Track Changes:
Identity in Version Control

Mateusz Zukowski

A thesis submitted in conformity with the requirements
for the degree of Masters of Information

Faculty of Information

University of Toronto

Copyright by Mateusz Zukowski 2014
TRACK CHANGES: IDENTITY IN VERSION CONTROL

Mateusz Zukowski

Masters of Information

Faculty of Information
University of Toronto

2014

ABSTRACT

The growing sophistication of version control systems, a class of tools employed in tracking and managing changes to documents, has had a transformative impact on the practice of programming. In recent years great strides have been made to improve these systems, but certain stubborn difficulties remain. For example, merging of concurrently introduced changes continues to be a labour-intensive and error-prone process. This thesis examines these difficulties by way of a critique of the conceptual framework underlying modern version control systems, arguing that many of their shortcomings are related to certain long-standing, open problems around identity. The research presented here casts light on how the challenges faced by users and designers of version control systems can be understood in those terms, ultimately arguing that future progress may benefit from a better understanding of the role of identity, representation, reference, and meaning in these systems and in computing in general.
ACKNOWLEDGEMENTS

Thanks to my advisors, Brian Cantwell Smith, Jim Slotta, and Steven Hockema, for their guidance and patience, and to my friends in the Group of n—especially to Jun Luo—without whose encouragement and support this thesis would not exist.
# TABLE OF CONTENTS

1. Introduction........................................................................................................................................... 1
   1.1 Digitally Mediated Collaboration................................................................................................. 3
   1.2 A Ship of Theseus Sets Sail........................................................................................................... 5
   1.3 Related Work ............................................................................................................................... 8
2. Critique.................................................................................................................................................. 10
   2.1 Content........................................................................................................................................... 10
   2.2 History .......................................................................................................................................... 13
   2.3 Projects and Repositories ............................................................................................................ 14
   2.4 Change .......................................................................................................................................... 16
   2.5 Source Code................................................................................................................................ 20
      2.5.1 Aesthetics ............................................................................................................................... 21
      2.5.2 Meaning ................................................................................................................................. 22
   2.6 Persistence ..................................................................................................................................... 24
   2.7 Deferred Registration .................................................................................................................... 30
   2.8 Merging.......................................................................................................................................... 34
      2.8.1 Concurrency ............................................................................................................................ 35
      2.8.2 Conflict .................................................................................................................................... 38
      2.8.3 Reconciliation .......................................................................................................................... 40
3. Conclusion .............................................................................................................................................. 47
   3.1 The Way Forward .......................................................................................................................... 48
4. Bibliography .......................................................................................................................................... 50
FIGURES

Figure 1 – Two representations of the same picture ................................................................. 11
Figure 2 – Line-level conflict .................................................................................................. 18
Figure 3 – Variable declaration and assignment in Java ............................................................. 24
Figure 4 – Function declaration and assignment in Java ............................................................. 24
Figure 5 – Changes as snapshots .............................................................................................. 25
Figure 6 – Changes as deltas ..................................................................................................... 26
Figure 7 – Roger Fenton’s The Valley of the Shadow of Death ..................................................... 31
Figure 8 – Branching and merging ............................................................................................ 37
Figure 9 – Branching (cloning) and merging (pulling) between two remote copies of the same repository .............................................................................................................. 37
Figure 10 – Merge resulting in a conflict .................................................................................... 39
Figure 11 – Conflict in a 2-way merge ...................................................................................... 40
Figure 12 – Conflict in a 3-way merge ...................................................................................... 41
Figure 13 – Criss-cross merge .................................................................................................. 42
Figure 14 – Pathological criss-cross merge .............................................................................. 42
Figure 15 – Semantic conflict ................................................................................................... 45

SIDEBARS

Three Generations of Version Control Systems ......................................................................... 15
History of Concurrency in Version Control ............................................................................. 35
1. INTRODUCTION

Keeping track of changes has become an essential part of the practice of software development. The modern complexities of the craft are such that programmers must manage source code not only spatially, in terms of files and directory structures, but also temporally, over complex, diverging and converging timelines resulting from multiple strands of work done in parallel. A class of tools called version control systems (VCSs) has been developed to help address these temporal concerns, allowing programmers to more easily navigate through and communicate about the changes they make to source code. The growing sophistication of these tools has had a transformative impact on the practice of programming (see for example Rigby et al. 2013; Spinellis 2012), and has in turn prompted an explosion in the number and diversity of collaboratively developed software projects (Doll 2013).

Yet despite considerable progress, version control systems continue to bump up against certain stubborn difficulties. For users of these systems, manually reconciling conflicting changes—a frequent occurrence when multiple programmers work on the same piece of code—remains a frustrating and error-prone process. And for the systems’ designers, the choice of appropriate algorithms and representations for keeping track of versions—for example, the choice of whether to record history as a series of transformation or as snapshots—remains a challenge and a point of dispute.

Meanwhile, newly opened avenues for collaboration in software development are putting pressure on previously stable concepts and practices. By leveraging features of newer version control systems, web-based services like GitHub, which host the source code, wikis, and issue trackers of a growing number of open source projects*, have made it trivial for anyone to create clones of those projects. These copies, known as “forks”, allow a user to split off and continue development of the project independently of the original. Some of the changes introduced in these clones eventually find their way back into the project they split from, and some diverge off in their own direction. This can lead to tangled histories and a proliferation of subtly different variants,

*That is, projects whose source code is licensed and published by the copyright holder with provisions that allow for open modification and/or repurposing. See Weber (2004) for an overview of open source software practices.
blurring the projects’ identities and complicating contingent notions of authenticity and credibility”.

The research presented here is an attempt to bring to the surface some of the issues faced by users and designers of modern version control systems. The core of the project is an analysis of several popular VCSs (namely Git, Subversion, and Mercurial), a comparison of their differing approaches to dealing with change, and ultimately an explication of their limitations. The argument is that many of those shortcomings can be traced down to a number of long-standing conceptual problems—issues of identity, representation, reference, and meaning—endemic not just in version control but in computing and information practice in general.

Though much will be said about the state of the art in version control systems, the implications of this project go beyond the concerns of software developers. As tools and practices first pioneered by programmers find their way into the wild—for example in large-scale collaborative authorship efforts like Wikipedia, or in increasingly popular cloud storage services like Dropbox—the issues raised here are of relevance anywhere digital documents are the product and the central mediating point of collaboration.

Finally, there is another, deeper theme underlying this work. Despite its apparent premise, this thesis is as much about version control as a film set in Rome is about a city in Italy. Version control is merely a convenient setting for a story about something more fundamental—about the difficulties inherent in reasoning formally about the informal. Though the designers of the systems discussed here never intentionally set out to tackle deep problems in ontology and metaphysics, this is exactly where they ended up. In a modest effort to build better tools for managing source code, they have unwittingly found themselves tackling some of philosophy’s most confoundingly slippery concepts. Underpinning much of the research presented here is a sensitivity to that wider body of literature, and the various perspectives it offers on the ontological commitments (having to do with reference, meaning, and change) implicitly embedded in version control systems.

Before turning to the inner workings of these systems, the next two sections present a motivation and a grounding within a wider, non-technical context. In section 1.1, I argue that the issues that version control systems have been designed to address are problematic not just in programming, but in document-mediated collaboration in general, and moreover, that analysis of these systems could prove helpful in a wider range of present and future applications. In section 1.2, I point out that some of those issues are manifestations of certain well-known conceptual problems—specifically, that the problem of tracking a document or project through its many

---

1 For instance, begging the question of whether a fork should be treated as a new project or a continuation of the original. Which one gets to keep the original name, and how should the other be referred to? Or, how are issues of governance to be resolved when all forks are given equal standing within the rules and affordances of the host system, as is largely the case on GitHub?
versions, as a single, persevering entity, is analogous to a well-known class of problems called “sorites paradoxes”, rooted in tension between formalism and vagueness.

1.1 DIGITALLY MEDIATED COLLABORATION

Sooner or later this document—this draft of a Masters thesis—will be sent off for proofreading. The proofreader will likely insert some comments, edit a few sentences, and maybe even move or remove entire sections.

But what if, while waiting for those edits, I—the original author—want to do some additional writing? What if I add to a section that the editor has just removed? What if the editor comments on a sentence that I concurrently revise? Or what if I want to maintain different versions of this thesis—say one targeted at computer scientists and another at a lay audience. How do I ensure that changes I make to one find their way into the other—but only such that those changes are appropriately matched to the other’s format?

These problems are not new; they are probably as old as writing itself. But it is only recently that computers have made copying and editing trivial, and that the internet has made real-time collaboration and co-authorship commonplace. Anyone who has collaborated on an article via email, has edited a Wikipedia article, or has been frustrated by an out-of-sync inbox, will attest that these problems seem to be popping up more and more often.

For the average user, confrontations with versioning and synchronization are increasingly commonplace, but for software developers these problems have been an unavoidable part of day-to-day work for decades. And while other disciplines have rich histories of dealing with these sorts of issues—lawyers, for example, also have complicated requirements for keeping track of versions—programmers have had the unique luxury of devising, implementing, and tinkering with their own automated solutions which they are able to immediately test and improve on. This capacity for continuous modification and improvement of one’s own tools of the trade has allowed for a uniquely short development cycle. The result is that software developers now have in hand some of the most sophisticated computer-based tools and practices for collaborating on multi-authored documents and projects.

A modern source code version control system typically offers the ability to associate individual changes with specific authors, to review those changes, and to automatically propagate them among team members. The capacity for multiple developers to work concurrently on the same files is standard (usually by allowing each collaborator to obtain a separate copy of the codebase to work on), as are facilities for automatically combining (“merging”) changes, and for manually resolving conflicts when changes cannot be reconciled automatically. Tools for visualizing the many
diverging and converging lines of work within a project have become standard, along with tools for comparing differences between versions. Some newer systems also provide the ability to operate on the history itself, allowing a developer to retroactively modify and curate the record of changes to make it easier for others to read and interpret.

Many of the tools and practices originating in version control are now trickling out into wider use outside of software development. For example, the document history ubiquitous in wiki systems like Wikipedia, with its ability to display differences between versions, is modeled on features of early source control systems. Microsoft Word’s recently introduced “compare and merge documents” function is another feature similar to those long available to programmers. And the methods used for resolving conflicts in cloud storage systems like Dropbox are primitive compared to some of the conflict resolution tools employed in software development.

The relative sophistication of programmers’ version control tools has made possible certain practices not yet commonly employed elsewhere. Programmers frequently operate on files in ways that people unaccustomed to these tools and practices would deem risky or inscrutable, for example concurrently working on the same file, or unrepentantly creating copies and variations of entire projects without fear of making reconciliation unmanageable down the line. One standard method of developing software using a VCS involves simultaneously maintaining no less than three different versions of a project: one for each new feature under development so that each feature can be worked on independently, another into which those features are integrated once they are finished, and another that tracks the state of the previously released version, to allow for the release of emergency bug fixes without having to include untested new features (Driessen 2010, also see section 0). Although this sort of workflow might be possible without a VCS, the ability to easily

---

A savvy reader might point out that much of what has been said here about version control systems could also be said about modern databases. This is especially true of database systems designed with concurrency in mind, which often implement conflict prevention and resolution techniques whose sophistication rivals that of version control systems. Compare for example the “vector clocks” implemented in the Riak database (Basho 2013; Lamport 1978), with the recursive 3-way merge algorithm in Git (discussed in section 2.8.3). While the various ways in which database systems deal with concurrency are certainly of interest, databases are typically situated at least one infrastructural layer deeper than the level being discussed here—that is, databases are not typically exposed to the end-user as tools in themselves. In fact, VCSs are sometimes built on top of database systems, as with Subversion, which uses BerkleyDB as one of its storage backends. To put it another way, version control systems are a better candidate for the present discussion because the problems discussed here are what version control systems are about, in the sense that version control systems are tools specifically designed for operating within that problem space. Databases, on the other hand, do their best to hide or abstract from the user the issues that arise from concurrency. To be fair, however, this distinction is becoming increasingly blurry, as the algorithms and representations employed in next-generation databases are running up against the same sorts of intractable ambiguities as version control systems.
transfer individual and wholesale changes between each of those versions is what makes the process feasible.

Certainly then if one wants to see where digitally-mediated collaboration is going, the programmer’s toolbox is a good place to start. But, as advanced as these tools may be, they are also first to stumble up against some of the obstacles blocking the way forward. Before moving on to a detailed look at how version control systems work and the sorts of things they are and are not able to do, we turn to what may be the most fundamental obstacle in digital collaboration: the concept of identity.

1.2 A SHIP OF THESEUS SETS SAIL

There is a story—a very old story—about a ship commanded by one Theseus, mythical founder of Athens. The ship, a wooden galley with two sets of thirty oars and a crew of Athens’ best and brightest, sets sail in search of adventure. Criss-crossing the Mediterranean, the vessel encounters countless storms that strip off many of its wooden planks. At each port of call, the lost wood is replaced, until eventually not a single original plank remains. After decades at sea, the ship triumphantly arrives back in Athens, but Theseus wonders whether his vessel really is the same ship as the one that set out all those years ago—and if not, at which point along the journey did it become something else."

The story, a perennial favourite among philosophers, is essentially about identity—about why some things that undergo change might still be considered the same thing, while others become something different. Clearly, whatever it is that made Theseus’ ship that particular ship could not have been a simple matter of having been made of one particular piece of wood over another. The ship’s identity seems to persist in spite of its physical parts having been entirely replaced. But neither can that identity be a purely attributed property, since even if all records and memories of it had faded, intuition tells us that it would still have been the same ship, regardless of whether anyone knew it as such.

* The original version of the story, from Plutarch’s Life of Theseus, quoted in this thesis’ epigraph, had the ship returning to Athens with all of its original planks intact. It is only later that the ship begins to change, when, honoring their hero, the Athenians try to preserve the vessel for posterity by gradually replacing any wood that rots or falls away. Although the colorized version recounted above is somewhat different, most readers familiar with the story would likely recognize it as essentially the same. The point very much worth noting here is that something like a story—a series of abstract concepts and references to non-existent mythical beings—is susceptible to the same paradox as a physical ship. That is, a story that retains nearly none of the words it started out with continues to be recognized as effectively the “same” story. This is worth keeping in mind, as documents—the subject of this thesis—are arguably more like stories than they are like wooden ships.
Perhaps what makes the story so paradoxical is that the concept of identity—as Lewis (1986; Noonan 2009) put it—is at once “utterly unproblematic” and yet confoundingly slippery. In our everyday experience things seem to just have identity, and any confusion about that identity is generally seen as epistemic. When you confuse someone else’s car for your own, it’s not that the car is momentarily yours and then—once you realize your mistake—not yours. The car’s identity as “not yours” was always there, you just weren’t aware of it; much as if because of dim lighting you had mistaken the car’s colour for brown, when in fact it was red all along. On the other hand, despite identity’s apparent obviousness in day-to-day experience, one would be hard pressed to explain the concept without quickly venturing into some decidedly rarefied philosophical territory. Formal examination of the topic tends to quickly ascend to discussion of the indescribability of identicals, intrinsic and extrinsic properties, modal realism, perdurantism versus endurantism, and other lofty concepts (see for example Lewis 1986 and McKinnon 2002).

The puzzle of Theseus’ ship—seemingly obvious yet enduringly enigmatic—has seen some notable additions and elaborations along the way. Hobbes, for example, wondered what it would mean if all of the planks that fell off of the original ship had been subsequently gathered and used to build a new vessel; would there now be two Ships of Theseus? and if not, which of the two did Theseus command? (Chisholm 1976) More recently, the puzzle has featured in questions about consciousness and personal identity, in a hypothetical situation where one’s brain is incrementally replaced by cybernetic parts (Chalmers 1997; Pylyshyn 1980), or where a brain is partially transplanted into another body (Shoemaker 1999).

But the problem has, for the most part, remained a mere curiosity—a contentious thought experiment for philosophers, and an intriguing premise for authors of speculative fiction. The fact that our notions of identity seem so difficult to pin down has had little practical impact on our day-to-day lives. Yet this may be changing. Or at least that is the argument I aim to make here.

Computers have become so ubiquitous and so ordinary that it might seem strange to paint them as metaphysically suspect. Yet computing is deeply at odds with the world it inhabits. The discipline sits at the intersection of the ethereal formalism of pure mathematics, and the messy embodied reality of everyday life. We program our computers in terms of the former—demanding precise, unambiguous definitions—yet we expect them to function in the latter—a reality where vagueness and ambiguity are commonplace and entirely unproblematic. The result is a simmering conflict, one that manifests itself to the average user in the form of seemingly minor annoyances and incongruences; even ostensibly simple tasks, like copying a file or sending an email, can be
frustratingly complicated, especially if they deviate even slightly from prescribed parameters. And for programmers and designers of user interfaces, that conflict in many ways defines the discipline; finding ways to reconcile the gap between the rigid formalism of computing and the malleable messiness of everyday experience is what keeps many of us employed (even if not all of us know it).

Version control systems—an outwardly banal class of software tools—are in fact fertile territory for exploring this premise. VCSs track artifacts (documents, projects) whose makeup is constantly changing; a file in Git or Subversion could easily contain none of the text it started out with. Yet we expect these systems to somehow preserve a document’s identity, so that it can be traced and operated on back through all of its changes. In a way, version control systems make the Ship of Theseus real. We approach our projects and documents like real objects—like ships in the sea, with the same expectation of permanence in spite of change—yet we demand an exact definition of their identities, to appease the formal requirements of the computer.

Put plainly, the crux of the problem faced by version control systems is that a major aspect of what they do relies on their ability to track and preserve identity; yet the various notions of identity that these systems implement fall far short of what their users might naturally expect. This, I will argue, is due in part to the systems’ designers having only a nascent recognition of the problem, and in part to a sort of “infrastructural inertia” that limits what can be feasibly implemented within the constraints of current technologies and practices. But—and this, I hope, is where things get interesting—there may be more fundamental issues at play. For instance, to properly support identity, version control systems will likely need some way of referring to their content in ways that respect its mereological complexity. In sections 2.4 and 2.5, for example, I try to show how a simple reference to a piece of text is, in fact, anything but simple; the same paragraph might be referred to in terms of its lines, columns, words, and sentences, or its fonts, and pixels, or the ideas it conveys, and the feelings it evokes. Likewise, in section 2.8 I point out how version control systems’ ability to detect whether two versions of the “same” piece of content have undergone conflicting changes is limited by a fundamental inability to establish identity in terms of meaning; that is, the judgement of whether two pieces of content are the same can, in at least some important contexts, depend on whether they “mean” the same thing—something far out of reach of any current VCS, and perhaps out of reach of anything within the current computing paradigm.

---

1 Take the “desktop” user interface for instance. The metaphor of an office workspace, with its files, folders, and clipboards, is a thin one at best. But it’s not just that the entities represented by the icons don’t behave like what they purport to be. They don’t behave like anything we are accustomed to in the external world. It takes years of practice and a solid understanding of the underlying layers of infrastructure—perhaps right down to an abstract grasp of the von Neumann architecture—before one is able to use a computer without some sense of awkwardness. And even then the frustration persists, only with a lesser degree of mystery.
Thus, the variety of ways in which different version control systems deal with identity are the subject of this research. The ontological commitments embedded in these tools exemplify the full gamut of possible explanations of Theseus’ paradox; from Subversion’s naïve authoritarian take (it’s Theseus ship because a higher, absolute authority deems it as such) to Git’s nuanced approach (it might be Theseus ship for now, but the rules can and should change in the future as our ability to make such assertions improves).

Ultimately no version control system offers an entirely satisfying answer. But each provides insight into the ins and outs of the various ways of tackling the problem of identity, with a sort of concreteness and pragmatism not ordinarily seen in purely academic discussion of the subject. Moreover, along the way we get glimpses of the true depth of the problem, and why the present state of computer science and related fields is not well positioned to solving it.

Certainly, framing the challenges faced by version control systems in these terms does not tell the whole story. It might not even be the right story. But at the very least, it is an unorthodox one. Software developers rarely think about the deeper conceptual issues that underlie their tools and practices, and philosophers rarely turn their abstract ideas into concrete, practical tools. Bridging that gap is the subject of the work to follow, with the hope that it might provide insight in both directions.

1.3 RELATED WORK

Little if anything has been written in terms of a high-level conceptual analysis or critique of version control systems. Consequently, the research and ideas presented here are a synthesis of a variety of sources, attempting to apply a conceptual perspective from philosophy and information science to technical discussion of version control. The overall approach—that is, one leaning on ideas from philosophy as a way of better understanding problems in computing—owes much to B.C. Smith’s (1996) On the Origin of Objects. Many of the misgivings about the nature of information and the limits of VCS’ implicit notion of ‘content’ that surface throughout this thesis are articulated in Nunberg’s (1996) essay Farewell to the Information Age. Likewise, a discussion of the problems inherent in using formal analysis of content (specifically, of computer source code) to ascertain its relationship with the external world (i.e. its meaning) can be found in Smith’s (1985) Limits of correctness in computers.

In regards to how version control systems are used as a medium for collaboration, and their impact on the practices of software developers, Weber’s (2004) The Success of Open Source is an excellent starting point for situating those practices, while Fogel’s (2005) Producing Open Source Software describes them in detail. Raymond’s (2009) Understanding Version-Control Systems,
although unfinished and unpublished, provides perhaps the most thorough and in-depth conceptual analysis of modern version control systems, how they are used in practice, and some of their inherent problems and limitations.

Much of the analysis of the design of version control systems presented here is based on informal discussions and arguments that took place in public mailing lists and newsgroups. Linus Torvalds’ posts on the git@vger.kernel.org mailing list, whose archive is available at http://marc.info/?l=git as well as several other mirrors on the internet, are a primary source for understanding the many design decisions and ontological commitments that went into Git’s development. For an alternative perspective, the monotone-devel@nongnu.org mailing list offers insight into the design of Monotone, a distributed version control system that predated Git and took a somewhat different approach to addressing the same problems. Several archives of the mailing list are available on the internet, including one at http://osdir.com/ml/version-control.monotone.devel. A great deal of technical information on Git is available in Chapter 9: Git Internals, of Chacon’s (2009a) Pro Git. Similar information on Mercurial can be found in Chapter 4: Behind the Scenes of O’Sullivan’s (2009) Mercurial: The Definitive Guide. For comparison with a second-generation, centralized system, Collin-Sussman et al.’s (2011) Version Control with Subversion is a definitive source.

Perhaps the most thorough analysis of the 3-way merge algorithm (also known as diff3) described in section 2.8.3 is available by way of Khanna, Kunal, and Pierce (2007). A number of other experimental merge algorithms are described on the now-defunct revctl.org wiki, which has been partially archived at https://github.com/tonyg/revctrl.org and http://web.archive.org/web/20110305101029/http://revctrl.org/.

Far too much has been written on the topic of identity within the philosophical perspective for this brief review to do any justice to the topic. However, the entries in the Stanford Encyclopedia of Philosophy on Identity (Noonan 2009) and Identity Over Time (Gallois 2011) are a good starting point. More thorough consideration of the subject can be found in Lewis (1986), Wiggins (2001), and Williamson (1994), each reviewing the many issues around identity, and offering a different theory to explain the concept.

Finally, some of the problems of identity within the context of information technology and the internet (specifically having to do with referencing documents, for example by way of URLs) have been addressed by a series of discussions within the “Semantic Web” community, for example in Renear and Dubin (2003), Renear and Wickett (2010), and Huitfeldt, Vitali, and Peroni (2012).
2. CRITIQUE

What follows is both an introduction to and a critique of the current state of the version control systems (VCSs) commonly employed in software development—Git, Subversion, Mercurial, and the like. While in part it is intended as a primer for the uninitiated, readers already acquainted with these systems should find plenty to think about or disagree with. Each section is organized around a core version control concept (‘Content’, ‘History’, ‘Projects and Repositories’, etc.), with an explanation giving way to a problematization, aiming to point out the many subtle or hidden inconsistencies and absurdities embedded within. Although most users of these systems have some intuitive awareness of the issues raised here, it is rare to see those issues articulated for their own sake. An attempt at such an articulation is the core of this chapter, underpinned by the argument that many of the problems in version control can be reduced to the paradox of identity.

In an effort to pre-emptively dodge potential criticism, I should note that while what is said here generally applies to version control systems used in software development, there is a broader, more diverse category of such tools that differ from those that this text describes. These alternative takes on version control are certainly pertinent to the present discussion, not the least because they offer alternative perspectives on aspects of version control that a programmer-centered approach might take for granted. However, in order to keep the scope manageable, these alternative are only referred to sparingly for comparison, without the full examination they deserve.

2.1 CONTENT

In version control, “content” is the stuff that a system keeps under its purview; it is the material whose versions are to be controlled. In the version control systems employed in software development, content is usually synonymous with a collection of files and folders, behaving much like an ordinary computer file system, but with the addition of a temporal dimension that, among other affordances, allows for “rolling back” to a previous state.

For historical and infrastructural reasons, these VCSs have a deeply engrained bias for text and relatively impoverished capabilities for other media (images, sounds, videos, compiled programs). For example, most VCSs allow users to examine and merge differences between versions of text documents, but not pictures or sounds. For this reason—and also because this thesis it itself a textual document—textuality plays a central role in the discussion to follow. But this preference for text-based representations has some implications for the bigger concepts addressed here—for example, for the relationship between representation and identity, as in the case that the same thing can be represented by both, textual and non-textual means.
Consider for example Figure 1, showing a picture of a red circle encoded as a GIF (a binary bitmap), and as an SVG (a kind of XML format for describing vector-based graphics). The fact that the same picture can be represented in two different ways complicates the distinction between what is and isn’t “textual” (SVG being a textual medium for representing pictures, and GIF being a non-textual, binary format). Moreover, the relationship between the two representations—the fact that they both represent the same thing—is something entirely inscrutable to a VCS. That is, from a user’s point of view, the SVG and the GIF might be considered to have the same content, but to a VCS, the two are entirely unrelated. (To put it another way, if you were shown the output of the GIF, and then shown the output of the SVG, you would likely say that you had been shown the *same* content twice—an assertion that a VCS which sees content only in terms of its representation would be unable to make.)

**Figure 1 – Two representations of the same picture**

<table>
<thead>
<tr>
<th>Picture</th>
<th>GIF representation of the picture</th>
<th>SVG representation of the picture</th>
</tr>
</thead>
</table>
| ![Red Circle](image) | <svg height="100" width="100">  
(circle cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="red" /)  
</svg> | |
| ![SVG Code](code) | 4749 4638 3961 5300 5300 b300 00ff 0000 ef00 00df 0000 cf00 00bf 0000 af00 009f 0000 8000 0060 0000 5000 0020 0000 1000 00ff ffff 0000 0000 0000 0000 002c 0000 0000 5300 5300 ... | |

These sorts of considerations—having to do with what a piece of content represents and how it represents it—are generally considered out of scope for version control. As Nunberg (1996) points out, this attitude pervades digital systems in general, with content viewed as a kind of “noble substance that is indifferent to the transformation of its vehicles” (p. 4). Indeed, version control tools tend to treat their content as raw material—sequences of bytes whose meaning is (and ought to be) of no immediate consequence.

Although, as argued in later sections, this disregard for the relationship between meaning, representation, and identity stems more from necessity than ignorance, it is nevertheless a constant source of problems. For example, a system that establishes how alike two pieces of content are based solely on the similarity of their textual representations (see the discussion of *Levenshtein distance* in section 0) would fail to recognize any similarity at all between the GIF and SVG shown

---

Since GIF is a binary file format (that is, a stream of bytes that do not correspond to any particular letters), in order to show it here in a text document, it must be re-represented as something else—in this case, a series of hexadecimal numbers, with each number representing a byte. What is shown here then—at risk of complicating but perhaps also serving to reinforce the point—is a partial *representation of a representation* of a picture of a red circle.
in Figure 1. This has serious implications when such measures of similarity are used to establish identity, as in version control systems that use textual similarity to establish continuity through time (more on this in section 2.6). Another consequence, described in section 2.8.3, is that automatically combining changes to content without taking into account what it means or represents—as version control systems often do when combining two different series of changes—can cause conflicts that the system is unable to detect, potentially leading to a sort of pernicious high-level corruption.

Issues of representation and textuality aside, the notion of content in version control is in itself potentially problematic. When using a VCS, programmers are forced to decide which aspects of their project to put under version control and which to leave out. That is, they are forced to make a binary decision about what is and isn’t “content”. However, as Hockema and Coppin (2010) point out, such a distinction between content and container is not always clear, despite a tendency in information science to presume it as such; the mere act of designating something as content (e.g. placing a file under a VCS’ purview) can have subtle but important transformative effects on it. Likewise, stripping it of its context (e.g. leaving some contextually-related part out of the VCS’ purview) can make it unintelligible.

For example, the build environment—the set of tools and their settings under which the contents of the repository are developed—is not typically retained in a VCS; i.e. it is not typically considered to be “content”. Yet this context is essential. Leaving it out is a bit like archiving floppy disks without keeping around the software able to interpret the data written on them. And indeed, some details of the build environment are increasingly included in VCS repositories. But much like the problem of storing floppy disks and the software required to read them, it is difficult to draw a clear line between what ought and ought not be included. Shouldn’t one also preserve the floppy drive capable for reading those disks? And then what about the rest of the computer system that goes with it? And won’t we also need a human operator with the requisite skills to operate it? And the society in which that operator lived, so that we can contextualize his or her interpretation of the contents of those disks?

The blurriness of content as a concept in version control is such that an understanding of what should and should not be included develops in practitioners as a kind of intuitive skill. In time, experienced users tend to develop strong opinions about which sorts of files should be placed under version control and which are best left out, and moreover, they develop a sense of what effect these decisions might have down the road (for example how the decision might affect developer’s usage of that file, or of the repository as a whole). But when pressed to justify their reasoning, the

\* For example, a recently developed VCS called Verasity (Avram 2011) integrates the storage of certain peripheral or “meta” content, such as the bug tracking system and wiki associated with the project, in the same repository as the project’s source code.
2.2 HISTORY

The ability to keep a historical record of changes is the core function of a version control system. In a VCS such a record typically consists of a series of “commits”, where a commit represents a point in time when a user explicitly instructed the system to note newly introduced changes. Each such commit is generally tagged with a timestamp, some information about the committer (their name, email address, etc.), and a short note that the committer is asked to enter to describe or justify the change.

Different systems use different strategies for recording changes—for example, some store only the differences between versions, while others store full snapshots of the content as it appeared at the time of the commit (more on this in section 0). But the high-level conceptualization of a chronological record described above is common to all major VCSs, and it is this that I will refer to when speaking of “commit history” or “commit log”.

There is an important distinction in terminology to be made here between “history”, as what actually happened, and “history”, as the record of what happened. In technical discussions about version control, there is a tendency to use “history” to refer to the latter. This is perhaps in part because version control systems have no notion of there being any other concept of history. That is, VCSs are never able to passively observe history as it happens. The VCS is always exposed to changes that occur over time in terms of individual commits, explicitly triggered by a user. After making a change (for example, via a text editor), the user must save the file to disk and then take some explicit action to designate the content’s new state as a version (i.e. to commit it). Effectively, the act of making a change—editing the file’s contents—is entirely separate from the act of registering that change with the version control system.

This decoupling between the tools used to create and modify content and the tools used to record those changes has serious implications on what a version control system is ultimately able to do. In effect, the VCS sees its content by way of intermittent snapshots; it is able to see the result of change but not the act of change. Any notions about what happened in between those snapshots must either be inferred or provided by the user. This makes certain assertions about content identity difficult if not impossible, and is the root cause of many of the limitations discussed in subsequent sections. Section 0, for example, examines how the commit-based notion of history poses a number of difficult challenges for tracking the movement of content (for example, if a
paragraph or text is moved from one place to another; since a VCS cannot see the movement as it happens, it must be explicitly informed of it by the user, or otherwise it must infer it through retroactive analysis of its results). Section 2.7, however, makes a case for why such a seemingly impoverished view of history (i.e., of what actually happened) can be advantageous, as it avoids a set of problems inherent in a history that seems rich in information, yet is captured in the wrong terms.

Before moving on to further discussion of temporal concerns, it is worth taking a quick detour to explain the VCS concept of a “repository”, how it relates to a “project”, and how that relationship bears on the way projects are employed and perceived by users.

2.3 PROJECTS AND REPOSITORIES

Content in version control systems tends to be divided up in terms of repositories. A repository is essentially a collection of related files and folders, each with their own history. The repository draws conceptual and practical boundaries around its content; it is something like a self-contained universe, complete with its own historical record. That record can have multiple alternative timelines, so that changes to content occurring in one timeline are not present in another. Timelines can split off and merge back together many times over, but they always trace back to a single point—a so-called “initial commit” that captures the state of the repository’s content at the time when it was first added. Those timelines can diverge and converge within a repository, but with few exceptions, they cannot do so with a timeline from a different repository.

Exactly what makes one repository different from another is a rather nuanced question worth examining in detail. First, note that there exists an important distinction between centralized and distributed version control systems. The older generation of VCSs such as Subversion and CVS were “centralized”, in the sense that all users submitted their work to a single, central repository. With centralized systems there was little confusion about what made one codebase different from another. There was only one instance of each repository, and that instantiation was, for the most part, synonymous with identity. Newer VCSs however are “decentralized” or “distributed” (see sidebar on the next page on the generational differences in VCS). Each user has one or more local copies of the repository, to which they commit their work, and which they synchronize intermittently with other copies (typically residing on other servers or other workstations). Thus, in distributed version control systems, instantiation is separate from identity. A “different” repository is one that was not at some point cloned from one of the other repositories in the network.

*In some systems (for example, Git), one can “graft” together commit logs from unrelated projects, effectively combing unrelated timelines to form a new initial commit. (Fonseca, Narebski, and Thurner 2012)
But despite a very different take on identity in terms of instantiation, there is a level of abstraction in which both centralized and distributed systems predicate repository identity on the same, singular notion—the initial state of their content, also known as an “initial commit”. That is, two repositories with identical initial states can be said to be “the same” repository. If their content is different, it is only because they are “out of sync” with each other. In practice, however, the centralized VCSs tend to not recognize this abstract identity. Subversion, for example, uses an explicit, arbitrarily-assigned UUID (“universally unique identifier”) to identify each repository, so that two repositories with different UUIDs are considered different and not interoperable, regardless of whether their histories have identical initial commits. (Collins-Sussman, Fitzpatrick, and Pilato 2011)

The different ways in which systems choose to implement identity is interesting from a philosophical point of view, but it is also problematic in a very practical sense. Consider, for example, the relationship between projects and their repositories. Although multiple projects can be associated with one repository and a single project can be spread across multiple repositories, in recent years project hosting services like GitHub, Google Code, and Sourceforge have helped entrench a one-to-one relationship between repository and project. Because these services bundle a single hosted VCS repository within each project they host, a project does not just have a repository, it is entirely organized around that repository, and that repository becomes central to its identity. In most ways that count—how it appears users, how it is managed and governed, how it is referred to in discussion, and so on—the project is the repository.

The problem with making project identity so contingent on the VCS notion of repository identity, is that it is too impoverished to support users’ intuitions about the matter. For users and developers, software projects tend to have rather nuanced and hard to pin down identities, predicated on a complex mix of social, historical, aesthetic, and practical concerns. This mismatch

### Three Generations of Version Control Systems

<table>
<thead>
<tr>
<th>Generation</th>
<th>Networking</th>
<th>Operations</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Generation</td>
<td>Networking None</td>
<td>Operations One file at a time</td>
<td>Concurrency Locks</td>
</tr>
<tr>
<td>2nd Generation</td>
<td>Networking Centralized</td>
<td>Operations Multi-file</td>
<td>Concurrency Merge before commit</td>
</tr>
<tr>
<td>3rd Generation</td>
<td>Networking Distributed</td>
<td>Concurrency Changesets</td>
<td>Commit before merge</td>
</tr>
</tbody>
</table>

(adapted from Sink 2011; and Raymond 2009; note that this is a sampling of VCSs rather than an exhaustive list)
between how users and developers think about projects and how those projects can actually behave is a source of frequent confusion, and a growing problem for the quickly-growing thicket of open source projects.

For example, on GitHub, users are encouraged to create their own clones or "forks" of existing Git repositories (and thus projects). But because project identity on GitHub is largely predicated on Git’s simplistic notion of a single common root version (an "initial commit"), projects that have wildly diverging histories and are thus very different in just about every way, are presented as forks of "the same" project, whereas two entirely identical code bases mistakenly imported as two different Git repositories are considered by GitHub to be two entirely distinct projects. The former can often lead to confusion about authority, for example when the original repository is abandoned by its author and a fork becomes the de facto authoritative continuation of the project, with little obvious presentation of this fact to new users. The latter can lead to confusion about authenticity, whereby it is difficult to tell which unconnected copy of the repository is the authentic one, that is, the one sanctioned by the original author or supporting community.

2.4 CHANGE

The change between two versions of the same piece of content is referred to as a delta—that is, a "difference". Most VCSs have facility for showing these difference to the user, usually by way of a specialized visual representation called a diff.

For instance, consider the following two version of a piece of text, where the word "dog" is changed to "turtle":

The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy turtle.

A visual representation of the difference between the two might have the word "dog" highlighted in red to indicate that it was deleted, and "turtle" highlighted in green to indicate that it was added, like so:

The quick brown fox jumps over the lazy [¬dog¬] [+turtle+].
To generate these representations of change, most VCSs rely on some variety of the ubiquitous diff* utility. diff is an old and relatively simple piece of software developed independently of version control systems. It has no intrinsic notion of temporality and can be used as a standalone tool to compare any two text files, regardless of whether they have some temporal relationship (that is, regardless of whether they are different versions of the same document).

In its canonical form, diff looks for differences only at the line level. A single-character difference in a line of text will cause diff to designate the entire line as having been replaced by a new line. For example consider the following edit:

```
Row, row, row your boat
gently down the stream.
```

... subsequently changed to:

```
Row, row, row your skiff
gently down the stream.
```

The textual representation generated by applying the diff tool to these two pieces of text—the output itself is also called a diff—would look like this:

```
-Row, row, row your boat,
+Row, row, row your skiff,
Gently down the stream.
```

Note that the entire first line is marked as having been deleted (in red), and a new line has been added (in green). Quite obviously, this is not the only way this change could be recorded. Only the last word was changed, so why not record this as a change to the last word rather than the whole line? And why not record it as a modification rather than a wholesale replacement? Note that if we consider the line (or word) as altered rather than replaced, we retain some sense of identity between the first and second version that would otherwise be lost (more on this later).

These distinctions matter, at the very least because each describes a different sort of change. The choice of encoding the change as one sort of thing over another can misrepresent what actually

* Both the tool and the representations it generates are called “diff”, so diff will be used to refer to the tool, and diff to the output of the tool.
happened, and it can do so irrevocably, as such abstraction can strip away information necessary for any retroactive reconstruction of an alternative take.

Imagine a scenario where you and I are two authors editing the "row, row, row your boat" verse. You change "boat" to "canoe" in your repository, and I change all instances of "row" to "paddle" in mine (Figure 2). Intuitively, there is no reason why our mutual changes should conflict—we are each changing different words. Yet a typical version control system—which by way of diff sees the text only in terms of lines—would posit that we had both altered the first line. As such, our two versions of the verse are in conflict and will have to be manually reconciled, since the system has no definitive way of deciding which version of the new line is the one that should be retained.†

![Diagram of line-level conflict]

Figure 2 – Line-level conflict

One might be tempted to argue that this is simply a matter of granularity—that the conflict could be averted by re-tuning the diff tool to look at text in terms of words rather than lines. Indeed, alternative implementations of diff—namely wdiff or "word diff"—do exactly this. But, while advantageous in this particular case, such a switch from lines to words is more a lateral trade-off rather than a wholesale improvement. To borrow a cliché, the switch allows us see the trees at the cost of losing sight of the forest.

† That is, assuming that there really is some “actual” matter of fact that ought to be recorded. Whether such absolute fact exists is debatable, so perhaps it is better to say here that the choice of encoding can misrepresent the author’s intentions—arguably a more useful thing to record than anything else. But then again it is not at all clear whether there is any matter of fact about those intentions either.

† The question of how one choice of representation over another can impact collaboration is examined in detail in section 2.8.2.
The reasons why *diff* parses content in terms of lines rather than some other unit is largely historical, stemming from constraints imposed by the infrastructure of the early UNIX systems in which it was developed. Those constraints influenced early text editors to allow editing only on a per-line basis (Ritchie and Thompson 1970), and this concurrently evolved with, and into, the typical usage of a line as a basic unit of instruction in many programming languages.

But while in source code a line is often more meaningful than a word, in the above "row your boat" example—written in English rather than in a programming language—the choice of a word over a line as a basic unit seems more appropriate. Lets assume then that we are using *wdiff*, which parses text in terms of words rather than lines. A *wdiff* of the change from "boat" to "skiff" would look like the following:

```
Row, row, row your boat [-boat,-] {+skiff,+}
Gently down the stream.
```

This is certainly a better representation of my—i.e. the author's—intentions. The way I see it, I did not replace the whole line; I just changed the word "boat" to "skiff"

But what if later I decide to make a change only at the character level, for example choosing to capitalize "Boat"? *wdiff* does not know anything about the rules of capitalization in the English language, so it would see these as two different words and its output would have "boat" seemingly replaced by a different word, "Boat".

Or, what if I decided to switch to writing in Chinese instead of English? A Chinese version of my verse would look something like this:

```
排排排你的船
轻轻地往下流
```

In written Chinese, the notion of 'words' is more nuanced than in English. For one, note the absence of spaces. Depending on semantic context, a character may be a word in itself, or it might be part of a word made up of multiple characters. Even in English the rules are not as simple as they might seem on the surface. If "bedroom" is one word, is "living room" really two, or is it just one word with a stray space in the middle? Consider that just a few decades ago, "tomorrow" was usually written as "to-morrow".

The path to a better *diff* thus cannot be a simple matter of finer granularity. Granularity is highly contextual. Text can be parsed in a multitude of ways, and the appropriate choice of parsing
often depends on the semantics of the content and the intentions of the writer and the reader—
aspects that are difficult if not impossible to capture.

2.5 SOURCE CODE

The complexity and nuance of ordinary language may be too much for diff, such that it may be
impossible to formalize and encode exactly what has changed between two versions of some piece
of text. But what about programming languages? After all, these languages are specifically designed
to be parsable by computers. And, one would imagine, they ought to be considerably less
ambiguous about meaning than ordinary speech.

Indeed, in recent years alternative versions of the diff tool have been developed that are able
to leverage the predictable syntax of programming languages to automatically ascertain appropriate
granularity when assessing change between two versions of a piece of computer code. (Lahiri et al.
2012; Codice Software 2013)

Consider for example the following piece of code in Java:

```java
public String foobar() {
}
```

The above declares a function named foobar, with public exposure and a return value of
type String. Exactly what that means in Java is not important here, but worth nothing is that
unlike in English or Chinese, parsing this code into consistent, well-defined tokens is trivial, and
each of these tokens plays an unambiguous role within the language. For example, there are a
limited number of possible access modifiers (public, protected, and private), they must always
occur before the function name, they are always separated from other tokens by whitespace, they
are only allowed in certain formally-defined contexts, and so on. There is also much less ambiguity
about whether a function is "the same" function as another. The function’s signature (its name,
return type, and the arguments it takes) is enough to completely identify it, as one cannot declare
two functions with the same signature inside one class.

Thanks to all this ingrained regularity, a syntax-aware diff tool is able to make all sorts of
deterministic identity-preserving inferences that would be difficult if not impossible in human
language. For example, had I changed the above to:

```java
public Integer foobar() {
}
```
... a syntax-aware `diff` would have little trouble identifying it as the same function, only with an altered return type. Or had I moved it to another part of the source code file, there would be no ambiguity about whether the action was a case of deletion-and-creation versus movement.

Given all this, it would seem that syntax-aware `diff` tools are unquestionably superior to the traditional version. Why then do all of the popular version control systems continue to rely on the relatively crude, line-based implementation of `diff`? Although there are certainly compelling reasons to switch, the next two sections try to show why there are situations where awareness of syntax is not useless, but downright deleterious.

### 2.5.1 AESTHETICS

It is important to note that computer source code has two very different roles. It is a set of instructions directed at the computer, defining how it should transition between various internal states based on external input. But it is also a mode of communication between developers, about the program’s behaviour and about the external domain the program models or interacts with. This second role—one often overlooked by the layperson but well known to experienced programmers—introduces much of the confounding ambiguity of human language back into source code. For one, code often contains blocks of comments written in prose. These are generally ignored by syntax-aware `diff` and treated using traditional `diff` rules. But the nuance of human communication also frequently finds its way into the code itself, where it is not as easy to filter out.

For example, most programming languages are somewhat loose in their rules about whitespace. In Java, the following two blocks of code have exactly the same effect:

```java
public String foobar() {
}

public String foobar()
{
}
```

Note the different placement of the curly brackets. A syntax-aware `diff` tool would rightfully recognize a change from the former to the latter as inconsequential, because it would have no impact on the program’s behaviour (the two would likely compile to identical machine-level bytecode). Yet to a human reader there is a noticeable difference. Indeed, these two styles of writing Java function declarations are the subject of endless arguments among developers, with disagreement over the readability or aesthetics of one over the other. Developers—acting on the very human impulse to find ways to express oneself in every possible medium—have subverted this laxness in whitespace rules for all sorts of creative purposes.
One can tell a lot about a programmer just by looking at the way they use white space. I can usually guess what language a programmer wrote in prior to learning Java, how conscientious or fastidious they are, and even the level of resentment for their current job. In the interest of standardization, organizations often insist that all developers stick to one particular style of writing function signatures. I once worked at one such organization where very soon after the policy was introduced, some of the more disgruntled employees deliberately began using every style but the one that was sanctioned (along with taking all sorts of other, more egregious liberties with Java’s loose whitespace rules). The violation of the company’s guidelines became a kind of silent, passive protest—an expression of opinion that was clear to any programmer reading the code, but totally opaque to a syntax-aware diff.

Of course the syntax-aware tool is able to detect these sorts of changes. But since it cannot discern their meaning, it labels them as insubstantial—as changes to mere presentation rather than actual content. This relegation of the human aspects of code to mere fluff is, at best, naïve. If one buys into the premise that programming is more art than science, both in practice and in product—see Graham (2004) or Brooks (1995), for example—then the syntax-aware diff’s naïveté becomes outright deleterious, as it fails to respect the way programmers actually work, wherein code is a rich medium rather than a mere set of instructions for a computer. As such, the promise of the syntax-aware diff as a general improvement over its simpler, syntax-ignorant predecessors is only superficial. In the broader scope that encompasses source code’s human aspects, one could argue that these tools may actually be counter-productive, abstracting away much of what enables the subtle intricacies of software development in practice.

2.5.2 MEANING

Another, strictly practical reason why syntax-aware tools have not gained wider acceptance is the considerable effort required in tailoring the diff tool to each language’s particular. A syntax-aware diff designed for Java for example will not be able to make sense of C++ or Ruby. Implementing a full-fledged parser for each of these language is not trivial, and each such implementation must be kept up to date as its target language continues to evolve. In short, the work required to build, and maintain these tools might outweigh their benefits.

The situation is further complicated by the so-called dynamically-typed languages. The discussion so far has referred largely to statically-typed languages like Java, in which the identity of various elements in a program’s source code can, to a certain degree, be determined a priori, without ever running the program. But in some of the most popular languages in use today—the dynamically-typed Python, JavaScript, and Ruby—that sort of deeper analysis is not possible without actually running the code and analyzing its behaviour. These dynamically-typed languages can modify or introduce higher-level grammatical structures during—and as a result of—their own
execution. And, most problematically for syntax-aware tools, they can do so based on arbitrary external input from the user or environment. As a result, the identity of various tokens in dynamically-typed code can change in ways that cannot be definitively determined a priori. Since running two versions of a piece of code every time one wants to see how they differ is infeasible, and doing so for all possible inputs is outright impossible, the nature of dynamically-typed languages effectively limits the possible usefulness of syntax-aware diff tools, drastically reducing their ability to make deterministic inferences about what changed.

This difficulty inherent in dealing with dynamically-typed languages is another hindrance for the greater adoption of these tools, but it is not necessarily a fatal one. Some level of deterministic syntax analysis is still possible, and for the purposes of displaying change to a user—as opposed to definitively encoding it—probabilistic analysis (e.g. "48% chance that this line is the same as that other line") may be good enough. And, looking at the problem from a different angle, in the perpetual debate over the merits of dynamic versus static typing, the problems associated with determining token identity in dynamically-typed languages may serve as an argument in favour of greater adoption of statically-typed alternatives. That is, the fact that we can build better diff tools for statically-typed languages than for dynamically-typed ones might be a good reason to write more code in Java or Scala rather than in Ruby or JavaScript.

There is, however, a more fundamental problem that places an ultimate limit on what a syntax-aware diff can do. This limit exists even for those statically-typed languages that provide rich and sophisticated systems for formal analysis—languages like Haskell and Scala, with their support for algebraic data types. The problem—and there are actually two, but they are almost certainly related—depends in large part on how one defines identity.

What a syntactically-aware tool means by two things being "the same"—a question explored in later sections—is problematic because the sort of identity that can be established by syntactic analysis alone falls short of what most of us have in mind when we take two things to be the same. Syntactic analysis can, for example, establish that a token in a piece of code—say a variable named foo—is, in fact, the same token when it appears in a different part of the code—the same foo, rather than some other, superficially identical foo. But what it cannot determine is whether that foo refers to the same thing in one version of the code as in another. That is, syntactic analysis cannot, by definition, determine the meaning of tokens.

This is a point worth making, because it is one often overlooked. In fact, syntax-aware tools are sometimes misleadingly referred to as "semantic" (see for example, Lahiri et al. 2012 and Codice Software 2013). The suggestion is that by analyzing the syntactic role of the various tokens in source code, the tools are able to derive their meaning. This is true only if one buys into an extremely limited sense of "semantics", wherein meaning is defined by the ways in which a token can and
cannot be used within the syntactic rules of the language. For instance, in Java, a token can be recognized as a method rather than a variable by virtue of how and where it appears in the source code—method names, for one, are always followed by an opening parenthesis, whereas variables never are (see Figure 3 & Figure 4).

Figure 3 – Variable declaration and assignment in Java

```java
public String foo;
foo = "bar";
```

Figure 4 – Function declaration and assignment in Java

```java
public void foo() {
}
something.foo();
```

But while potentially helpful as a heuristic for determining internal identity (the foo in Figure 3 is almost certainly not the same as foo in Figure 4, since by the syntactic rules of the language the two are likely to refer internally to very different kinds of things—one a variable and the other a function), the distinction tells us next to nothing about what role either foo might have in regards to the external world.

2.6 PERSISTENCE

I have said about the limits of syntax-aware diff tools' ability to fully and unambiguously describe change. But it is important to keep in mind that despite these limitations, in practice syntax-aware tools are able to make some reasonably good guesses. If the inherent implementational and infrastructural problems are eventually overcome, it is entirely possible that syntax-aware diff will one day become more commonplace. For example, such tools could pave the way for better visual representation of change (via better colour coding of different types of changes, for example), so that a user could more easily glance at a diff and get the gist of what it implies.

But visually representing change is one thing. The hard part—grasping the full meaning of that change—is left to the user. Leaving the interpretation up to the human operator is entirely acceptable—even desirable—when the version control system is used as a tool for browsing a repository's history. But showing changes is only part of the problem. An VCS must also encode and store those changes, and it must do so in a way that allows for reliable recreation of a repository's state and manipulation of its commit log. This is where the difficulty in designing a version control
system crosses the line from one of epistemology to one of ontology—of having to take a position on what change is, not merely how to best analyze and describe it.

This problem has been approached in a number of different ways by different version control systems. The first few generations of VCSs recorded their content's history as a series of transformations—that is, byte-by-byte representations of deltas, indicating which bytes were deleted and which bytes were added between each version. Imagine for example that you had edited the "Row your boat" as follows:

Figure 5 – Changes as snapshots

**Version 1**

Row, row, row your boat,  
gently down the stream.  
Merrily, merrily, merrily,  
life is but a dream.

**Version 2**

Row, row, row your skiff,  
gently down the stream.  
Merrily, merrily, merrily  
life is but a dream.

**Version 3**

Row, row, row your skiff,  
gently down the river.  
Merrily, merrily, merrily  
be weary of the angry beaver.

Under older version control systems like CVS and early editions of Subversion, the commit history—composed entirely of diffs or "deltas"—would look something like this:

---

*To better accommodate non-textual content, and for optimal efficiency, Subversion uses a specialized diff format based on byte-level addresses rather than lines. For the purpose of illustration however, I show the diffs as standard line-based diff output.*
Figure 6 – Changes as deltas

Version 1

0a1,4
> Row, row, row your boat,
> gently down the stream.
> Merrily, merrily, merrily,
> life is but a dream.

Version 2

1c1
< Row, row, row your boat,
---
> Row, row, row your skiff,

Version 3

2c2
< gently down the stream.
---
> gently down the river.
4c4
< life is but a dream.
---
> be weary of the angry beaver.

The details of this format are not important; the thing to note is that rather than storing the text as it appeared at each version, only the difference between each successive version is stored. The very first version is represented as an addition of four lines (0a1,4 means "the following four lines were added at line 0"). The second version stores only the replacement of the first line (1c1 means "1 line changed at line 1"). And so on. Note that in order to find out what the full text looked like at any particular version, the system would have to "play back" all of the transformations, starting with the very first, all the way to the desired version number, effectively re-applying the series of actions that led to that particular state.

One advantage of this way of storing versions—besides a kind of conceptual tidiness in thinking of an object's history as a chain of changes—are the savings in the storage space required to retain the full history. However, this delta-based storage scheme was not necessarily chosen by the systems' designers because it is inherently superior. Like so many aspects of version control systems' design, the scheme may have simply been an adaptation of practices predating the widespread use of VCS. Prior to (and even after) the adoption of VCS in programming practice, changes to code were often communicated between programmers via diff-like "patch" files.
transmitted by email, containing just the transformations needed to import a collaborator’s modifications to one’s own copy of the code.

However, this delta-based storage scheme has some major practical pitfalls. One problem is that if even one delta in the series were to be lost or corrupted, it would be impossible to accurately recreate any subsequent versions. Such archival fragility is an obviously undesirable trait for a version control system. Another problem is that if the history is very long, recreating a file’s or repository’s state at any given point can be computationally expensive, making certain operations on such repositories painfully slow. Finally, as I have argued at length, the explicit storage of each change as one particular kind of transformation over another—say a deletion of a line and addition of another rather than the movement of the line to a new location—will unavoidably miss some part of the story, at the very least because it demands an irrevocable commitment to one level of granularity over another.

It is for these reasons that some of the newer version control systems—Git, and to some extent Mercurial—have opted for an alternative, orthogonal representation of change. Instead of encoding history as a series of transformations these systems represent it as a series of snapshots. Each snapshot contains the full and complete content, as it appeared at that point in time. As such, the structure of a Git repository resembles Figure 5 more than Figure 6.

So whereas in older systems state is derived, and transformations are the first-class entities, in these newer systems it is the transformations that are derived, with content snapshots reified as first-class entities.

It is worth nothing that although the above is an accurate high-level overview, the situation is more complicated when one looks at how these systems actually encode and store history. Git, for example, archives each change as a full content snapshot, but then applies a “packing” algorithm to compress the repository’s history, so that much as in systems that represent history in terms of transformations, only the deltas are actually stored on disk. Git boasts improved archival resilience not because it is impervious to the problems of rebuilding state from lost or damaged deltas, but because it uses cryptographic hashing to validate and enforce integrity whenever the repository is operated on. Mercurial’s design on the other hand takes a hybrid approach, employing a mix of deltas and snapshots. Much like in older systems, a file’s history is represented in terms of deltas, but only until it reaches a certain size threshold, at which point the entire snapshot is stored. This ensures that any delta-related corruption is limited to smaller chunks, going back to the last full snapshot. (O’Sullivan 2009)

As for the problem of representing changes per se—that is, deciding whether to register a given change as one sort of thing over another, for example a move versus a deletion and addition—an indirect benefit of a snapshot-based history is that it makes it possible to skirt the issue altogether.
Because the changes are not explicitly represented at the time when they are committed, their representation can be deferred, until, for example, the user asks to see the difference between two versions, or until diverging histories must be merged together.

Consider for example the way Git deals with file renames. In Git, files do not have explicit identity, at least not the way we typically think of it in terms of documents or containers. Rather, the tree structure of tracked files and folders is stored as a whole, separate from the files’ content, with the tree treated as if it were its own document. If a user renames a file called "row_your_boat.txt" to "row_your_skiff.txt", a Git repository would record the first version as a tree containing a node named "row_your_boat.txt", and the second as a tree with a node named "row_your_skiff.txt". Note that there is nothing explicitly linking those two nodes; nothing directly indicating that the node labelled "row_your_skiff.txt" in the first version is somehow the same node as the one labeled "row_your_boat.txt" in the second. Just as Git does not have any particular introspection into the identities of the elements of any of the other kinds of content under its purview—it does not know that a line of text that reads "row, row, row your boat", later changed to "row, row, row your skiff", is actually the "same" line—it likewise has no facility for explicitly tracking the identity of individual files.

Nevertheless when a user asks to see the difference between two versions of a directory in which a file was renamed, Git will (usually) have little trouble showing that such a name change occurred. The git-diff command line tool would, for example, show output resembling the following:

```
  diff --git a/row_row_row_your_boat.txt b/row_row_row_your_skiff.txt
  similarity index 100%
  rename from row_row_row_your_boat.txt
  rename to row_row_row_your_skiff.txt
```

The software is able to do this by way of retrospective inference. It detects that the two blobs of content associated with the two nodes are very similar—in this case identical, as indicated by the "100% similarity index"—and guesses that those nodes likely represent the same file. The algorithm Git currently uses for detecting this similarity is rather simple; the score is nothing more than a comparison of the number of lines that would have to be added. This is technically known as the "Levenshtein distance" (Ukkonen 1985). If this number is zero, the similarity score is 100%. If one line were deleted out of a total of 1000, the score would be 99.9%. And if all of the lines would have to be deleted or added, the similarity would be 0%. A configurable option in the git-diff utility determines the percentage threshold at which a content blob is considered to be the same or different. By default, this is set to 50%. (Hamano 2013b; Hamano 2013a)
Of course one can think of many ways in which this method of detecting similarity might go wrong. And indeed, it sometimes does. One might also think of more sophisticated algorithms Git could potentially use that might provide more accurate results. For example, a proposed patch to Git’s similarity detection would apply the Levenshtein algorithm not only to the content blobs but also to filenames within the tree, so that filename similarity could be used to augment the content similarity score. (Schindelin 2009) Syntax-aware content analysis could also be helpful here, since certain sorts of structural changes might be indicative of similarity, even if at a byte-by-byte level they appear very different (for example, as discussed in prior sections, in many programming languages, certain white-space differences can be ignored).

The important point here however is that the accuracy of such rename detection does not have any effect on Git’s ability to reliably keep track of its content. Rename detection is employed strictly as a user interface enhancement. Linus Torvalds, the chief designer of Git, once remarked that this functionality is implemented only as a matter of "politeness" to the user (Torvalds 2007). The version control system, Torvalds argues, has no business knowing anything about the specific syntax or semantics of its content (the notion of a "file" and the operations that can be performed on it counting as a kind of syntax). Any attempt at doing so introduces problematic complexity, and, more importantly, the possibility of embedding inaccuracies in the story-as-written. Instead, Git’s design—and in this Git differs fundamentally from Mercurial and its other peers—focuses on writing a history that tells a story told by the computer only where the computer can tell that story unambiguously. The rest is left out, to be told retrospectively by the user or by computer-aided analysis.

Although the level of "politeness" offered by Git is debatable (supporters of Mercurial and other competing systems often talk with disdain about Git’s unintuitive user interface), Git’s conceptual simplicity makes it fast, efficient, and reliable, and has undoubtedly played a role in its widespread adoption. Other systems have taken a less principled approach, attempting to find a middle ground between Git’s insistence on the opaqueness of content, and the need to accommodate human intuitions about identity. For instance, Mercurial explicitly records each file rename by recording the change as metadata within the file’s history. This effectively gives files some semblance of identity within Mercurial’s conceptual model. But it comes at the cost of introducing additional entities, which, due to the limits imposed by any such fixed level of granularity, can ultimately tell only part of the story.
2.7 DEFERRED REGISTRATION

In section 2.4 I tried to show how the necessary act of registration—of conceptualizing something as one sort of thing over another (for example, as a series of lines versus a series of words)—limits a version control system’s ability to refer to its content, and in turn limits what it can say about that content. Registering text in terms of lines, for example, obscures the identity of the words within those lines. But choosing a more fine-grained level of registration—say, individual characters—comes at the cost of the story at other levels of abstraction. Moreover, such "levels" are not necessarily hierarchal; one registration could be orthogonal to another, for example as when one registers text in terms of columns rather than horizontal lines. The problem then is not just a matter of choosing the right "thickness" for slicing up one’s subject matter, but also choosing the right direction in which to cut it.

The situation is no different with file renames. In fact, the observation that renames are just a special case of the more general problem of tracking the movement of content (with, for instance, another case being the movement of some block of text within a file), is a core tenet of Git’s design. (Torvalds 2005a; Torvalds 2005b)

Git’s design tries to get around the problems arising from such decisions about representation by avoiding or deferring registration as much as possible. Yet ultimately Git cannot avoid the need to register its data as something. It keeps track of content in terms of blobs, directory trees, and commits, and if one were to count entities in the underlying infrastructure, as inodes, blocks, bytes, and such (Chacon 2009a). But unlike most other VCSs, Git intentionally limits its ontology to only those entities absolutely necessary to provide resilient and consistent storage of versioned data, and nothing more.

The implications of this minimalism are worth considering in detail. Like Mercurial, Git allows users to browse content in terms of files, complete with filenames that can change over time. But unlike Mercurial, Git does not record the filename change as a distinct act or event per se. In Git, a file (i.e. an entry in a tree object) has a name in one version, and a different name in another. There is nothing explicitly linking the two. Imagine a set of two photographs taken in sequence, in which someone is shown wearing a cowboy hat in the first, and then—without explanation—a beret in the next. In Mercurial, and most other VCSs (Bazar, Subversion, etc.), all such photographs would come with something like a callout caption (a labelled arrow maybe), strictly reserved for explaining that the person in the first picture is the same as the one in the second. Note that both Mercurial and Git allow for free-form, user-written notes attached to each commit, but what we are talking about here is something else. The "person-tracking" caption, like the file-tracking one,

---

* Borrowing the term from Smith (1996, 191-198).
would have to be processable without the need for human interpretation, so its format would have to be highly constrained and formalized.

Attaching such metadata seems perfectly sensible. It provides a clear, unambiguous explanation of what happened. Surely it is better to have this extra bit of information than to have nothing?

There is a famous photograph from the Crimean War—one of the earliest instances of wartime photojournalism—in which a battlefield road is pictured littered with cannonballs (Fenton 1855). This photograph recently became the focus of controversy when it was noted that there is a second version in which the road is clear of cannonballs (Figure 7). Because the original photographs were not numbered, nor did the author provide any notes explaining the discrepancy, it was argued that the famous scene was in fact staged. Susan Sontag, the author and political activist, cited the picture as a prime example of war-as-theatre, of drama produced for mass consumption, detached from the reality it purports to represent (Sontag 2003, 49–51). Others however countered that it was Sontag who misrepresented history—the scene without the cannonballs was simply photographed later, after the road had been cleared to make it passable. (E. Morris 2007; B. Morris 2011)

Surely the Crimean War photos are an example of the need for Mercurial-style metadata when our view of history is limited to these sorts of intermittent snapshots. But had Roger Fenton, the original photographer, included any such notes, there is no guarantee that his account would have been any more truthful than the photographs. If Sontag is right and Fenton had fabricated the scene, he could just as easily have added notes to corroborate the illusion. And had such notes existed, historians may have had less reason to question the photos’ veracity.

Fabricated histories are not a common problem in version control systems, although they are certainly a possibility. Unintentionally misleading histories, however are a concern. A version history that focuses on the wrong details—for example, telling a superficially mundane story of files
whose names had changed when the author’s actual intent was a project-wide adoption of a new structure—can be insidiously problematic in that they are at once accurate and misleading. By analogy, a history of the World Wars retold only in terms of the participants’ technological achievement might be a valid and captivating story in itself, but is certainly not the whole story, and reading it as such might prove misleading to a naïve reader unaware of the bigger narrative.

One of the reasons why Git’s design intentionally rejected the notion of a file rename as an explicit meta-entity, a fixed part of the system’s vocabulary for describing change, is that its use introduces the danger of exactly this sort of misrepresentation—of fixing the history to one particular telling that might fail to capture some unanticipated but important aspect of what happened.

Consider the argument most commonly cited in favour of rename tracking: explicitly tracking filename changes bestows files with identity (something that they would otherwise lack), making it much easier to trace the history of any particular file. Without this identity, an individual file does not have any continuity through its revisions. Imagine, for example, that you have a file called “skiff.txt”. You want to see how its content has changed over time, all the way back to when it was first added. But what if it was at some point renamed, so that the file was once called “boat.txt”? If your version control system does not explicitly track this name change, it cannot easily infer that when traversing backwards through its commit log, where “skiff.txt” disappears it should start looking at “boat.txt” instead. That is, the history does not provide any explicit link between the content associated with “skiff.txt” and “boat.txt”. This is exactly the problem in Git, and exactly the sort of situation Mercurial’s rename tracking is meant to address. The notion of a "rename" in Mercurial effectively bestows a file with a sense of continuous existence, one that is predicated on its name.

But a file’s identity—at least the way we intuitively think of it—or more specifically the identity of the document the file represents (the relationship between files and documents is in itself problematic, but let’s put that aside for now)—involves a lot more than just the file’s name. A change in name is only one of many ways a file’s identity might be altered. Consider for example a scenario where an author decides to split one file into two. Having grown too big, "my_story.txt" becomes “the_beginning.txt” and “the_end.txt”. A version control system’s rename tracking, the way it is generally implemented, would not be of much help here. Mercurial, Bazaar, and the like, are only able to track the name change across an individual file. The user would have to designate either “the_beginning.txt” or “the_end.txt” as the new name for “my_story.txt”, or the two files would have to be recorded as brand new, without a link to their past history as “my_story.txt".
One solution might be to have the version control implement "split" tracking in addition to rename tracking. But then why not also add "join" tracking, for when multiple files are joined into one? Or how about "translation" tracking, for when multiple versions of the same text file are translated into different languages? And “inter-repository import” tracking, for when files are brought in from an external repository?

There is no end to the kinds of higher-order changes that such a system could support. But the difficulty lies not only in that there are an infinite number of things that can happen to a file—this is the sort of problem the designers of just about every software system must deal with, and well designed systems tend to be the ones that correctly anticipate only the minimal set of functions that should be implemented to support users’ needs. The more fundamental problem here is that because the system’s vocabulary for describing changes will always be finite—limited to what was anticipated by the system’s designers—the record of what happened will unavoidably be an approximation, told in the limited terms available at the time when the change was recorded. Moreover, the problem isn’t just that that record is an approximation (arguably all accounts of history are), but that since it can be inscribed only in the terms sanctioned by the system’s designers, it can be actively misleading, as when a system lacking a notion of “split” tracking (see above) marks "the_end.txt" as the only new name for "my_story.txt", obscuring the fact that the "the_beginning.txt" was equally deserving of the designation.

Git attempts to get around this problem by choosing not to tell any story about files at all. Instead, it leaves the narrative to be told in retrospective, by tools that try infer it, looking backwards at the raw version snapshots. The idea is that since we can expect the vocabulary for describing files' identities to keep changing, and since the algorithms for inferring a shifting sense of identity are likely to keep improving, and—maybe most importantly—since we may find ourselves looking at that history from previously unanticipated perspectives, it is better not to write the history as it happens but to read it out from the always-changing present, knowing that the story we see tomorrow may be different from the one we are able to infer today.

Earlier, drawing the parallel with the Crimean War photographs, I left out the last part of the story—one that may help illustrate the merits of Git’s non-committal approach, rooted in the assumption that it is better to count on the future improvement of retroactive analysis techniques than to try to capture historical narrative as it happens. The debate over whether the cannonball scene had been staged remained unsettled until 2012, when an amateur historian noticed a subtle discrepancy between the two images. Using Photoshop, he superimposed both versions and created an animation that rapidly switched between the two. Juxtapositioning the pictures in this way, the historian saw something that had until then gone entirely unnoticed; darker shapes around many of the rocks strewn around the scene—previously dismissed as shadows—were in fact trails in the
surrounding dirt. In the photo with the cannonballs the rocks had slid down the slopes beside the road, almost certainly due to having been recently trampled over. The most likely explanation is that workers, exerting considerable effort to roll cannonballs onto the road, kicked over many of the surrounding rocks, causing them to slide down the slopes, leaving behind the small trails captured in the second frame. There was now little doubt left. The photographs had almost certainly been staged. (E. Morris 2007; Palmer 2009; the two-frame animation of the photographs can be viewed at http://sbp.so/fenton)

2.8 MERGING

Torvalds’ minimalist take on what should and should not be represented in a version control system seems to have paid off. Git has become ubiquitous in software development, and is increasingly finding uses outside of programming (see for example Hynaszkiewicz 2013; Stieben 2011; Bons 2013; “Penflip - a Social Writing Platform” 2013). The reasons for this success are debatable, but the system’s design has almost certainly played a key role. Git is fast, reliable, flexible, and—if one is willing and able to climb a relatively steep learning curve—quite effective at keeping track of even the most convoluted project histories.

This is not to say that all problems in version control have been solved. Far from it. Git’s innovation is not so much to solve those problems as it is to reframe them. The approach, essentially, is to defer as much as possible all acts of registration—i.e. of representing content and changes to content as one thing versus another—with the underlying assumption that: 1) such registration is largely perspectival, and thus best left up to whoever will be making use of that content, and 2) it is easier to develop and improve the mechanisms for registration later than it is to try to get them exactly right ahead of time.

This shift to using retroactive analysis of a relatively impoverished historical record to figure out what happened has allowed Git to sidestep some of the issues that had until then been considered pivotal. For example, the problem of handling file renames had been widely understood as a matter finding an optimal way to persist files’ identities across versions. That files had identities was largely taken for granted. The designers of most of the systems developed prior to or in parallel with Git (e.g. Mercurial, Darcs, Monotone) opted to explicitly represent each act of moving or renaming a file (Luchini, Ollivier, and Dimov 2006; Kow 2010). Others (e.g. Bazaar) assigned unique identifiers to files, much like ‘inodes’ in a standard file system (Canonical Ltd. 2011). Git was and continues to be the only major VCS that does not keep track of file renames at all. Torvalds considered the problem out-of-scope, and left it up to others to solve by way of post-hoc

---

* See, for example, the discussion in Ben-Kiki (2004).
2.8.1 CONCURRENCY

Most modern version control systems allow a repository to have multiple, parallel commit histories (see section 2.3). That is, an author can record a series of changes, go back to the start, and record an alternate sequence of changes that take the repository’s content in an entirely different direction. The same mechanism also allows multiple authors work to work on the same repository, so that each can record their own changes concurrently and independently of the others.

Note that this notion of “concurrency” is somewhat idiomatic. Changes in version control are considered concurrent not because they happen at the same time, but because they happen in sequences of commits that run alongside and independently of each other. For example, if I email you a copy of this thesis, and

---

History of Concurrency in Version Control

The long-term trend in the development of version control systems has been to allow users more freedom to concurrently make changes, at the cost of an increased likelihood of conflicts.

The earliest systems—SCCS and RCS—prevented concurrent changes altogether by enforcing strict content locking. Before being able to work on a file, the user had to obtain an exclusive lock that prevented anyone else from changing that same file. In principle, this had the advantage of avoiding conflicts entirely, but in practice such locks tended to impede collaboration rather than aid it. For example, users often forgot to release locks, needlessly preventing others from carrying out their work. RCS, a successor to SCCS, tried to mitigate the problem by allowing users to forcibly release each other’s locks. (Rochkind 1975) However, this meant that users could lose their locks prior to committing their changes, and the system would subsequently refuse to accept those changes. Moreover, this locking mechanism did not scale for larger teams and larger files. With larger teams, there is a higher likelihood of two users wanting to work on the same file. And with larger files, there is a higher chance of the file having content that two users might need concurrent access to.

(continued on p. 36)

---

* Provisory rename detection was implemented in Git by Junio Hamano (2005), a collaborator who joined the project after Torvalds released the initial source code. In a prior discussion on the topic, Torvalds (2005c) had said that he considered rename detection a secondary problem, left to be addressed by others.
we both subsequently make changes to our copies of the document—I change mine a day after sending it to you, and you change yours a week later—the changes would be considered concurrent, even though they did not happen at exactly the same time.\footnote{Not that true temporal simultaneity would make much of a difference. There is always some delay in communication. A shorter delay might affect how two collaborators negotiate concurrent changes, but it would not affect the need for reconciling how those changes are applied to the shared document.}

In this sense, changes in a VCS can be considered concurrent even when undertaken by one author. For example, since I am currently using Git to keep track of my work on this thesis, I could easily maintain a more formal version of the document in which I avoid all use of personal pronouns. Whenever I make sizable changes to one version, I would want to integrate those changes into the other. The act of splitting off a separate commit history in this way is called “branching”, and the ensuing sequence of commits is called a “branch”. These branches tend to have their own labels, and thus their own identities—for example, I might call one version of my thesis the “formal” branch, and the other the “informal”. A point in the commit history where I bring changes from one branch into another is called a “merge”, and any branch can have many such merges, as changes are intermittently brought from one branch into the other (Figure 8, below). \cite{Chacon2009b}

Consequently, the second generation of VCSs—client-server based systems like CVS and Subversion—eschewed locks in favour of a “merge before commit” strategy. That is, users were permitted to work simultaneously on the same files, but prior to being able to commit their changes back to the central server, the user was required to pull in any changes that others may have concurrently committed. Unfortunately, in practice this had some undesirable consequences; for example, since merging could sometimes turn out to be an onerous process \cite{Raymond2009, Sink2011}, users tended to hold off on committing their changes, effectively undermining the intended use of the system.

The third and current generation of VCSs—for example Git and Mercurial—are decentralized, and thus do not employ a central server able to keep absolute control of all commits. Instead, each user has their own copy of the repository, which they are free to commit to at any time. This allows for a “commit before merge” strategy, whereby a user is not prevented from committing changes even if such changes potentially conflict with those of others. \textcopyright{}\cite{Raymond2009, Sink2011}
Although this might not be the sort of thing that writers of prose typically do (perhaps because the typical tools of the craft do not easily support it), this sort of workflow has become common practice in software development. For example, new features are often developed in separate branches so that they can be worked on independently of the main version of the program—the main version might be concurrently receiving bug fixes, so it must be kept in a state of readiness for release, without those new, incomplete features (Fowler 2009; Driessen 2010). Moreover, distributed version control systems make use of the branching and merging mechanism to facilitate collaboration among teams of programmers. As shown in Figure 9, each programmer’s copy of the codebase is treated as a branch, with synchronization effectively equivalent to merging (Driessen 2010; Atwood 2007).

Prior to the rise of distributed version control systems, working with branches tended to be awkward and problematic (Brady 2011). For example, under Subversion, the mere act of switching

\*The use of arrows here is somewhat unorthodox. Typically, in a directed acyclic graph the arrows point the other way, from child to parent. Here, however, the arrows’ direction is used to indicate sequence, so that arrows show the order in which commits were added.
work from one branch to another could often take long enough that many programmers went out of their way to avoid it (“switching” meant running a command that would replace the files in a “checked out” copy of the repository on the user’s workstation with ones fetched from a remote repository server—a potentially slow operation). Likewise merging was often laborious and error prone. Because older versions of Subversion lacked the ability to keep track of merge points, one had to manually keep note of which changes had already been merged and which still needed to be reconciled (Brassman 2012; Fitzpatrick, Pilato, and Collins-Sussman 2006a, sec. Basic Merging; Fitzpatrick, Pilato, and Collins-Sussman 2006b, sec. Traversing Branches).

Given all of these problems, many programmers had come to avoid or minimize the use of branches, despite their potential advantages. Others resorted to onerous workflow processes to try to mitigate the pitfalls. For example, some teams established a practice of “merge days”—a dreaded weekly or monthly ritual where the entire team would sit in the same room and go through the process of combining each other’s changes line by line (Brady 2011; Papadimoulis 2006). It is not surprising then that some saw better facilities for reconciling of concurrently introduced changes as the key to better version control (Shuttleworth 2007b). Since merging was an unpleasant but important part of the programmer’s workflow, the goal for the next generation of VCSs was to make merges fast (so that programmers would not have to defer or work around them), reliable (to avoid situations where data could be lost or accidentally overwritten), and automated (to reduce the amount of work involved in reconciliation).

And indeed, many of the issues with merging described above have been mitigated by the latest generation of version control systems such as Git and Mercurial (Bokov 2010). For instance, since these newer systems do not rely on constant communication with a central server (instead maintaining their own local copies of all repository data) operations like switching between branches or comparing differences against previous versions are considerably faster. The designers of newer VCSs were also careful to ensure that merge points are explicitly retained, so that programmers do not have to keep track of which changes had already been merged (specifically, newer systems tend to structure their commit histories as directed acyclic graphs rather than trees, so that a merge point can be represented as a commit with multiple parents—one for each branch being merged).

But despite all of these advancements, at least one major problem remains—the question of what to do about conflicts.

2.8.2 CONFLICT

A conflict arises when two or more changes are made to the same piece of content, with no automatic way of deciding which of the changes should be retained. See for example Figure 10: in one branch, the last word in the first line is changed to “canoe”, and in another, the same word is
changed to “raft”. When the user tries to merge the changes from one branch into the other, an ambiguity arises, and the system is not able to automatically determine whether the merged version should use “canoe” or “raft”. This is known as a conflict, and its resolution must be left up to the user.

Figure 10 – Merge resulting in a conflict

Manually resolving conflicts can be difficult and disruptive. A conflict often expresses opposing intent on how the software is supposed to behave (that is, its requirements), or differing opinions about how it ought to achieve that behaviour (its implementation). Either way, resolution tends to require significant cognitive resources, and if multiple authors are implicated, significant social coordination (for example having to schedule a meeting, a risk of a quarrel if there is disagreement about whose changes should win, and so on). What is worse, since merging can be so problematic, programmers tend to put it off until absolutely necessary. The result is that conflicts tend to arise when programmers are least able to deal them—at times of stress (e.g. when code is being prepared for an imminent release), and outside the context in which the code to be resolved was originally written (the programmer must recall what the code in conflict was for, how it works, why it was written that particular way and not another, etc.).

* Although for the sake of simplicity in this example I describe the conflict in terms of words, a VCS would typically show the entire line as being in conflict. This means that any concurrent changes that touch the same lines will be considered conflicting, regardless of whether the changes within that line overlap. For example, if the word “row” had been replaced with “paddle” in one branch, and the word “boat” with “canoe” in the other, most VCSs would still consider this to be a conflict, despite the fact that to you and me it seems obvious that the changes do not overlap, at least not in terms of the which words were changed. (The reasons for why VCSs register changes to text exclusively at the line level are explained in more detail in section 2.4).
Given all of the associated problems, programmers tend to have an aversion to conflicts, preferring to avoid them if possible. For example, team members will often informally arrange not to work concurrently on the same files. Yet the advent of distributed version control systems has only increased the likelihood of conflicts (see the sidebar on p. 35 on the history of concurrency in VCS). Consequently, the designers of distributed VCSs put a great deal of effort into devising ways to automate conflict resolution.

But here too—perhaps more than anywhere else—version control systems run up against issues of identity, for instance in having to decide whether two different changes apply to the same part of a document (a conflict), or to two different parts (so that automatic resolution is possible). The next section looks at the various strategies for dealing with conflicts employed by the latest version control systems, arguing that all current strategies are ultimately deficient, and that that deficiency is underpinned by an overly-simplistic conception of identity.

2.8.3 RECONCILIATION

With better facilities for reconciliation of changes widely seen as a key feature for version control systems, a great deal of effort has been devoted to improving automatic merging. In general, the goal is to generate the fewest number of “false” conflicts (that is, conflicts that could have been resolved automatically without the user’s input), without ever failing to detect legitimate conflicts, as doing so would result in changes being silently lost or overridden. In other words, a merge strategy must strike a balance between ensuring that the user is notified of any potentially unintended changes, while avoiding any unnecessary interruption. (Shuttleworth 2007b).

To illustrate the problem, let’s start with a simple case: two versions of the same text, with all but one of the lines different, as shown below in Figure 11:

Not knowing the files’ histories—that is, not knowing which changes occurred in the left or right version relative to the original—there is no way to tell which of the conflicting lines should be retained.

In a version control system, each of these two versions would typically have a commit history, and given that a repository’s history can usually be traced back to a single commit (see section 2.3), it is should be possible to find the original version that both versions diverged from. Given a single
common ancestor, we can see which lines had been changed in which version and use this information to more accurately determine which changes are really in conflict and which can be merged automatically. This is known as a "3-way merge", and is illustrated in Figure 12.

Since the 3-way merge algorithm is able to see the individual changes introduced by each variant relative to its ancestor (second line on the left side, and the third line on the right), we can safely merge the non-conflicting lines without accidentally losing changes introduced in either of the two branches. Only the last line needs to be manually resolved, since it was changed concurrently on both sides. Note that any intermediate versions that might exist between the left and right variants and their common ancestor are irrelevant; the strategy compares only the contents of the three snapshots (left, right, and common ancestor), disregarding all else.

This 3-way merge algorithm is the basic merging mechanism employed by most version control systems. (Khanna, Kunal, and Pierce 2007) However, in recent years this strategy has come into question, in part because it has been shown to incorrectly handle certain so-called “pathological” merge cases (N. Smith 2005b; Cohen 2005a). Perhaps the best known of these is the so called “criss-cross merge”, where due to a pattern of reciprocal merges, two commits in separate branches end up with more than one common ancestor. Figure 13 (below) shows an example of a commit history with an archetypical criss-cross merge. Because the two branches pictured in the diagram merge into each other, commits $4a$ and $4b$ are both equally good candidates as common ancestors for $5a$ and $5b$ (and for any other subsequent commits).
When the common ancestor is ambiguous, the 3-way merge algorithm can sometimes fail to detect conflicts and silently lose data (that is, it can become “pathological”). Consider for example Figure 14 (based on Cohen 2005; Smith 2005; and Bosh 2011) demonstrating how a conflict can go unnoticed. 4a and 4b both modify the second line ("quietly down the stream" and “gently down the river”), so that a conflict arises when the two commits are merged in 5a and 5b. The user then decides to resolve the conflict in 5a by retaining the second line from 4a, and in 5b by retaining the entire line from 4b. Now, if another merge were attempted, combining 5a and 5b, the result would depend on whether 4a or 4b is selected as the common ancestor. However, since there is no particular reason to select one over the other—that is, both 4a and 4b are equally valid common ancestors—the merge becomes unpredictable (that is, two different users performing what appears to be the same merge operation can end up with different results, depending on which ancestor the system happens to select).
However, what makes the case truly “pathological” is that the 3-way merge algorithm will subsequently fail to correctly designate either of the latter two merges as conflicts. Consider for example the potential merge where 4a is chosen as the common ancestor. When only the two versions and their ancestor are taken into account, it appears that 5a has no changes and 5b has an altered second line; consequently the algorithm does what would otherwise be the right thing, and automatically chooses the second line from 5b. Now, imagine that the two branches actually belonged to two different authors. Note that in 5a, one author explicitly chose to retain the line “quietly down the stream” while in 5b the other author chose “gently down the river”. If either author were to merge their branch with the other’s, their choice of second line of the verse would be silently nullified.

There are a number of other such problematic scenarios, some that result in loss of data and many more that result in so-called “false conflicts” (that is, situations where the system alerts the user of a conflict even though no actual conflict exists) (Bosh 2011; Cohen 2005a; Ritcher 2011). The existence of these cases has prompted designers of version control systems to search for better alternatives.

For example, Monotone (one of the early third-generation VCSs), attempts to solve the criss-cross problem by selecting not the last common ancestor, but the last common unambiguous ancestor (in Figure 13, that would be commit 2). However, this strategy tends to generate a lot of confusing false conflicts (Bosh 2011), and in certain situations can become insidiously pathological (N. Smith 2005b; N. Smith 2005a).

Git implements a so called “recursive 3-way merge”, which automatically merges any ambiguous common ancestors, and then uses that merged version as the base for a standard 3-way merge (Kuivinen 2005b; Kuivinen 2005a). Although an improvement over the standard 3-way merge, the recursive 3-way merge can sometimes create confusing conflicts (especially if the automatic intermediate merge of the ambiguous common ancestors itself results in a conflict), it can occasionally generate non-deterministic merges (that is, the same merge can have unpredictably different results), and in rare situations it can also silently nullify changes (see “Recursive Three Way Merge” in Bosh 2011).

Other systems have opted to replace 3-way merge altogether, rather than try to improve on it. The best known such alternative is a class of algorithms called “weave merge”, whereby instead of using a single common ancestor, the merged output is weaved together line by line from an analysis of each line’s derived history (Bosh 2011). Although proponents claim that this strategy is better able to deal with certain pathological cases (Cohen 2005b; N. Smith 2005a), there are no formal proofs to show this is to be the case. Moreover, this style of merging has so far failed to gain traction, and as a result is poorly studied and sparsely documented.
In any case, there is reason to believe that weave-style strategies are unlikely to prove superior to 3-way merge. In an oft-cited exchange between Linus Torvalds, the original author of Git, and Bram Cohen, one of the chief proponents of weave-style merging, Torvalds (2005c) argued that despite a tendency in the VCS community to assume the contrary, the merge problem is not a matter of whether a given algorithm is able to handle all possible edge cases, but whether it is 1) simple and predictable, and 2) forms a solid base for subsequent commits. An algorithm that relies on a complicated analysis of a file's history fails on both counts; its complexity is apt to make it unpredictable, and the merge it generates only increases the complexity of future analysis (note that in 3-way merge, the result of the reconciliation can become a base for future merges, limiting how far back in the commit log a future merge operation needs to look, whereas in a weave merge, the new commit only increases the amount of data that must be evaluated). This same sentiment is echoed by Eric Raymond in the conclusion to his Understanding Version Control Systems (2009):

I suspect that, as our algorithms get better, we’re going to find that the best choices are not the most theoretically clever ones but rather the ones that are easiest for human beings to intuitively model.

This is why, even though I find constructions like today’s elaborate merge theory fascinating to think about, I’m not sure it is actually going anywhere useful. Naïve merge algorithms with poor behavior at edge cases may actually be preferable to more sophisticated ones that handle edge cases well but that humans have trouble anticipating effectively.

Perhaps implicit in this view is an understanding that the problem may ultimately be unsolvable—that a perfect merge algorithm that never fails to detect a true conflict and never displays a false one may be out of reach, and that our efforts are better spent towards making the system more efficient and more consistent.

What is rarely spoken about is why this limitation exists in the first place. What is it that holds us back from developing a merge algorithm that always gets things right? If what I have been driving at throughout this thesis is right, then the answer may have something to do with a deficiency in how version control systems construe identity.

Despite the pains developers of version control systems have gone to in order to find algorithms that can correctly detect conflicts, there is a class of examples that will consistently defeat any such algorithm. Consider for example the 3-way merge scenario shown below:
Since the changed lines on the left and right side do not overlap, a 3-way merge would reconcile the two without detecting any conflicts. Yet the result is obviously contradictory—the first and third line speak of a woeful, dead beaver, while the second and fourth are joyful and warn of an angry and thus presumably living beaver. The merge also introduces something like a grammatical error in the fourth line, since “but” no longer makes sense as a conjunction with the third line. This sort of semantic conflict is entirely inscrutable to even the most sophisticated merge algorithm, as detecting it would require a sensitivity to the subtleties of mood, and an ability to interpret and evaluate propositions—capabilities far beyond anything currently feasible in a version control system.

While the difficulty in detecting this sort of conflict certainly has to do with the computer's inability to derive meaning, a more general way to construe the problem may be in terms of a failure to adequately recognize the content’s identity.

At it’s most essential, a conflict in version control occurs when there are two or more concurrent changes to a single piece of content. As such, detecting a conflict is entirely predicated
on determining whether any two changes pertain to the same thing. But in this, version control systems are remarkably unsophisticated. All make the assessment based solely on “lines” (see section 2.4); that is, two changes are considered concurrent if they both modify the same line. A VCS would not assess the merge depicted in Figure 15 as conflicting because it individuates the text based solely on lines, and as such none of the changes are seen to affect the same content.

But if we expand the notion of whether two things can be referred to as one to something closer to the intuitive sense that we all use in everyday life, both changes in Figure 15 certainly do appear to operate on the same thing. It is simply that when assessing conflicts in regards to the content’s meaning, lines are an inadequate choice of reference. But, circling back to the point made earlier in section 0, the problem cannot be entirely alleviated by simply eschewing lines in favour of a bigger granules (say, paragraphs). The issue is that the appropriate unit of reference is impossible to establish up front—it depends entirely on the particularities of the content, and on the context in which it is evaluated.

This is a rather pessimistic conclusion, since there are few obvious paths for how one might facilitate this sort of dynamically-adjustable choice of reference for individuating content. But it is worth noting that despite the example given in Figure 15, in practice, a failure to detect a conflict using one individuation criteria rarely results in a conflict in any other such criteria. Note, for example that it is not at all trivial to devise an example of the sort of high-level conflict shown in in Figure 15 that does not also result in a line-level conflict. I encourage the reader, for instance, to try to break the rhyming pattern of the verse without also introducing a line-level conflict. This holds true too in source code, where, perhaps surprisingly (because problematic examples would not be difficult to come up with), clean merges that result in semantic conflicts are relatively rare. If there is anything to this seeming cohesiveness between different levels of granularity, future work might benefit from a better understanding of this relationship.

\* It may be worth pointing out that “same content” here refers to content at the same location, in an indexical (that is, deictic) sense. In other words, it is not that the same piece of content was concurrently changed (in the substantive sense), but rather that two or more changes occurred at the same location.
3. CONCLUSION

The story so far has not been a particularly happy one. I have argued that version control systems, and—maybe more worryingly— the infrastructure within which they are built, are at best dysfunctional, and at worst fundamentally broken. The core of the problem is that certain questions of identity for which there seem to be no straight answers—for instance, whether Theseus’ ship, torn apart and rebuilt anew, is still the same ship, or whether this Masters thesis, which started out with an entirely different premise, is still the same thesis—have turned out to be real, practical problems, in need of solutions.

The ability to refer to pieces of content through their many revisions in a precise, unambiguous manner is crucial in enabling the sort of collaboration version control systems aim to enable. Yet content—a problematic concept in itself (see section 2.1)—can be cut into pieces in a million different ways, many of which are incongruent, and few of which have a use that can be anticipated up front (section 2.4). But the system must record that content and its history one way or another, and in doing so it commits to one particular story at the cost of others. A history told in terms of lines, for example, loses sight of one told in terms of individual words, and a record that traces the continuity of single files can say little of a file split into two (section 2.6 and 0). Then there is the problem of meaning (or more precisely, of intentionality)—how content relates to the external world has direct bearing on how it ought to be parsed, yet that relationship is for the most part poorly understood by information science, and largely ignored in version control (section 2.5).

Each of these limitations, inherent not just in version control systems but in the present state of computing in general, is a potential source of trouble, frustrating users and limiting the sorts of things a version control system can do. But nowhere are these limitation more visible than in the problem of merge conflicts (section 2.8). A commitment to one kind of representation of content and its history over another bears on how a VCS can refer to that content, which in turn impacts its ability to support concurrency—that is, of safely allowing multiple simultaneous operations on the same objects. Ultimately, I argued that there seems to be no way to always get it right (section 2.8.3). A consequence of many of the fundamental commitments in version control is that some concurrent operations will inevitably be falsely recognized as conflicts, and some actual conflicts will go unrecognized.

But this is a rather negative note to go out on. So instead, I will end with a brief look at how version control systems might move forward, despite the considerable obstacles they face.
3.1 THE WAY FORWARD

First, it is worth noting that despite their problems, version control systems have proven wildly successful; both, in adoption—among software developers and increasingly in other areas—and in enabling new modes of collaboration. GitHub, in large part thanks to its innovative facilitation of version control, has fundamentally changed the way programmers work together, and has prompted explosive growth in collaboratively developed open source software. And it is not just that the latest crop of version control systems have overcome a number of difficult problems plaguing the previous generation of VCSs (problems of concurrency, speed, etc.), but that in doing so they created a space for new, largely unanticipated forms of collaboration. The novel concurrency model in systems like Git has made possible collaboration among loose teams of hundreds or even thousands of contributors—something not only infeasible in the past, but thought by some to be unthinkable or undesirable (after all, what good could come of such chaos?).

Second, much of the criticism I have leveled against version control systems has to do with their inability to bend themselves to users’ expectations. I argued that the way we make use of identity in everyday life is much more nuanced and flexible than what a version control system is able to support. But this line of criticism neglects to take note of the way people actually use tools. In practice, we adapt to our tools as much as we adapt them to ourselves. Software developers especially are accustomed to finding workarounds to rigid requirements. Coordinating with team members to avoid working on the same files, for example, is a simple but effective way of avoiding merge conflicts. Although that particular solution does not scale well for larger or more disparate teams, it is entirely probable that solutions to many of the problems discussed here will come not from improvements to version control systems, but to peripheral tools and practices.

In regards to version control systems themselves, what might the next generation be like? If there are any clues in the critique I offered, one is that there is promise in the sort of light-weight approach taken by Git. Minimizing the number of ontological commitments made up front may offer the best chance of avoiding many of the problems that arise from premature abstraction. By deferring as much as possible any subjective assertions—for example, of whether a piece of content was moved versus deleted and re-created—a version control system can avoid having to predict in advance all of the possible ways of slicing up of content and its history that might prove relevant down the line. Furthermore, since humans will in all likelihood continue to be better than computers at drawing conclusions about identity, the key to future progress may lie not in trying to devise smarter algorithms for identifying identity, but in developing better ways of presenting information about content and its history to users, so that they are better equipped to make those assertions for themselves.
Finally, as stated in the outset, the path to a better version control system is only one potential outcome for this project. Identifying and articulating the key obstacles is half the battle in itself. Programmers rarely have the luxury of stepping back and examining their tools and practices from the sort of distance afforded by this project. Taking on a conceptual perspective framed by the notions of identity, reference, meaning, and the like, if nothing else, provides an unorthodox look at version control, perhaps revealing something not otherwise easily seen. Likewise, that conceptual perspective can itself benefit from its direction at a decidedly practical problem. Those who tend to think about such things—philosophers, that is—rarely have the opportunity to apply their ideas to something so concrete. A goal in making the connection between philosophy and revision control is that it might benefit both sides—gathering ideas from a discipline that has been concerned with such things for thousands of years, while providing it with a sort of testing ground that until recently was not possible.

I lay no claim as to whether in any of this I have succeeded, but the attempt was certainly worthwhile.
4. BIBLIOGRAPHY


Montréal, Canada. doi:10.4242/BalisageVol5.Renear01.


