Darcs Patch Theory

by Jason Dagit ⟨dagit@codersbase.com⟩

Darcs is a free, open source source code revision control system implemented in Haskell. One of its distinguishing features is the theory of ***patches****, that provides the theoretical foundations on which the tool is built. In this article, I will try introduces the basic ideas and operations of patch theory. We setup the conflict problem and we hope to explain the solution in later installments of* The Monad.Reader*.*

Basic Elements of Patch Theory

The goal of patch theory is to describe changes to a repository. Although patch theory could be treated purely as a mathematical abstraction, we focus on the details that relate to Darcs [1]. Please pay close attention to the relationship between **changes** and **patches.**

Definition 1. A **repository** is a set of changes.

Here we specifically mean a mathematical set; we mean to emphasize that the order of the elements is unimportant. This implies that merging two repositories should simply be the union of those sets.

Definition 2. In the abstract sense, a **change** is just an element of a repository. In Darcs, each change modifies the working copy or other changes.

Two changes are the same if they have the same effect, but it is important to notice that we cannot compare two changes unless they modify the same thing, which is to say they originate from identical states.

Definition 3. A **context** is the state after some number of changes have been added.

In order to manipulate changes, Darcs stores each change using a patch.

Definition 4. A **patch** is a fixed and concrete representation of a change. All discussions of patches in this document assume the representation used by Darcs, unless noted otherwise.

Each patch has two contexts associated with it. There is the context that the patch originates from and the context resulting from the patch. Given a set of patches we can only form sequences from the patches if their contexts match up. Additionally, we can only apply the changes represented by a patch if we have our data in the context required by the patch. Initially, these two requirements seem to go against the definition of a repository.

We want our repositories to store unordered sets of changes, but we have just stated that given a set of patches (which represent our changes) that there is an implicit order we must adhere to in order to apply the patches.

The Famous Commute

One way to get around the implicit ordering constraint described above is to create new representations of each change (i.e., new patches) so that they are in a different order.

Before we explain commute, here is an introduction to our notation. In the following example we use capital letters to represent patches and lower cased super scripts are used to mark the contexts.

To denote the patch A from context o to a, we would write ${}^{o}A^{a}$. A repository might contain two patches, ${}^{o}A^{a}$ and ${}^{a}B^{b}$, in which case we could put them in a sequence and simply write, ${}^{o}A^{a}B^{b}$. Note that since the ending context of A matches the starting context of B we only write the 'a' once. Often when the contexts are understood, they are omitted and just the patches are named.

Suppose we had two changes, C_1 and C_2 such that ${}^{o}A_1^{a}$ and ${}^{e}A_2^{c}$ both represent C_1 and, similarly, both ${}^{o}B_1^{e}$ and ${}^{a}B_2^{d}$ represent C_2. We could then write two different sequences of patches for our two changes. We could write,

$$ {}^{o}A_1^{a}B_2^{d} $$

or,

$$ {}^{o}B_1^{e}A_2^{c}. $$

Since both sequences represent having both changes their final contexts are actually the same and so we have

$$ c = d. $$

Patch theory defines a swap operation between two patches with suitable contexts called commute. The purpose of commute is to take two patches, which can

be placed in sequence, and create different representations of each patch so that the sequence order is reversed. Using the example above, it is possible to switch between the two sequences by commuting the two patches.

Not all patches can have their order reversed by the commute operation. For example, suppose one change creates a file and another change modifies the contents of that file. Since you cannot modify the contents of a file that hasn't been created, all patches representing modifications to the file must come after the patch that creates the file. When commute fails for this reason, we say that one patch depends on the other.

There are also several properties that each patch type must satisfy, but not all are listed here. For example, we must be able to invert each patch, or undo it's change. The inverse of patch B is denoted, $\overline{B}$.

Additionally, there are three properties which combined give an elegant way to merge two patches, but these properties are beyond the scope of this article. Instead of listing the properties, we give a simple example of how merge works.

Suppose we have the patches

$$^{o}A^{a} \text{ and } ^{o}B^{b}$$

and we would like to merge them. This requires us to find patch representations which can be put in sequence. Suppose we apply the inverse of A to get

$$^{o}A^{a}\overline{A}^{o}.$$

By the (unlisted) properties of the commute function we know that when commute succeeds we can commute

$$^{a}\overline{A}^{o}B^{b}$$

to get

$$^{a}B_1^{c}\overline{A}_1^{b}.$$

So now we can write the following sequence of patches

$$^{o}A^{a}B_1^{c}\overline{A}_1^{b}.$$

We can always remove patches from the end of a sequence by just throwing it away. We can remove patch from the right end to get the merged sequence,

$$^{o}A^{a}B_1^{c}.$$

The reader is encourage to check the Wikibook [2] for a clear, through and easy-to-read explanation of the merge algorithm. The reader should also verify that the patches can be merged in the other order, e.g. starting with patch B.

The Infamous Conflict

In the previous section, during the merge we assumed that the commutes always succeeded. This is not always the case. When we are performing a merge and two patches fail to commute, we say that the patches conflict with each other. This could happen if the two patches try to modify the same line of a file. So far we have two different times when commute can fail and both times means something different: in the context of a merge, it means the patches are conflicting; but when the patches are already in a sequence and fail to commute, it means there is a dependency between the patches. This difference is subtle but significant.

The conflict problem is how to retain the property that repositories are an unordered set of changes, but merging these sets can fail to produce a set of patches that can be ordered sequentially (i.e., merged).

There seems to be two main ways of solving the problem. One way is to hold all of the patches in a sequence using special patches to note the places where the sequence forks (current darcs does this with mergers and conflictors but the design is flawed). The other approach is to store all the patches in the repository and to "disable" some patches until the conflicts go away. The latter approach is easier conceptually, but requires recomputing the sequence of patches more frequently. For the Summer of Code we chose to implement the latter approach.

The conflict problem is worsened by scenarios where the user wants a patch that depends on a "disabled" patch. This requires Darcs to store all patches it has seen and have a way to determine when a new patch represents a change that has already been "disabled" (possibly by different patch representing the same change). This requirement led us to implement a "remerge" algorithm which is capable of taking a sequence of disabled patches and enabling as many of the patches as possible. This way it is possible to merge two repositories by disabling any possibly conflicting patches and then re-enabling some of them. We also added new patch types to allow the user to specify which changes should be active and which ones should be disabled.

References

[1] `http://darcs.net/unstable`.

[2] Wikibooks. Understanding darcs/patch theory. `http://en.wikibooks.org/wiki/Understanding_darcs/Patch_theory`.