Darcs Patch Theory
(more or less)

Ian Lynagh

September 7, 2008

1 Overview

In this document we present a description of a darcs-like patch theory. We do not describe exactly the theory that darcs 2 is built on, as it is known to have some problems, but our theory is similar to darcs 2's. We hope that darcs 3 will be based on it.

In Section 2 we quickly introduce some general notation that we will be using.

In Section 3 we say what we require "patches" to satisfy. The behaviour of patches are specified as axioms, which we will build on top of.

In Section 4 we add unique names to our patches.

In Section 5 we define patch sequences, and some properties that they satisfy.

In Section 6 we take a break from the formal definition, and give a description of how merges work (when conflicts are not involved).

In Section 7 we introduce a "contexted patch" datastructure.

In Section 8 we define catches and repos, and make conjectures that they satisfy properties that we want them to satisfy.

2 Notation

We write $\forall a \in A, b \in B \cdot p\ a\ b$ to mean "for all a in A and b in B, $p\ a\ b$ holds".

We will be defining some relations which relate pairs and singles of values. In order to be clear about what the relation is acting on, we will write the single value v as $\langle v \rangle$, and the pair of values v and w as $\langle v, w \rangle$.

There are certain letters that we use to represent certain types of thing. These things may not be familiar to you yet, but you can refer back to this section as you need to later on. They are:

Patches	p, q, r, s, t, u, v
Contexted patches	w, x, y, z
Catches	c, d, e, f, g

If we are using a to represent a thing, then we will use $\bar{a}$ to represent a sequence of things, and A to represent a set of things.

We use subsections of *italic text* when giving intuition about what the formal description means.

3 Patches

We start by introducing *patches*, and the concept of *patch commutation*.

Definition 3.1 (patches)
We have a (possibly infinite) set of patches $\mathbf{P}$.

Definition 3.2 (patch-commute)
$\forall p \in \mathbf{P}, q \in \mathbf{P} \cdot$
$\quad (\exists p' \in \mathbf{P}, q' \in \mathbf{P} \cdot \langle p, q \rangle \longleftrightarrow \langle q', p' \rangle)$
$\vee \ \langle p, q \rangle \longleftrightarrow \text{fail}$

Explanation
We write $\langle p, q \rangle \longleftrightarrow \langle q', p' \rangle$ if p and q commute, resulting in q' and p'. In other words, doing p and then q is equivalent to doing q' and then p', where p and p' are morally equivalent, and likewise q and q' are morally equivalent.

For example, adding "Hello" to line 3 of a file and then adding "World" to line 5 of the file is equivalent to adding "World" to line 4 of the file and then adding "Hello" to line 3 of a file. We therefore say that $\langle$add "Hello" 3, add "World" 5$\rangle \longleftrightarrow \langle$add "World" 4, add "Hello" 3$\rangle$.

We can represent this diagramatically thus:

$$\begin{tikzcd}
 & \bullet \arrow[dr, bend left, "q"] & \\
\bullet \arrow[ur, bend left, "p"] \arrow[dr, bend right, "q'"'] & & \bullet \\
 & \bullet \arrow[ur, bend right, "p'"'] &
\end{tikzcd}$$

Here the dots are repository states, and the arrows are patches; pq and $q'p'$ get us from the same state, to the same state, but via different intermediate states.

We write $\langle p, q \rangle \longleftrightarrow \text{fail}$ if p and q do not commute. In other words, it is not possible to do the moral equivalent of q before p.

For example, $\langle$replace line 3 with "Hello", replace line 3 with "World"$\rangle \longleftrightarrow \text{fail}$

Axiom 3.1 (patch-commute-unique)
$\forall p \in \mathbf{P}, q \in \mathbf{P}, r \in (\mathbf{P} \times \mathbf{P}) \cup \{\text{fail}\}, s \in (\mathbf{P} \times \mathbf{P}) \cup \{\text{fail}\} \cdot$
$(\langle p, q \rangle \longleftrightarrow r) \wedge (\langle p, q \rangle \longleftrightarrow s) \Rightarrow r = s$

Explanation
Either p and q always *commute, or they* never *commute. If they do commute, then the result is always the same.*

Now that we know that the result of a commute is unique, we can afford to be a bit lax about quantifying over variables. For example, if we are dealing with two patches p and q and we say that $\langle p, q \rangle \longleftrightarrow \langle q', p' \rangle$ then it is implied that we are considering the case where p and q commute, and that the result of the commutation is q' and p'.

Axiom 3.2 (patch-commute-self-inverse)
$\forall p \in \mathbf{P}, q \in \mathbf{P}, p' \in \mathbf{P}, q' \in \mathbf{P} \cdot$
$(\langle p, q \rangle \longleftrightarrow \langle q', p' \rangle) \Leftrightarrow (\langle q', p' \rangle \longleftrightarrow \langle p, q \rangle)$

Explanation
If you commute a pair of patches, and then commute them back again, then you end up back where you started. Note also that we require that if commuting one way succeeds then commuting the other way also succeeds.

4 Named patches

From this point on, all patches will have unique names. For the motivation for this decision, see Appendix A.

Explanation
Just to clarify what we mean by "all patches will have unique names": When a patch is created, it is given a globally unique name. However, if we copy a patch from one place to another, or commute a pair of patches, then the names remain unchanged.

Definition 4.1 (patch-name)
We define n to tell us the name of a patch, i.e. $n(p)$ is the name of the patch p.

Definition 4.2 (names-invertible)
Names have an inverse.

Definition 4.3 (names-signed)
We say that a name is either positive or negative.

Explanation
Here's some intuition for what this means: When you record a patch in darcs, the patch name is positive (think: it adds something to the repo). If you roll back that patch, then darcs makes the corresponding negative patch (which removes something from the repo).

Axiom 4.1 (commute-preserves-names)
$(\langle p, q\rangle \longleftrightarrow \langle q', p'\rangle) \Rightarrow (n(p) = n(p') \wedge n(q) = n(q'))$

Explanation
As we said earlier, commuting a patch does not change its name.

Axiom 4.2 (patch-name-inverse-sign)
If $n(p)$ is positive then $n\left(p^{-1}\right)$ is negative. If $n(p)$ is negative then $n\left(p^{-1}\right)$ is positive.

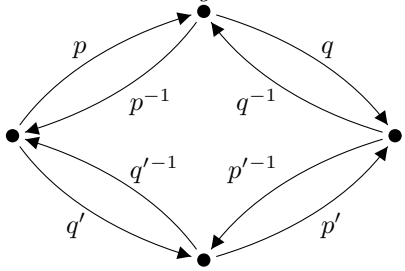
Explanation
The inverse of a normal patch is a rollback, and vice-versa.

Axiom 4.3 (patch-inverse-exists)
$\forall p \in \mathbf{P} \cdot \exists p^{-1} \in \mathbf{P} \cdot n(p) = n\left(p^{-1}\right)$

Explanation
Every patch has an inverse.

Axiom 4.4 (patch-commute-square)
$\forall p \in \mathbf{P}, q \in \mathbf{P}, \forall p' \in \mathbf{P}, q' \in \mathbf{P} \cdot$
$(\langle p, q\rangle \longleftrightarrow \langle q', p'\rangle) \Leftrightarrow (\langle q'^{-1}, p\rangle \longleftrightarrow \langle p', q^{-1}\rangle)$

Explanation
Axiom 4.3 tells us that all patches have an inverse. Therefore we can augment our patch commutation diagram with inverse patches:



This axiom tells us that we can rotate the diagram 90 degrees clockwise and we again have a valid commutation diagram. The arrows from left to right along the top are $q'^{-1}p$ and along the bottom are $p'q^{-1}$, thus $\langle q'^{-1}, p\rangle \longleftrightarrow \langle p', q^{-1}\rangle$.

In actual fact, by applying this axiom twice or three times, we can see that all four of the rotations are equivalent, and thus $\langle p'^{-1}, q'^{-1}\rangle \longleftrightarrow \langle q^{-1}, p^{-1}\rangle$ and $\langle q, p'^{-1}\rangle \longleftrightarrow \langle p^{-1}, q'\rangle$.

5 Patch Sequences

Definition 5.1 (patch-sequences)
We write the empty sequence of patches as ϵ.

We compose patches with juxtaposition.

Definition 5.2 (patch-names)
We define N to tell us the names in a sequence of patches, i.e.
$N(\epsilon) = \emptyset$
$N(p\bar{q}) = n(p) \cup N(\bar{q})$

Definition 5.3 (context)
A context is a set of names.

Explanation
When making patch diagrams, we said that the arrows are patches, but glossed over what the dots are. In fact, these dots are contexts. Recall our commute diagram, now explicitly annotated with contexts:

$$\begin{tikzcd}
 & B \cup \{n(p)\} \arrow[dr, bend left, "q"] & \\
B \arrow[ur, bend left, "p"] \arrow[dr, bend right, "q'"'] & & B \cup \{n(p), n(q)\} \\
 & B \cup \{n(q)\} \arrow[ur, bend right, "p'"'] &
\end{tikzcd}$$

Here we start in a context B and add names to the context as we apply patches.

Note that adding a name and its inverse to a context does not get us back to where we started, i.e. if a is a name and B is a context (where $a \notin B$ and $a^{-1} \notin B$) then $B \neq B \cup \{a, a^{-1}\}$. In actual fact we are a little more lax in our diagrams, as we have tended to represent B and $B \cup \{a, a^{-1}\}$ with the same dot.

Definition 5.4 (patch-sequence-commute)
We extend $\longleftrightarrow$ to work with combinations of patches and patch sequences in the natural way:
$\langle p, q\bar{r}\rangle \longleftrightarrow \langle q'\bar{r}', p'\rangle$
  if $\langle p, q\rangle \longleftrightarrow \langle q', p''\rangle$
    $\langle p'', \bar{r}\rangle \longleftrightarrow \langle \bar{r}', p'\rangle$
$\langle \bar{p}q, r\rangle \longleftrightarrow \langle r', \bar{p}'q'\rangle$
  if $\langle q, r\rangle \longleftrightarrow \langle r'', q'\rangle$
    $\langle \bar{p}, r''\rangle \longleftrightarrow \langle r', \bar{p}'\rangle$
$\langle \bar{p}q, r\bar{s}\rangle \longleftrightarrow \langle r''\bar{s}'', \bar{p}''q''\rangle$
  if $\langle q, r\rangle \longleftrightarrow \langle r', q'\rangle$
    $\langle \bar{p}, r'\rangle \longleftrightarrow \langle r'', \bar{p}'\rangle$
    $\langle q', \bar{s}\rangle \longleftrightarrow \langle \bar{s}', q''\rangle$
    $\langle \bar{p}', \bar{s}'\rangle \longleftrightarrow \langle \bar{s}'', \bar{p}''\rangle$

In all cases, if the above rules don't apply the commute fails.

Definition 5.5 (patch-sequences)
We classify some patch sequences as *sensible*.

Explanation
For example, the sequence 'add file foo; insert "hello world" into foo' is sensible, whereas 'insert "hello world" into foo' is not sensible, as it doesn't make sense to insert text into a file without first adding the file.

Axiom 5.1 (doing-nothing-is-sensible)
ϵ is sensible.

Explanation
If you haven't done anything then you certainly haven't done anything that isn't sensible.

Axiom 5.2 (sensible-subsequences)
If $\bar{p}q$ is sensible then $\bar{p}$ is sensible.

Explanation
If you are doing sensible things, then you can stop at any time, and the rseult will be sensible.

Axiom 5.3 (sensible-prefix-commute)
If $\bar{p}r$ is sensible, $\bar{q}$ is sensible and $N(\bar{p}) = N(\bar{q})$ then $\bar{q}r$ is sensible.

Explanation
It doesn't matter how you get to a context (as long as it is via a sensible route), what you can sensibly do from that context is always the same.

Axiom 5.4 (commute-preserves-sensibility)
If $\bar{p}qr$ is sensible and $\langle q, r\rangle \longleftrightarrow \langle r', q'\rangle$ then $\bar{p}r'q'$ is sensible.

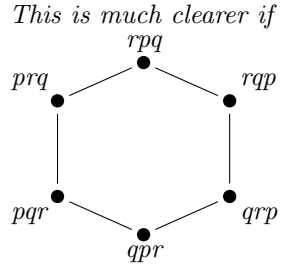
Explanation
Successful commuting of patches doesn't affect whether a sequence is sensible.

Axiom 5.5 (sensible-inverse)
If $\bar{p}\bar{q}$ is sensible and all the patches in $\bar{q}$ are positive then $\bar{p}\bar{q}\bar{q}^{-1}$ is sensible.

Explanation
Undoing patches takes us back to an earlier sensible state.

Axiom 5.6 (patch-commute-associates)
$$\forall p \in \mathbf{P}, q \in \mathbf{P}, r \in \mathbf{P}, p_q \in \mathbf{P}, p_r \in \mathbf{P}, p_{qr} \in \mathbf{P}, q_p \in \mathbf{P}, q_r \in \mathbf{P}, q_{pr} \in \mathbf{P}, r_p \in \mathbf{P}, r_q \in \mathbf{P}, r_{pq} \in \mathbf{P} \cdot$$
$$(\langle q, r\rangle \longleftrightarrow \langle r_q, q_r\rangle) \wedge (\langle p, q\rangle \longleftrightarrow \langle q_p, p_q\rangle) \wedge (\langle p_q, r\rangle \longleftrightarrow \langle r_p, p_{qr}\rangle)$$
$$\Leftrightarrow (\langle q_p, r_p\rangle \longleftrightarrow \langle r_{pq}, q_{pr}\rangle) \wedge (\langle q_{pr}, p_{qr}\rangle \longleftrightarrow \langle p_r, q_r\rangle) \wedge (\langle r_{pq}, p_r\rangle \longleftrightarrow \langle p, r_q\rangle)$$

Explanation
This is much clearer if we consider it in diagramatic form:



This is not like most of our diagrams; here the dots all represent the same repository state, and we move from one to the other by commuting the order of the patches.

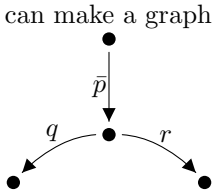
We start off with the patch sequence pqr. This axiom says that if q and r commute (i.e. you can move from the bottom left to the top left) and p can be commuted past qr (you can move from bottom left to bottom right), then q and r still commute after p has been commuted past them (you can move from the bottom right to the top right), and you can commute p back again (move from the top right to the top left).

Sequences will be considered equal if they are equal up to commutation; in particular, when we talk about something like $\bar{p}\bar{q}$ we mean "a sequence $\bar{r}$ which, after some number of commutations, is equal to $\bar{p}\bar{q}$".

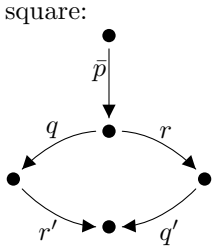
6 Patch Merge

In order to give you some idea of where we are going, in this section we will describe how merging of non-conflicting patches works. This whole section is only informational.

To start with, let's see how to merge single patches. Suppose we have the two repos $\bar{p}q$ and $\bar{p}r$, where $n(q) \neq n(r)$, i.e. they differ only in the last patch. We wish to merge these two repos. We can make a graph containing the patches in both repos thus:

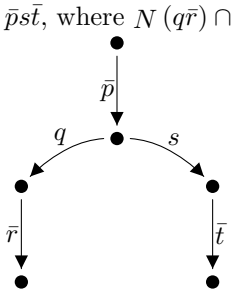


In order to merge the two repos, we want a single sequence of patches incorporating the effects of both q and r. To do this, we can complete the bottom of our diagram to form a commutation square:

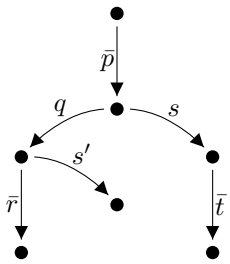


Provided the commute succeeds, we have that the result of the merge is qr' (or, equivalently, rq'). If the commute does not succeed then the patches cannot be cleanly merged, i.e. q and r conflict.

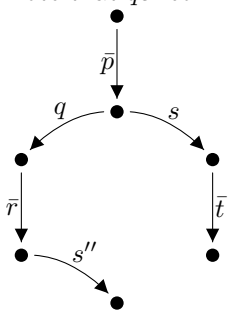
Now let's generalise to arbitrarily large merges. In one repo we have $\bar{p}q\bar{r}$, and in the other we have $\bar{p}s\bar{t}$, where $N(q\bar{r}) \cap N(s\bar{t}) = \emptyset$:



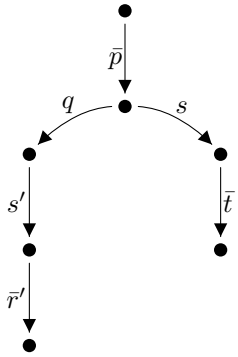
We start off by merging q and s:



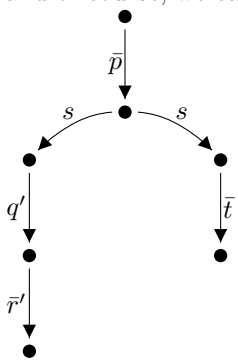
Note that qs' commutes to sq'. Now recursively merge s' with $\bar{r}$:



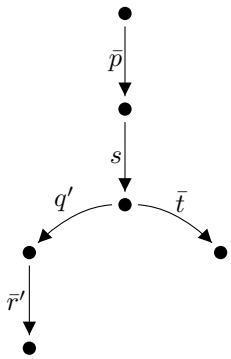
For the same reason that q and s' commute, s'' can be commuted past all the patches in $\bar{r}$:



and of course, we can then commute s' and q:



Coalescing our two $\bar{p}s$ nodes gives us:



so we have merged s. Now we merge each patch in $\bar{t}$ in the same way, resulting in:

7 Contexted patches

Definition 7.1 (contexted-patch)
For all patches $r \in \mathbf{P}$, $: r$ is a *contexted patch* of r. For all contexted patches $\bar{q} : r$ and patches $p \in \mathbf{P}$, $p\bar{q} : r$ is a *contexted patch* of r if and only if $\langle p, \bar{q}r \rangle \longleftrightarrow$ fail and $\bar{q} \neq p^{-1}\bar{q}'$ for some sequence $\bar{q}'$.

Explanation
Sometimes we will want to keep a "reference" to a patch p. However, for a patch to be useful we have to know what context it should be applied in. The purpose of a contexted patch is to keep track of a patch, and the context in which it applies.

The patch sequence before the colon is the context; the patch after the colon is the patch that we are interested in.

Definition 7.2 (patch-commute-past)
Given a patch p and a contexted patch $\bar{q} : r$, we define $\rightarrow$, pronounced *commute past*, thus:
$\langle p, \bar{q} : r \rangle \rightarrow \langle \bar{q}' : r' \rangle$ if $(\langle p, \bar{q} \rangle \longleftrightarrow \langle \bar{q}', p' \rangle) \wedge (\langle p', r \rangle \longleftrightarrow \langle r', p'' \rangle)$
$\langle p, \bar{q} : r \rangle \rightarrow$ fail otherwise

Definition 7.3 (patch-commute-past-set)
We extend $\rightarrow$ to work on sets of contexted patches in the natural way, i.e. for any patch p and set of contexted patches S:
$\langle p, S \rangle \rightarrow \langle \{ (\bar{q}' : r') \mid (\bar{q} : r) \in S \wedge \langle p, \bar{q} : r \rangle \rightarrow \langle \bar{q}' : r' \rangle \} \rangle$
if all the commute pasts succeed, and $\langle p, S \rangle \rightarrow$ fail otherwise.

We assume that contexted patches magically maintain their invariant, i.e. if we have a patch p and a contexted patch $\bar{q} : r$ then when we write $p\bar{q} : r$ what we mean is, if $\langle p, \bar{q} : r\rangle \to \langle \bar{q}' : r'\rangle$ then $\bar{q}' : r'$, otherwise $p\bar{q} : r$.

Definition 7.4 (contexted-patch-conflict)
We define $\leftrightarrows$, pronounced "does not conflict with", such that $(\bar{p} : q) \leftrightarrows (\bar{r} : s)$ holds if $\left\langle q^{-1}, \bar{p}^{-1}\bar{r} : s\right\rangle \to \langle \_\rangle$, and does not hold otherwise.

Explanation
Here $\bar{p} : q$ and $\bar{r} : s$ are two contexted patches starting from the same context. By inverting one of them we can put them into a single patch sequence $q^{-1}\bar{p}^{-1}\bar{r}s$. By building the contexted patch $\bar{p}^{-1}\bar{r} : s$ if $\bar{p}$ and $\bar{r}$ both contain a patch t, that patch will be removed (as contexted patches magically maintain their invariant).

This is almost, but not quite the same thing as saying $\bar{p}q$ and $\bar{r}s$ can be cleanly merged. For a counter-example, consider $: t$ and $t : u$. Clearly t and tu can be cleanly merged, giving tu, but $\left\langle t^{-1}, t : u\right\rangle \to$ fail. Likewise, starting with the contexted patches the other way round, we get $\left\langle u^{-1}, t^{-1} : t\right\rangle \to$ fail.

8 Catches and Repos

Now that we have laid the foundations, it is time to introduce conflictors. We will now be dealing with another datatype, which may either be a patch (as previously defined) or a *conflictor*. We will call these beasts *catches*.

Explanation
The basic idea is that, if two catches do not conflict, then we can merge them and get the effect of both, as we described earlier. However, if they do conflict then when we merge, we get the effect of neither. Thus to compute the contents of a repository we take the effects of all the patches that are in the repository and do not conflict with any other patch in the repository.

Definition 8.1 (catches)
A catch is either $[p]$ (the non-conflicted patch $p \in \mathbf{P}$), $[\bar{r}, X, \bar{p} : q]$ (a conflicted patch $q \in \mathbf{P}$), or $[\![\bar{r}, X, \bar{p} : q]\!]$ (the inverse of a conflicted patch $q \in \mathbf{P}$). In both cases, $\bar{r}$ is a sequence of patches, X is a set of contexted patches, and $\bar{p} : q$ is a contexted patch.

Definition 8.2 (repos)
A *repo* is a sequence of catches (up to commutation, to be defined later) satisfying some requirements (that, again, we will give later).

Let $\mathbf{R}$ be the (possibly infinite) set of repos.

Aside
Do we really need inverse conflictors, or can we just use conflictors and invert their internals?

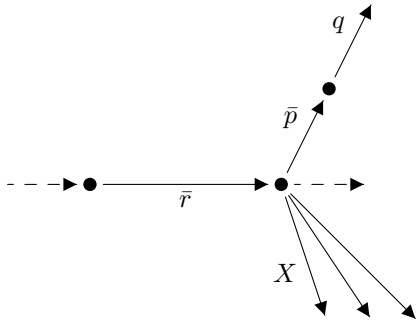
The meaning of $[p]$ is hopefully clear, but what is the meaning of $[\bar{r}, X, \bar{p} : q]$? To answer this question, we need to consider it in the context of a repo.

Suppose we have the repo $\bar{c}\,[\bar{r}, X, \bar{p} : q]$. The effect of the conflictor on the repo is $\bar{r}$, and as we have already said, $\bar{p} : q$ is the (contexted) patch that this conflictor represents. X is the set of (contexted) patches in $\bar{c}$ that q conflicts with.

Aside
In darcs 2, the transitive set of conflicts is stored. I don't believe that this is needed, and not having it makes things simpler. Not having it may also mean more things commute.

But how is $\bar{r}$ calculated? $\bar{r}$ is the inverses of the subset of the patches in X that do not appear in the effect of any conflictor in $\bar{c}$. In other words, the first catch to conflict with any given patch reverts that patch.

We can picture a conflictor $[\bar{r}, X, \bar{p} : q]$ in a repo as looking like this:



Definition 8.3 (catch-effect)
We define $\mathscr{E}$ to tell us the effect that a catch has on the repo:
$\mathscr{E}([p]) = p$
$\mathscr{E}([\bar{r}, X, y]) = \bar{r}$
$\mathscr{E}([\![\bar{r}, X, y]\!]) = \bar{r}^{-1}$

Definition 8.4 (catch-conflicts)
We define $\mathscr{C}$ to tell us the patches that a catch conflicts with, i.e.:
$\mathscr{C}([p]) = \emptyset$
$\mathscr{C}([\bar{r}, X, y]) = N(X)$
$\mathscr{C}([\![\bar{r}, X, y]\!]) = N(X)^{-1}$

Definition 8.5 (catch-names)
We extend n and N to work on catches in the natural way, i.e.:
$n([p]) = p$
$n([\bar{r}, X, \bar{p} : q]) = n(q)$
$n([\![\bar{r}, X, \bar{p} : q]\!]) = n(q)^{-1}$
$N(\epsilon) = \emptyset$
$N\left(c\bar{d}\right) = \{n(c)\} \cup N\left(\bar{d}\right)$

Definition 8.6 (repo-properties)
A repo must satisfy the following:

ϵ is a repo.

$\bar{c}[p]$ is a repo if

- $\bar{c}$ is a repo
- $n(p)$ is positive
- $n(p) \notin N(\bar{c})$
- $\mathscr{E}(c)\, p$ is sensible

$\bar{c}\bar{d}\bar{e}[\bar{r}, X, \bar{p} : q]$ is a repo if:

- $\bar{c}\bar{d}\bar{e}$ is a repo
- $\bar{e}$ is a sequence of patches (i.e. there are no conflictors in $\bar{e}$)
- $\mathscr{E}(\bar{e}) = \bar{r}^{-1}$
- $\mathscr{E}\left(\bar{d}\right) = \bar{p}^{-1}$

- $N\left(\bar{d}\bar{e}\right) = N\left(X\right)$
- Every catch in d is either a conflictor, or in $\mathscr{C}\left(d\right)$
- $n\left(y\right)$ is positive
- $n\left(y\right) \notin N\left(\bar{c}\bar{d}\bar{e}\right)$
- There is no catch f and catch sequence $\bar{g}$ such that $\bar{d}\bar{e} = f\bar{g}$ and f does not conflict with $[q]$.
- $\mathscr{E}\left(c\right)q$ is sensible

$\bar{c}\bar{d}\bar{d}^{-1}$ is a repo if $\bar{c}\bar{d}$ is a repo and all of $N\left(\bar{d}\right)$ are positive.

Definition 8.7 (catch-inverse)
We define
$[p]^{-1} = \left[p^{-1}\right]$
$[\bar{r}, X, y]^{-1} = [\![\bar{r}, X, y]\!]$
$[\![\bar{r}, X, y]\!]^{-1} = [\bar{r}, X, y]$

8.1 Commute

We now define commutation of catches.

Definition 8.8 (catch-commute)
We extend $\longleftrightarrow$ to operate on catches, as defined below.

We can break the rules down into 3 classes: Those where the patches are unrelated, and simply commute freely; those where the patches are the result of a conflicting merge, and the conflicts gets reshuffled when they commute; and a fail case for anything else.

Currently we don't give the commute rules for anything involving inverse conflictors. We don't believe that those rules will raise any new problems, though.

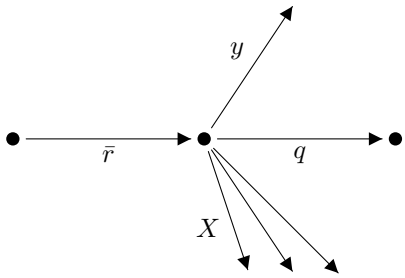
Unrelated

First the simple non-conflicted patch case:
$\langle [p], [q] \rangle \longleftrightarrow \langle [q'], [p'] \rangle$ if $\langle p, q \rangle \longleftrightarrow \langle q', p' \rangle$

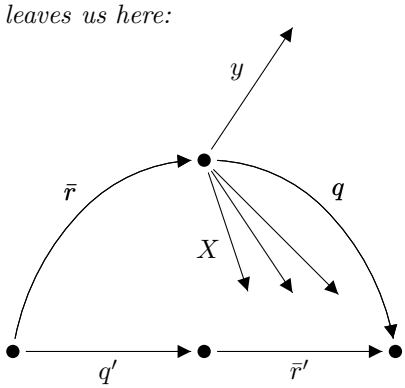
Explanation
If our catches just wrap up patches, then they commute like the underlying patches.

Now commute a conflictor past a patch, where everything goes smoothly:
$$\langle [\bar{r}, X, y], [q] \rangle \longleftrightarrow \langle [q'], [\bar{r}', X', y'] \rangle \text{ if } \begin{array}{l} \langle \bar{r}, q \rangle \longleftrightarrow \langle q', \bar{r}' \rangle \\ \left\langle q^{-1}, y \right\rangle \rightarrow \langle y' \rangle \\ \left\langle q^{-1}, X \right\rangle \rightarrow \langle X' \rangle \end{array}$$

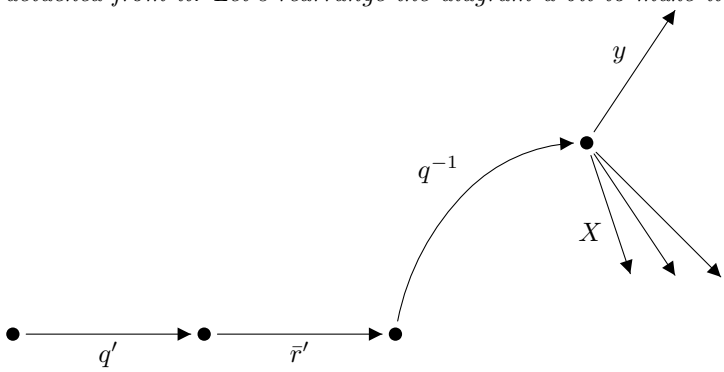
Explanation
OK, now things start to look a little more daunting, but if you work through it then it's really not that bad. We start off with this situation:



Now, we want to commute the two patches, so clearly we need to commute $\bar{r}$ and q, which leaves us here:



The $q'\bar{r}'$ is what we want, but the y and X that should be part of the conflictor have become detached from it. Let's rearrange the diagram a bit to make it clearer what we need to do:



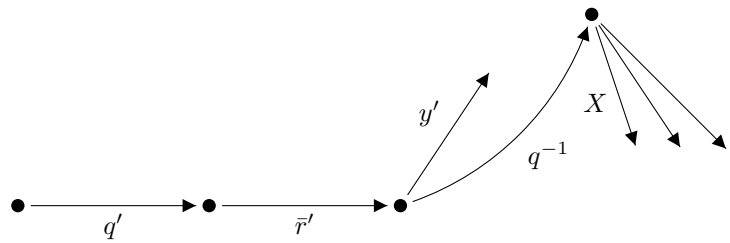
So we need to add q^{-1} into the context of y and X. We have two ways of doing this: We can simply add q^{-1} to the context, which cannot fail, or we can commute q^{-1} past y and X, which can. Being able to commute patches is good, so if possible we would prefer to use the option that doesn't fail; but is that possible?

First, let's consider y, and for the purpose of contradiction, let us assume that the catch commute can suceed even if q^{-1} cannot be commuted past y.

Suppose we have two repos $[s]$ and $[t]$, where s and t conflict. We merge them, giving us $[s]\left[s^{-1}, \{: s\}, : t\right]$, and then record another patch $[u]$, giving us $[s]\left[s^{-1}, \{: s\}, : t\right][u]$. We choose u such that $\left\langle s^{-1}, u\right\rangle \longleftrightarrow \left\langle u', s'^{-1}\right\rangle$ and $\left\langle t^{-1}, u\right\rangle \longleftrightarrow$ fail. Then we can commute $[u]$ past both catches, giving us $[u][s']\left[s'^{-1}, \{: s'\}, u^{-1} : t\right]$. So far so good.

But before we commute u past, we can first commute s and t, giving us $[t]\left[t^{-1}, \{: t\}, : s\right][u]$. But now it is not possible to commute u past!

This isn't what we want, so we now know that we must require that the commute past y succeeds. This explains why we require $\left\langle q^{-1}, y\right\rangle \rightarrow \langle y'\rangle$, and gets us to here:



Now we need to work out whether we need to require that q^{-1} commutes past X or not. Again, for the purpose of contradiction, assume that we don't require this.

Suppose we have three repos $[s]$, $[t]$ and $[u]$, where s conflicts with t and u. Then we merge these three repos to give us something like

$$[s]\left[s^{-1}, \{: s\}, : t\right]\left[\epsilon, \{: s\}, : u\right]$$

Next record another patch v, which commutes with everything except for s^{-1}. So now we have

$$[s]\left[s^{-1}, \{: s\}, : t\right]\left[\epsilon, \{: s\}, : u\right][v]$$

Now, we can commute v past the u conflictor, but not the t conflictor (as it cannot commute past the effect, s^{-1}):

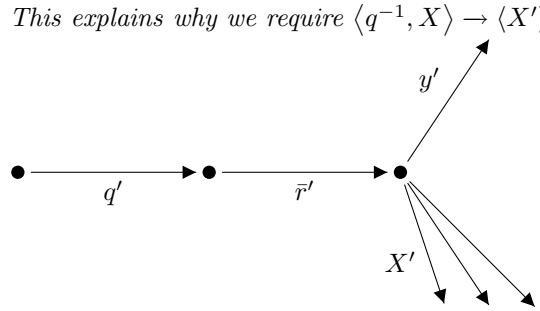
$$[s]\left[s^{-1}, \{: s\}, : t\right][v]\left[\epsilon, \left\{v^{-1} : s\right\}, : u'\right]$$

Likewise, we can first commute the t and u conflictors and then commute v past the t conflictor, but not the u conflictor:

$$[s]\left[s^{-1}, \{: s\}, : u\right][v]\left[\epsilon, \left\{v^{-1} : s\right\}, : t'\right]$$

By unpulling patches from these last two repos, we get repos with the patches $\{s, t, v\}$ and $\{s, u, v\}$, but it is not possible to make a repo containing the patches $\{s, v\}$. This causes problems when we try to merge these two repos, as explained in Appendix A, so this is also not something that we want to allow.

This explains why we require $\left\langle q^{-1}, X\right\rangle \to \langle X'\rangle$, and brings us to our destination:

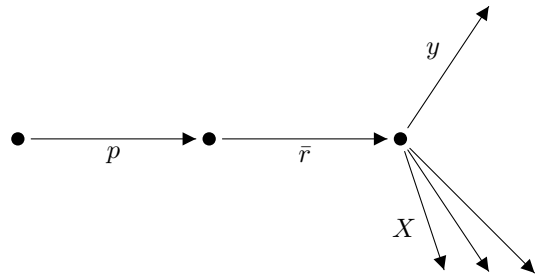


Next we have the case where the patch and conflictor start off the other way round:

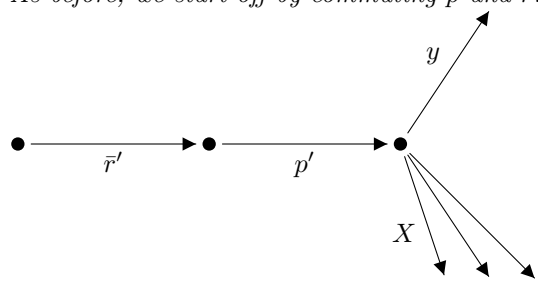
$$\langle [p], [\bar{r}, X, y]\rangle \longleftrightarrow \langle [\bar{r}', X', y'], [p']\rangle \text{ if } \begin{array}{l} \langle p, \bar{r}\rangle \longleftrightarrow \langle \bar{r}', p'\rangle \\ \langle p', X\rangle \to \langle X'\rangle \\ \langle p', y\rangle \to \langle y'\rangle \end{array}$$

Explanation

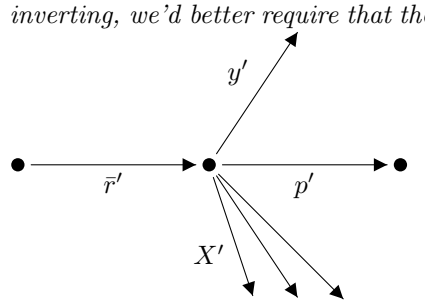
This is similar to the previous case. We start off here:



As before, we start off by commuting p and $\bar{r}$:



We now need to get X and y into the right context, and as we want commute to be self-inverting, we'd better require that the commute pasts succeed, as we did in the previous case:

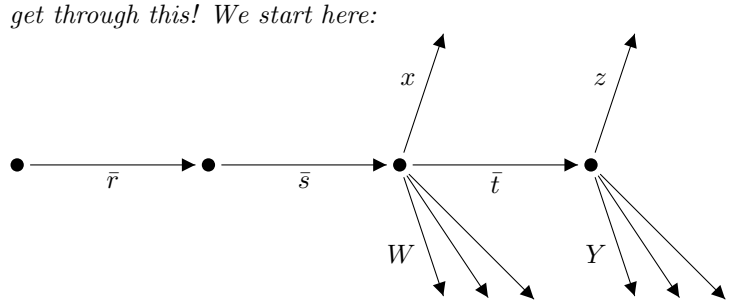


And the last and most complex of the unrelated cases, commuting a conflictor past another conflictor:

$\langle [\bar{r}\bar{s}, W, x], [\bar{t}, Y, z] \rangle \longleftrightarrow \left\langle [\bar{r}\bar{t}', \bar{s}'Y, z'], \left[\bar{s}', \bar{t}^{-1}W, x'\right] \right\rangle$
if $N\left(\bar{r}^{-1}\right) \subseteq N(Y)$
$N\left(\bar{s}^{-1}\right) \cap N(Y) = \emptyset$
$\langle \bar{s}, \bar{t} \rangle \longleftrightarrow \langle \bar{t}', \bar{s}' \rangle$
$x \leftrightarrows \bar{t}z$
$\forall w \in W \cdot (w \leftrightarrows \bar{t}z)$
$\forall y \in Y \cdot (x \leftrightarrows \bar{t}y)$
$\left\langle \bar{t}^{-1}, x \right\rangle \rightarrow \langle x' \rangle$
$\langle \bar{s}', z \rangle \rightarrow \langle z' \rangle$

Explanation

The number of rules and side conditions grows, but if we take it one step at a time we can get through this! We start here:



The first thing to note is that we have $\bar{r}\bar{s}$ where you probably expected $\bar{t}$. What we have done

here is to break up the effect of the first conflictor into those patches that both *conflictors conflict with, $\bar{r}$, and those patches that only the first conflictor conflicts with, $\bar{s}$. $\bar{r}$ won't move; it'll become part of the z conflictor after the commute, thus maintaining the invariant that the first conflictor that conflicts with a patch reverts it. The first two side conditions just say that we have correctly split up the effect of the first conflictor.*

With that out of the way, let's start to look at how we perform the commute. We start, as usual, by commuting the effects of the two patches. Now comes the difficult part: Where do we need to check for conflicts?

First let's consider the repo $[p]\,[q]$, where q depends on p, and the repo $[u]$, where p conflicts with u. If we merge these as $[p]\,[q]\left[q^{-1}p^{-1}, \{: p, p : q\}, : u\right]$ then we cannot commute the p and q catches. Therefore, in the alternate merge $[u]\left[u^{-1}, \{: u\}, : p\right]\left[, \{: u\}, p : q\right]$ the catches shouldn't commute either. Therefore we need to check that p and $p : q$ don't conflict, or in general, $x \leftrightarrows \bar{t}z$.

Now let us consider whether we need to worry about z and W conflicting. Suppose first we record o, p and q, which conflict, in separate repos, and then merge them, giving us

$$[o]\left[o^{-1}, \{: o\}, : p\right]\left[, \{: o, : p\}, : q\right]$$

Next record u (which commutes with o, p and q) and v (which commutes with q but not o or p) in two copies of this repo, and merge, giving us

$$[o]\left[o^{-1}, \{: o\}, : p\right]\left[, \{: o, : p\}, : q\right][u]\left[u^{-1}, \{: u\}, : v\right]$$

Now, we can commute u and v, giving us

$$[o]\left[o^{-1}, \{: o\}, : p\right]\left[, \{: o, : p\}, : q\right][v]\left[v^{-1}, \{: v\}, : u\right]$$

and by the earlier rules we know that the q catch does not commute with the v catch. But we could also have commuted q and u, giving us

$$[o]\left[o^{-1}, \{: o\}, : p\right][u]\left[, \{: o', : p'\}, : q'\right]\left[u^{-1}, \{: u\}, : v\right]$$

Now we musn't be allowed to commute the q and v conflictors, and we must therefore, in the general case, check that z and W do not conflict. This gives us the $\forall w \in W \cdot (w \leftrightarrows \bar{t}z)$ side condition, and by considering commuting the patches back the other way we must also have $\forall y \in Y \cdot (x \leftrightarrows \bar{t}y)$.

If this is satisfied then we know that the commute pasts will succeed.

This only leaves conflicts between W and Y. However, I claim that it is not possible to get into a situation where there are any conflicts here; Any patch in Y would have had to be commuted past the x conflictor at some point in order to get into this situation, and therefore cannot conflict with W.

Conflicted merge

Now we get to the interesting cases. First, if we have a patch, and a conflictor that has only conflicted with that patch, then they swap places:
$\left\langle [p], \left[p^{-1}, \{: p\}, : q\right]\right\rangle \longleftrightarrow \left\langle [q], \left[q^{-1}, \{: q\}, : p\right]\right\rangle$

Explanation
This should make sense if you consider that the result of merging two conflicting patches p and q is $[p]\left[p^{-1}, \{: p\}, : q\right]$ (i.e., from right to left, the second catch represents q, conflicts with p, and as it is the first patch to conflict with p it has to invert p's effect), or equivalently $[q]\left[q^{-1}, \{: p\}, : q\right]$.

If we have two conflictors, but the one on the right only conflicts with the one on the left, then it becomes a patch after it has commuted:
$\left\langle [\bar{r}, X, y], \left[\epsilon, \{y\}, \bar{r}^{-1} : q\right]\right\rangle \longleftrightarrow \left\langle [q], \left[q^{-1}\bar{r}, \left\{\bar{r}^{-1} : q\right\} \cup X, y\right]\right\rangle$

Explanation
This case comes up when you merge a patch q with a conflictor representing p. If q is the first in the resulting sequence then it is still just $[q]$, and the q conflictor must record that it conflicts with p. On the other hand, if p comes after p then it must also turn into a conflictor, as it needs to record that is has conflicted with p.

And the inverse of the previous case:
$\left\langle [p], \left[p^{-1}\bar{r}, \left\{\bar{r}^{-1} : p\right\} \cup X, y\right]\right\rangle \longleftrightarrow \left\langle [\bar{r}, X, y], \left[\epsilon, \{y\}, \bar{r}^{-1} : p\right]\right\rangle$

Explanation
This is just the inverse of the previous case.

And now the case where both are conflictors, and conflict with each other:
$\left\langle [\bar{r}\bar{s}, W, x], \left[\bar{t}, \left\{\bar{t}^{-1}x\right\} \cup Y, z\right]\right\rangle \longleftrightarrow \left\langle [\bar{r}\bar{t}', \bar{s}'Y, \bar{s}'z], \left[\bar{s}', \{z\} \cup \bar{t}^{-1}W, \bar{t}^{-1}x\right]\right\rangle$
if $\langle \bar{s}, \bar{t}\rangle \longleftrightarrow \langle \bar{t}', \bar{s}'\rangle$
$N\left(\bar{r}^{-1}\right) \subseteq N(Y)$
$N\left(\bar{s}^{-1}\right) \cap N(Y) = \emptyset$

Explanation
In this case we just move the conflict from one to the other. The splitting of the effect of the z conflictor into two parts is the same as we saw earlier, in the "unrelated" conflict-conflictor commute.

Fail

Finally, if none of the above hold, then
$\langle c, d\rangle \longleftrightarrow \text{fail}$

Conjecture 8.1 (catch-commute-conflicting-patches-succeeds)
$\forall(\bar{c}de) \in \mathbf{R}\cdot$
$d \in \mathscr{C}(e) \Rightarrow (\langle d, e\rangle \longleftrightarrow \langle \_, \_\rangle)$

Explanation
If two patches conflict (and thus, were merged at some point in the past), then they can be commuted.

Conjecture 8.2 (catch-commute-unique)
$\forall(\bar{c}de) \in \mathbf{R}, r \in (\mathbf{C} \times \mathbf{C}) \cup \{\text{fail}\}, s \in (\mathbf{C} \times \mathbf{C}) \cup \{\text{fail}\}\cdot$
$(\langle d, e\rangle \longleftrightarrow r) \wedge (\langle d, e\rangle \longleftrightarrow s) \Rightarrow r = s$

Explanation
This states that Axiom 3.1, i.e. that the result of a commute is unique, holds for catches too.

Conjecture 8.3 (catch-commute-valid)
$\forall(\bar{c}de) \in \mathbf{R}, d' \in \mathbf{C}, e' \in \mathbf{C}\cdot$
$(\langle d, e\rangle \longleftrightarrow \langle e', d'\rangle) \Rightarrow (\bar{c}e'd') \in \mathbf{R}$

Explanation
If we have a valid repository (one in which the invariants are all satisfied) and we commute two catches, then the result is also a valid repository.

Conjecture 8.4 (catch-commute-self-inverse)
$\forall(\bar{c}de) \in \mathbf{R}, \forall d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$(\langle d, e\rangle \longleftrightarrow \langle e', d'\rangle) \Leftrightarrow (\langle e', d'\rangle \longleftrightarrow \langle d, e\rangle)$

Explanation
This states that Axiom 3.2, i.e. that successful commute is self-inverting, holds for catches too.

Conjecture 8.5 (catch-commute-square)
$\forall(\bar{c}d) \in \mathbf{R}, (\bar{c}e) \in \mathbf{R}, \forall d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$(\left\langle d^{-1}, e\right\rangle \longleftrightarrow \left\langle e', d'^{-1}\right\rangle) \Leftrightarrow (\left\langle e^{-1}, d\right\rangle \longleftrightarrow \left\langle d', e'^{-1}\right\rangle)$

Explanation
This states that Axiom 4.4, i.e. that commute squares can be rotated, holds for catches too.

Definition 8.9 (catch-conflicts)
If $\forall(\bar{c}d) \in \mathbf{R}$ and $(\bar{c}e) \in \mathbf{R}$, We say d conflicts with e if and only if $\left\langle d^{-1}, e\right\rangle \longleftrightarrow$ fail.

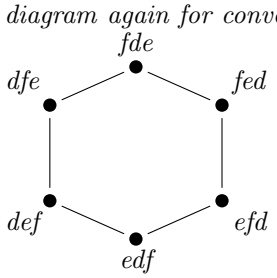
Explanation
XXX

Conjecture 8.6 (catch-commute-preserves-effect)
$\forall(\bar{c}de) \in \mathbf{R}, \forall d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$(\langle d, e\rangle \longleftrightarrow \langle e', d'\rangle) \Rightarrow \mathscr{E}(de) = \mathscr{E}(e'd')$

Explanation
XXX

Conjecture 8.7 (catch-commute-associates)
$\forall(\bar{c}def) \in \mathbf{R}, d_e \in \mathbf{C}, d_f \in \mathbf{C}, d_{ef} \in \mathbf{C}, e_d \in \mathbf{C}, e_f \in \mathbf{C}, e_{df} \in \mathbf{C}, f_d \in \mathbf{C}, f_e \in \mathbf{C}, f_{de} \in \mathbf{C} \cdot$
$(\langle e, f\rangle \longleftrightarrow \langle f_e, e_f\rangle) \wedge (\langle d, e\rangle \longleftrightarrow \langle e_d, d_e\rangle) \wedge (\langle d_e, f\rangle \longleftrightarrow \langle f_d, d_{ef}\rangle)$
$\Leftrightarrow (\langle e_d, f_d\rangle \longleftrightarrow \langle f_{de}, e_{df}\rangle) \wedge (\langle e_{df}, d_{ef}\rangle \longleftrightarrow \langle d_f, e_f\rangle) \wedge (\langle f_{de}, d_f\rangle \longleftrightarrow \langle d, f_e\rangle)$

Explanation
This states that Axiom 5.6, i.e. that commute associates, holds for catches too. Here's the diagram again for convenience:



8.2 Merge

We now define merge; this has no equivalent in the land of patches.

Definition 8.10 (merge)
We write the merge operator for catches as $+$. It cannot fail: It introduces conflictors instead.
$\forall(\bar{c}d) \in \mathbf{R}, (\bar{c}e) \in \mathbf{R} \cdot$
$(n(d) = n(e)) \vee$
$\exists d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$d + e = \langle e', d'\rangle$

Explanation
What we're saying here is that if we have two repos that differ in only their last patch then we can always merge them. The result of the merge is going to be the repo cd', or equivalently dc'.

We define + thus:
$c + d = \langle d', c' \rangle \qquad \text{if } \left\langle c^{-1}, d \right\rangle \longleftrightarrow \left\langle d', c'^{-1} \right\rangle$
$[p] + [q] = \left\langle \left[p^{-1}, \{: p\}, : q\right], \left[q^{-1}, \{: q\}, : p\right] \right\rangle$
$[p] + [\bar{r}, X, y] = \left\langle \left[p^{-1}\bar{r}, X \cup \left\{\bar{r}^{-1} : p\right\}, y\right], \left[\epsilon, \{y\}, \bar{r}^{-1} : p\right] \right\rangle$
$[\bar{r}, X, y] + [q] = \left\langle \left[\epsilon, \{y\}, \bar{r}^{-1} : q\right], \left[q^{-1}\bar{r}, X \cup \left\{\bar{r}^{-1} : q\right\}, y\right] \right\rangle$
$[\bar{r}\bar{s}, W, x] + [\bar{r}\bar{t}, Y, z] = \left\langle \left[\bar{t}', (\bar{s}'^{-1}Y) \cup \left\{\bar{t}'^{-1}x\right\}, \bar{s}'^{-1}z\right], \left[\bar{s}', (\bar{t}'^{-1}W) \cup \left\{\bar{s}'^{-1}z\right\}, \bar{t}'^{-1}x\right] \right\rangle$
$\quad \text{if } N(\bar{s}) \cap N(\bar{t}) = \emptyset$
$\quad\quad \left\langle \bar{s}^{-1}, \bar{t} \right\rangle \longleftrightarrow \left\langle \bar{t}', \bar{s}'^{-1} \right\rangle$

Explanation
The first rule handles the case where there is no conflict when merging, while the remaining four handle the various conflicting cases.

Note that the third and fourth rules are identical, but with the arguments and results in the opposite order.

In the final case we need a side condition to ensure that we have separated out all of the patches that both conflictors are currently the first to conflict with.

Conjecture 8.8 (merge-commute-cannot-fail)
The commute in the merge definition cannot fail.

Explanation
If it fails then something has gone badly wrong, as merge is not allowed to fail!

Conjecture 8.9 (merge-makes-repos)
If $(\bar{c}d) \in \mathbf{R}, (\bar{c}e) \in \mathbf{R}, d' \in \mathbf{C}, e' \in \mathbf{C}, d + e = \langle e', d' \rangle$ then $\bar{c}de' \in \mathbf{R} \wedge \bar{c}ed' \in \mathbf{R}$.

Explanation
This states that merging two valid repositories results in another valid repository.

Conjecture 8.10 (merge-effect)
XXX This is rubbish. Fix it. Or just rely on the spec of the effect of a repo, which we haven't given yet.

If $(\bar{c}d) \in \mathbf{R}$, $(\bar{c}e) \in \mathbf{R}$ and $d + e = \langle e', d' \rangle$ then $\mathscr{E}(de') = \mathscr{E}(ed')$ if $\left\langle d^{-1}, e \right\rangle \longleftrightarrow \langle \_, \_ \rangle$, and id otherwise.

Explanation
XXX Actually, I think this is wrong if the two catches are not both patches.
Anyway, the idea is that we should apply the affects if the merge succeeds, but there should be no net effect if it doesn't.

Conjecture 8.11 (merge-symmetric)
If $(\bar{c}d) \in \mathbf{R}$, $(\bar{c}e) \in \mathbf{R}$ and $d + e = \langle e', d' \rangle$ then $e + d = \langle d', e' \rangle$.

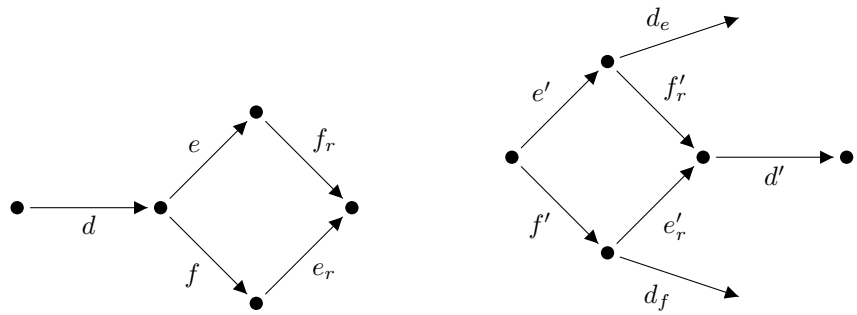
Explanation
Merging d and e is the same as merging e and d.

Conjecture 8.12 (merge-commute)
If $(\bar{c}d) \in \mathbf{R}$, $(\bar{c}e) \in \mathbf{R}$ and $d + e = \langle e', d' \rangle$ then $\langle d, e' \rangle \longleftrightarrow \langle e, d' \rangle$.

Explanation
The two ways of constructing the result of a merge are equivalent.

Conjecture 8.13 (merge-commute2)
If $(\bar{c}de) \in \mathbf{R}, (\bar{c}df) \in \mathbf{R}, e + f = \langle f_r, e_r \rangle, \langle d, e \rangle \longleftrightarrow \langle e', d_e \rangle, \langle d, f \rangle \longleftrightarrow \langle f', d_f \rangle$ then
$\langle d_e, f_r \rangle \longleftrightarrow \langle \_, d' \rangle, \langle d_f, e_r \rangle \longleftrightarrow \langle \_, d' \rangle, e' + f' = \langle f'_r, e'_r \rangle, d_e + f'_r = \langle \_, d' \rangle, d_f + e'_r = \langle \_, d' \rangle$

Explanation
First, a couple of diagrams may help make it clearer what is going on:



This conjecture says that if d commutes past e and f then we can commute it before or after merging, and get consistent results either way.

A Named Patch Motivation

When you record in darcs, you create a *mega patch* composed of many catches (which, at the point at which you record it, are all just patches). You need to give a name to this mega patch, but to avoid confusion with the names given to patches and catches we'll say that mega patches have a *title*.

A non-obvious result is that, if we don't give names to patches, then dependencies of mega patches *cannot* behave as one would expect when duplicate changes are involved. Furthermore, one can construct situations where the simple, natural merge algorithm fails.

The detail of what is inside a conflictor is irrelevant, as this is a universal property, so for this section we will simply write $[\bar{p}, q]$ for a conflictor representing q that has effect $\bar{p}$.

To start off with, we record a mega patch p in one repo, and q in another repo. p and q contain the same single patch. We also record a mega patch r that depends on p, i.e. p and r do not commute. So we have three repos: p, q, pr.

Next merge p and q, resulting in $p\left[p^{-1}, q\right]$, and then merge this with pr, resulting in $p\left[p^{-1}, q\right][, r]$.

Now, this must commute to $pr\left[r^{-1}p^{-1}, q\right]$, and we can then unpull the q conflictor to get pr.

But going back to $p\left[p^{-1}, q\right][, r]$, this must also commute to $q\left[q^{-1}, p\right][, r]$. But if the patches in p and q are not named, and are identical to each other, then $\left[p^{-1}, q\right]$ and $\left[q^{-1}, p\right]$ look identical to the r conflictor! So this must commute to $qr\left[r^{-1}q^{-1}, p\right]$ and again we can unpull to get qr. But in the land of named patches, r is supposed to depend on p!

This is disturbing enough, but now suppose that we create these pr and qr repos and then try to merge them. We first want to get all the mega patches that are common to both repos out of the way. We look at the titles of the mega patches in each repo and conclude that r is common to both. We thus want to commute the repos so that they are $r'p'$ and $r'q'$ respectively. But r depends on p/q, so this commute fails!