

[16] Swierstra, W.: A Functional Specification of Effects. PhD thesis, University of Nottingham (November 2008)

[17] Swierstra, W., Altenkirch, T.: Beauty in the beast. In: Proceedings of the ACM SIGPLAN workshop on Haskell workshop. Haskell '07 (2007) 25–36

[18] The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004) Version 8.0.

[19] Martin-Löf, P.: Intuitionistic type theory. Volume 17. Bibliopolis Naples, Italy (1984) Notes by Giovanni Sambin.

[20] Appleton, B., Berczuk, S., Cabrera, R., Orenstein, R.: Streamed lines: Branching patterns for parallel software development. In: Pattern Languages of Programs. (1998)

[21] Cláudio Lourenço, Neves, R., Cunha, A., Kang, E., José Barros: The truth about git: modeling git with alloy (2013)

[22] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) **11**(2) (2002) 256–290

[23] Lippe, E.: Camera: Support for Distributed Cooperative Work. PhD thesis, Universiteit Utrecht (1992)

[24] Van Den Hamer, P., Lepoeter, K.: Managing design data: The five dimensions of CAD frameworks, configuration management, and product data management. Proceedings of the IEEE **84**(1) (1996) 42–56

[25] Virtanen, T.: Git for computer scientists. `http://eagain.net/articles/git-for-computer-scientists/`

[26] Pool, M.: Integrals and derivatives `http://sourcefrog.net/weblog/software/vc/derivatives.html`.

[27] Mimram, S., Di Giusto, C.: A categorical theory of patches. In: Mathematical Foundations of Programming Semantics. (2013)

[28] Angiuli, C., Morehouse, E., Licata, D.R., Harper, R.: Homotopical patch theory. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. ICFP '14 (2014) 243–256

[29] Jacobson, J.: A formalization of darcs patch theory using inverse semigroups. Technical report, Technical Report CAM report 09-83, UCLA (2009)

[30] Dagit, J.: Darcs patch theory. The Monad.Reader **9** (November 2007)

[31] Dagit, J.: Type-correct changes — a safe approach to version control implementation. Master's thesis, Oregon State University (2009)

[32] Roundy, D.: Darcs: distributed version management in Haskell. In: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, ACM (2005) 1–4

[33] Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data. SIGMOD (1989) 399–407

[34] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. POPL (2005) 233–246

[35] Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming: an introduction. In: Advanced Functional Programming. Springer (1999) 28–115

[36] Löh, A.: Exploring Generic Haskell. PhD thesis, Universiteit Utrecht (2004)

[37] Reynolds, J.C.: Theories of programming languages. Cambridge University Press (2009)