Darcs 2.1.0.1 (2.1.0.1 + 114 patches)

Darcs

David Roundy

April 22, 2009

Appendix A

Theory of patches

A.1 Background

I think a little background on the author is in order. I am a physicist, and think like a physicist. The proofs and theorems given here are what I would call "physicist" proofs and theorems, which is to say that while the proofs may not be rigorous, they are practical, and the theorems are intended to give physical insight. It would be great to have a mathematician work on this to give patch theory better formalized foundations.

From the beginning of this theory, which originated as the result of a series of email discussions with Tom Lord, I have looked at patches as being analogous to the operators of quantum mechanics. I include in this appendix footnotes explaining the theory of patches in terms of the theory of quantum mechanics. I advise against taking this analogy too seriously, although it could provide some insight into how a physicist might think about darcs.

A.2 Introduction

A patch describes a change to the tree. It could be either a primitive patch (such as a file add/remove, a directory rename, or a hunk replacement within a file), or a composite patch describing many such changes. Every patch type must satisfy the conditions described in this appendix. The theory of patches is independent of the data which the patches manipulate, which is what makes it both powerful and useful, as it provides a framework upon which one can build a revision control system in a sane manner.

Although in a sense, the defining property of any patch is that it can be applied to a certain tree, and thus make a certain change, this change does not wholly define the patch. A patch is defined by a *representation*, together with a set of rules for how it behaves (which it has in common with its patch type). The *representation* of a patch defines what change that particular patch makes, and must be defined in the context of a specific tree. The theory of patches

is a theory of the many ways one can change the representation of a patch to place it in the context of a different tree. The patch itself is not changed, since it describes a single change, which must be the same regardless of its representation[1].

So how does one define a tree, or the context of a patch? The simplest way to define a tree is as the result of a series of patches applied to the empty tree[2]. Thus, the context of a patch consists of the set of patches that precede it.

A.3 Applying patches

A.3.1 Hunk patches

Hunks are an example of a complex filepatch. A hunk is a set of lines of a text file to be replaced by a different set of lines. Either of these sets may be empty, which would mean a deletion or insertion of lines.

A.3.2 Token replace patches

Although most filepatches will be hunks, darcs is clever enough to support other types of changes as well. A "token replace" patch replaces all instances of a given token with some other version. A token, here, is defined by a regular expression, which must be of the simple [a–z...] type, indicating which characters are allowed in a token, with all other characters acting as delimiters. For example, a C identifier would be a token with the flag `[A-Za-z_0-9]`.

What makes the token replace patch special is the fact that a token replace can be merged with almost any ordinary hunk, giving exactly what you would want. For example, you might want to change the patch type `TokReplace` to `TokenReplace` (if you decided that saving two characters of space was stupid). If you did this using hunks, it would modify every line where `TokReplace` occurred, and quite likely provoke a conflict with another patch modifying those lines. On the other hand, if you did this using a token replace patch, the only change that it could conflict with would be if someone else had used the token "`TokenReplace`" in their patch rather than TokReplace—and that actually would be a real conflict!

[1] For those comfortable with quantum mechanics, think of a patch as a quantum mechanical operator, and the representation as the basis set. The analogy breaks down pretty quickly, however, since an operator could be described in any complete basis set, while a patch modifying the file `foo` can only be described in the rather small set of contexts which have a file `foo` to be modified.

[2] This is very similar to the second-quantized picture, in which any state is seen as the result of a number of creation operators acting on the vacuum, and provides a similar set of simplifications—in particular, the exclusion principle is very elegantly enforced by the properties of the anti-hermitian fermion creation operators.

A.4 Patch relationships

The simplest relationship between two patches is that of "sequential" patches, which means that the context of the second patch (the one on the left) consists of the first patch (on the right) plus the context of the first patch. The composition of two patches (which is also a patch) refers to the patch which is formed by first applying one and then the other. The composition of two patches, P_1 and P_2 is represented as P_2P_1, where P_1 is to be applied first, then P_2[3]

There is one other very useful relationship that two patches can have, which is to be parallel patches, which means that the two patches have an identical context (i.e. their representation applies to identical trees). This is represented by $P_1 \parallel P_2$. Of course, two patches may also have no simple relationship to one another. In that case, if you want to do something with them, you'll have to manipulate them with respect to other patches until they are either in sequence or in parallel.

The most fundamental and simple property of patches is that they must be invertible. The inverse of a patch is described by: P^{-1}. In the darcs implementation, the inverse is required to be computable from knowledge of the patch only, without knowledge of its context, but that (although convenient) is not required by the theory of patches.

Definition 1 *The inverse of patch P is P^{-1}, which is the "simplest" patch for which the composition $P^{-1}P$ makes no changes to the tree.*

Using this definition, it is trivial to prove the following theorem relating to the inverse of a composition of two patches.

Theorem 1 *The inverse of the composition of two patches is*

$$(P_2P_1)^{-1} = P_1^{-1}P_2^{-1}.$$

Moreover, it is possible to show that the right inverse of a patch is equal to its left inverse. In this respect, patches continue to be analogous to square matrices, and indeed the proofs relating to these properties of the inverse are entirely analogous to the proofs in the case of matrix multiplication. The compositions proofs can also readily be extended to the composition of more than two patches.

A.5 Commuting patches

A.5.1 Composite patches

Composite patches are made up of a series of patches intended to be applied sequentially. They are represented by a list of patches, with the first patch in the list being applied first.

[3]This notation is inspired by the notation of matrix multiplication or the application of operators upon a Hilbert space. In the algebra of patches, there is multiplication (i.e. composition), which is associative but not commutative, but no addition or subtraction.

The first way (of only two) to change the context of a patch is by commutation, which is the process of changing the order of two sequential patches.

Definition 2 *The commutation of patches P_1 and P_2 is represented by*

$$P_2 P_1 \longleftrightarrow P_1{}' P_2{}'.$$

Here P_1' is intended to describe the same change as P_1, with the only difference being that P_1' is applied after P_2' rather than before P_2.

The above definition is obviously rather vague, the reason being that what is the "same change" has not been defined, and we simply assume (and hope) that the code's view of what is the "same change" will match those of its human users. The '$\longleftrightarrow$' operator should be read as something like the == operator in C, indicating that the right hand side performs identical changes to the left hand side, but the two patches are in reversed order. When read in this manner, it is clear that commutation must be a reversible process, and indeed this means that commutation *can* fail, and must fail in certain cases. For example, the creation and deletion of the same file cannot be commuted. When two patches fail to commutex, it is said that the second patch depends on the first, meaning that it must have the first patch in its context (remembering that the context of a patch is a set of patches, which is how we represent a tree). [4]

Merge The second way one can change the context of a patch is by a **merge** operation. A merge is an operation that takes two parallel patches and gives a pair of sequential patches. The merge operation is represented by the arrow "$\Longrightarrow$".

Definition 3 *The result of a merge of two patches, P_1 and P_2 is one of two patches, P_1' and P_2', which satisfy the relationship:*

$$P_2 \parallel P_1 \Longrightarrow P_2{}' P_1 \longleftrightarrow P_1{}' P_2.$$

Note that the sequential patches resulting from a merge are *required* to commutex. This is an important consideration, as without it most of the manipulations we would like to perform would not be possible. The other important fact is that a merge *cannot fail*. Naively, those two requirements seem contradictory. In reality, what it means is that the result of a merge may be a patch which is much more complex than any we have yet considered[5].

[4] The fact that commutation can fail makes a huge difference in the whole patch formalism. It may be possible to create a formalism in which commutation always succeeds, with the result of what would otherwise be a commutation that fails being something like a virtual particle (which can violate conservation of energy), and it may be that such a formalism would allow strict mathematical proofs (whereas those used in the current formalism are mostly only hand waving "physicist" proofs). However, I'm not sure how you'd deal with a request to delete a file that has not yet been created, for example. Obviously you'd need to create some kind of antifile, which would annihilate with the file when that file finally got created, but I'm not entirely sure how I'd go about doing this. ∴ So I'm sticking with my hand waving formalism.

[5] Alas, I don't know how to prove that the two constraints even *can* be satisfied. The best I have been able to do is to believe that they can be satisfied, and to be unable to find an case in which my implementation fails to satisfy them. These two requirements are the foundation of the entire theory of patches (have you been counting how many foundations it has?).

A.5.2 How merges are actually performed

The constraint that any two compatible patches (patches which can successfully be applied to the same tree) can be merged is actually quite difficult to apply. The above merge constraints also imply that the result of a series of merges must be independent of the order of the merges. So I'm putting a whole section here for the interested to see what algorithms I use to actually perform the merges (as this is pretty close to being the most difficult part of the code).

The first case is that in which the two merges don't actually conflict, but don't trivially merge either (e.g. hunk patches on the same file, where the line number has to be shifted as they are merged). This kind of merge can actually be very elegantly dealt with using only commutation and inversion.

There is a handy little theorem which is immensely useful when trying to merge two patches.

Theorem 2 *$P_2' P_1 \longleftrightarrow P_1' P_2$ if and only if $P_1'^{-1} P_2' \longleftrightarrow P_2 P_1^{-1}$, provided both commutations succeed. If either commutex fails, this theorem does not apply.*

This can easily be proven by multiplying both sides of the first commutation by $P_1'^{-1}$ on the left, and by P_1^{-1} on the right. Besides being used in merging, this theorem is also useful in the recursive commutations of mergers. From Theorem 2, we see that the merge of P_1 and P_2' is simply the commutation of P_2 with P_1^{-1} (making sure to do the commutation the right way). Of course, if this commutation fails, the patches conflict. Moreover, one must check that the merged result actually commutes with P_1, as the theorem applies only when *both* commutations are successful.

Of course, there are patches that actually conflict, meaning a merge where the two patches truly cannot both be applied (e.g. trying to create a file and a directory with the same name). We deal with this case by creating a special kind of patch to support the merge, which we will call a "merger". Basically, a merger is a patch that contains the two patches that conflicted, and instructs darcs basically to resolve the conflict. By construction a merger will satisfy the commutation property (see Definition 3) that characterizes all merges. Moreover the merger's properties are what makes the order of merges unimportant (which is a rather critical property for darcs as a whole).

The job of a merger is basically to undo the two conflicting patches, and then apply some sort of a "resolution" of the two instead. In the case of two conflicting hunks, this will look much like what CVS does, where it inserts both versions into the file. In general, of course, the two conflicting patches may both be mergers themselves, in which case the situation is considerably more complicated.

Much of the merger code depends on a routine which recreates from a single merger the entire sequence of patches which led up to that merger (this is, of course, assuming that this is the complicated general case of a merger of mergers of mergers). This "unwind" procedure is rather complicated, but absolutely critical to the merger code, as without it we wouldn't even be able to undo

the effects of the patches involved in the merger, since we wouldn't know what patches were all involved in it.

Basically, unwind takes a merger such as

```
M( M(A,B), M(A,M(C,D)))
```

From which it recreates a merge history:

```
C
A
M(A,B)
M( M(A,B), M(A,M(C,D)))
```

(For the curious, yes I can easily unwind this merger in my head [and on paper can unwind insanely more complex mergers]—that's what comes of working for a few months on an algorithm.) Let's start with a simple unwinding. The merger `M(A,B)` simply means that two patches (`A` and `B`) conflicted, and of the two of them `A` is first in the history. The last two patches in the unwinding of any merger are always just this easy. So this unwinds to:

```
A
M(A,B)
```

What about a merger of mergers? How about `M(A,M(C,D))`. In this case we know the two most recent patches are:

```
A
M(A,M(C,D))
```

But obviously the unwinding isn't complete, since we don't yet see where `C` and `D` came from. In this case we take the unwinding of `M(C,D)` and drop its latest patch (which is `M(C,D)` itself) and place that at the beginning of our patch train:

```
C
A
M(A,M(C,D))
```

As we look at `M( M(A,B), M(A,M(C,D)))`, we consider the unwindings of each of its subpatches:

```
          C
A         A
M(A,B)    M(A,M(C,D))
```

As we did with `M(A,M(C,D))`, we'll drop the first patch on the right and insert the first patch on the left. That moves us up to the two `A`'s. Since these agree, we can use just one of them (they "should" agree). That leaves us with the `C` which goes first.

The catch is that things don't always turn out this easily. There is no guarantee that the two `A`'s would come out at the same time, and if they didn't,

we'd have to rearrange things until they did. Or if there was no way to rearrange things so that they would agree, we have to go on to plan B, which I will explain now.

Consider the case of `M( M(A,B), M(C,D))`. We can easily unwind the two subpatches

```
A          C
M(A,B)     M(C,D)
```

Now we need to reconcile the `A` and `C`. How do we do this? Well, as usual, the solution is to use the most wonderful Theorem 2. In this case we have to use it in the reverse of how we used it when merging, since we know that `A` and `C` could either one be the *last* patch applied before `M(A,B)` or `M(C,D)`. So we can find `C'` using

$$A^{-1}C \longleftrightarrow C'A'^{-1}$$

Giving an unwinding of

```
C'
A
M(A,B)
M( M(A,B), M(C,D) )
```

There is a bit more complexity to the unwinding process (mostly having to do with cases where you have deeper nesting), but I think the general principles that are followed are pretty much included in the above discussion.

A.6 Conflicts

There are a couple of simple constraints on the routine which determines how to resolve two conflicting patches (which is called 'glump'). These must be satisfied in order that the result of a series of merges is always independent of their order. Firstly, the output of glump cannot change when the order of the two conflicting patches is switched. If it did, then commuting the merger could change the resulting patch, which would be bad. Secondly, the result of the merge of three (or more) conflicting patches cannot depend on the order in which the merges are performed.

The conflict resolution code (glump) begins by "unravelling" the merger into a set of sequences of patches. Each sequence of patches corresponds to one non-conflicted patch that got merged together with the others. The result of the unravelling of a series of merges must obviously be independent of the order in which those merges are performed. This unravelling code (which uses the unwind code mentioned above) uses probably the second most complicated algorithm. Fortunately, if we can successfully unravel the merger, almost any function of the unravelled merger satisfies the two constraints mentioned above that the conflict resolution code must satisfy.

A.7 Patch string formatting

Of course, in order to store our patches in a file, we'll have to save them as some sort of strings. The convention is that each patch string will end with a newline, but on parsing we skip any amount of whitespace between patches.

Merger patches Merge two patches. The MERGERVERSION is included to allow some degree of backwards compatibility if the merger algorithm needs to be changed.

```
merger MERGERVERSION
<first patch>
<second patch>
```

Named patches Named patches are displayed as a "patch id" which is in square brackets, followed by a patch. Optionally, after the patch id (but before the patch itself) can come a list of dependencies surrounded by angle brackets. Each dependency consists of a patch id.