1 Related work

See [1] and [2]. (TODO: in some places where we cite [1], it might make more sense to cite earlier work instead.)

This write-up is based on conversations on the `darcs-users` mailing list between James Cook and Ben Franksen.

2 Patch universes

Our overall goal is to extend any universe of patches so that patches can always be commuted. To do that, we need to first define what a patch universe is.

In Section 2.1, we give an unusual but fairly simple definition. In the following sections we relate this definition to other formulations: Section 2.2 describes a way of interpreting our definition; Section 2.3 shows how standard patch theory properties follow from our definition; and in Section 2.4 we comment on whether our definition follows from standard patch theory properties.

2.1 Definition

We start by defining a notion of *patch graph* without inverses. Then we define *invertible patch graphs* where patches have inverses, and state a final property that such a graph must have in order to be a *patch universe*.

A *patch graph* is a possibly infinite directed multigraph, where the edges are labelled by *names*. The nodes are called *contexts* and the edges are called *patches*. This is captured in the following definition:

Definition 1 (Patch graph)**.** *A* patch graph *consists of arbitrary sets C and N together with a set $P \subseteq C \times C \times N$. Elements of C, N and P are called* contexts*,* names *and* patches*.*

For a patch $p = (c_1, c_2, n)$, its starting context *is* $\mathrm{start}(p) = c_1$*, its* ending context *is* $\mathrm{end}(p) = c_2$*, and its* name *is* $\mathrm{name}(p) = n$*.*

It is often useful to have an inverse operation on patches. The next definition shows how to add inverses to a patch graph. An alternative approach would be to describe axioms that inverses should satisfy, but it seems simpler to simply prescribe directly how inverses behave.

Definition 2 (Adding inverses)**.** *Given a patch graph $G = (C, N, P)$, the* extension with inverses of G*, denoted* $\mathrm{addinv}(G)$*, is a new graph where names gain a sign (like* positive *and* negative *names in [2]), original patches from G have positive names, and we add an inverse of every patch in G with a negative name.*

More precisely: $\mathrm{addinv}(G) = (C, \{1, -1\} \times N, P')$*, where* $P' = \{(c_1, c_2, (1, n)) \mid (c_1, c_2, n) \in G\} \cup \{(c_2, c_1, (-1, n)) \mid (c_1, c_2, n) \in G\}$.

The inverse *of a name $n = (s, n')$ is $n^{-1} = (-s, n')$. The* inverse *of a patch $p = (c_1, c_2, n)$ is $p^{-1} = (c_2, c_1, n^{-1})$.*

Definition 3 (Invertible patch graph)**.** *An* invertible patch graph *is a patch graph G such that $G = \text{addinv}(G')$ for some patch graph $G' = (C, P, N)$.*

When we want to distinguish names and patches in G vs. G', we use the terms unsigned name *and* unsigned patch *for names and patches in N and P, and* signed name *and* signed patch *(or just* name *and* patch*) for names and patches in $\{1, -1\} \times N$ and P'.*

An invertible patch graph needs one more property, involving *patch sequences* and *signed name multisets*, in order to be a *patch universe*.

Definition 4 (Patch sequence)**.** *A* patch sequence *in a patch graph (C, P, N) is a finite path in the graph.*

More precisely, a patch sequence is a sequence of contexts $c_0, c_1, \ldots, c_n \in C$ together with a sequence of patches $p_1, \ldots, p_n \in P$ where for all $i \in \{1, \ldots, n-1\}$, $\text{start}(p_i) = c_{i-1}$ and $\text{end}(p_i) = c_i$.

When n is is at least 1, we may omit the context sequence $c_0, \ldots, c_n$ and describe the sequence only in terms of the patch sequence $p_1, \ldots, p_n$.

The length n of the sequence is denoted $\text{length}(S)$. We say $\text{start}(S) = c_0$ and $\text{end}(S) = c_n$.

(The reason we explicitly make the context sequence $c_0, \ldots, c_n$ part of the definition is to ensure a length-0 sequence still has a starting and ending context.)

Definition 5 (Signed name multiset)**.** *Let $G = \text{addinv}(G')$ be an invertible patch graph, and let N be the set of unsigned names, i.e. names in the patch graph G'. A* signed name multiset *is a function $A : N \to \mathbb{Z}$.*

The name signature *of a patch sequence S is the signed name multiset A where $A(n)$ is equal to the number of times the positive name $(1, n)$ appears in S minus the number of times the negative name $(-1, s)$ appears in S.*

For example, consider this patch sequence:

$$c_0 \xrightarrow{\ p\ } c_1 \xrightarrow{\ q\ } c_2 \xrightarrow{\ p'^{-1}\ } c_3 \xrightarrow{\ q'\ } c_4$$

where $\text{name}(p) = (1, n_p)$, $\text{name}(p'^{-1}) = (-1, n_p)$, and $\text{name}(q) = \text{name}(q') = (1, n_q)$. The name signature of the sequence is A where $A(n_q) = 2$ and $A(n') = 0$ for all other names n'.

The essential property of a patch universe is that a sequence of patches where each name occurs the same number of times positively and negatively must be a cycle.

Definition 6 (Name-balanced)**.** *A patch sequence in an invertible patch graph is* name-balanced *if its name signature is zero. In other words, every name appears the same number of times positively and negatively in it.*

Definition 7 (Balance-respecting)**.** *An invertible patch graph is* balance-respecting *if every name-balanced sequence is a cycle: that is, whenever S is a name-balanced sequence,* $\text{start}(S) = \text{end}(S)$.

TODO: come up with a better name

Definition 8 (Patch universe)**.** *A* patch universe *is an invertible patch graph that is balance-respecting.*

TOOD: motivate the definition and/or give examples

2.2 Interpretation

To make Definition 8 a bit more concrete, we can use the following interpretation:

- A context encodes a full file tree: directories, files, and the files' contents. Contexts could optionally have additional information; for example, if two unrelated patch sequences arrive at the same file tree, we may choose to distinguish them as different contexts.
- A patch on its own doesn't mean much: it's just a "before" and "after" context, together with an identity for the patch. However, a *patch name* n can be thought of as a partial function on contexts: given any starting context c, look for a patch with name n and starting context c, and if that exists, define $f_n(c)$ to be the ending context of that patch.

  f_n is well-defined because there can't be more than one patch with name n and starting context c, as proved in the following lemma.

Lemma 1. *In any patch universe, if c is a context, n is a name, and p, p' are patches such that* $\text{name}(p) = \text{name}(p')$ *and* $\text{start}(p) = \text{start}(p')$*, then* $\text{end}(p) = \text{end}(p')$.

Proof. Consider the path (p^{-1}, p'). This path is name-balanced, so by the balance-respecting property, $\text{end}(p') = \text{start}(p^{-1}) = \text{end}(p)$. □

2.3 Commuting and permutivity

In this section, we relate our definition of patch universe to the more usual approach of defining patch operations in terms of a commuting relation, by showing that properties that are normally taken as assumptions follow from our definition of patch universe.

Definition 9 (Commute relation)**.** *In any patch universe, we define a binary relation $\leftrightarrow$ on length-2 patch sequences as follows. $(p, q) \leftrightarrow (q', p')$ iff:*

- $\text{start}(p) = \text{start}(q')$ *and* $\text{end}(q) = \text{end}(p')$ *(so the length-4 sequence (p, q, q'^{-1}, p'^{-1}) is a cycle); and*
- $\text{name}(p) = \text{name}(p')$ *and* $\text{name}(q) = \text{name}(q')$.

Lemma 2 (Properties of commuting)**.** *In any patch univers, the relation $\leftrightarrow$ satisfies the following properties for any patches p, q, r, s:*

- *Symmetry: If $(p, q) \leftrightarrow (r, s)$, then $(r, s) \leftrightarrow (p, q)$.*

- *Rotation: If* $(p, q) \leftrightarrow (r, s)$*, then* $(r^{-1}, p) \leftrightarrow (s, q^{-1})$.

These are taken from Jacobson's Definition 2.1 [1]. We leave out the effect-preserving property because in our theory patches do not have the same kind of effect. (See Section 2.2.)

Proof. Symmetry follows immediately from Definition 9.

To prove the rotation property, suppose $(p, q) \leftrightarrow (r, s)$. The following items together imply $(r^{-1}, p) \leftrightarrow (s, q^{-1})$:

- (r^{-1}, p) is a patch sequence. We must show $\text{end}(r^{-1}) = \text{start}(p)$. This is true because $\text{end}(r^{-1}) = \text{start}(r)$ (Definition 2) and $\text{start}(r) = \text{start}(p)$ (definition of $\leftrightarrow$).
- (s, q^{-1}) is a patch sequence. This is true by a similar argument.
- $\text{start}(r^{-1}) = \text{start}(s)$. This is true because $\text{end}(r) = \text{start}(s)$ (since (r, s) forms a patch sequence) and $\text{start}(r^{-1}) = \text{end}(r)$.
- $\text{end}(p) = \text{end}(q^{-1})$. This is true for a similar reason.
- $\text{name}(p) = \text{name}(s)$. This follows directly from $(p, q) \leftrightarrow (r, s)$.
- $\text{name}(r^{-1}) = \text{name}(q^{-1})$. This is true because $\text{name}(r) = \text{name}(q)$ (follows directly from $(p, q) \leftrightarrow (r, s)$) together with the observation that if $\text{name}(r) = \text{name}(q)$ then $\text{name}(r^{-1}) = \text{name}(q^{-1})$ (follows from Definition 2). □

Lemma 3 (Uniquely commuting [1])**.** *In any patch universe, for every length-2 patch sequence* (p, q)*, there is at most one* (r, s) *such that* $(p, q) \leftrightarrow (r, s)$.

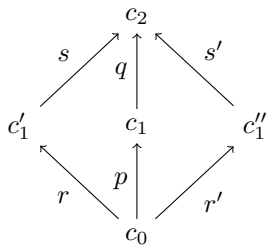
Proof. Suppose $(p, q) \leftrightarrow (r, s)$ and $(p, q) \leftrightarrow (r', s')$. We wish to show $r = r'$ and $s = s'$.

Since a patch is nothing more than a triple (starting context, ending context, name) (Definition 1), it suffices to show that r and r' have the same starting and ending contexts and name, and similarly for s and s'.

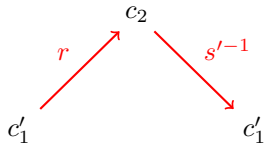
Most of this follows from the definition of $\leftrightarrow$:

- r and r' both have the same name as q.
- s and s' both have the same name as p.
- r and r' both have the same starting context as p.
- s' and s' both have the same ending context as q.

It remains to show that r and r' have the same ending context and that s and s' have the same starting context: in other words, we need to show that the paths (r, s) and (r', s') visit the same intermediate context. In the following diagram, we're trying to show $c'_1 = c''_1$:



Consider the path (s, s'^{-1}):



This is a name-balanced path, so by the balance-respecting property, its starting and ending contexts are the same. Intuitively, this tells us $c_1' = c_1''$ in the above diagram. More carefully, the balance-respecting property gives $\text{start}(s) = \text{start}(s')$. Since (r, s) and (r', s') are paths, we have $\text{end}(r) = \text{start}(s)$ and $\text{end}(r') = \text{start}(s')$, allowing us to also conclude $\text{end}(r) = \text{end}(r')$. □

Lemma 4 (Permutivity [1])**.** *Suppose S is a patch sequence, and the patch sequences S' and S'' are both derived from S by applying a sequence of commute operations amounting to the same overall permutation σ. (TODO: make this more precise; for now, note that we're loosely following [1, Definition 4.4], where $\sigma = s_1 \cdots s_k = t_1 \cdots t_k$.)*

Then $S' = S''$.

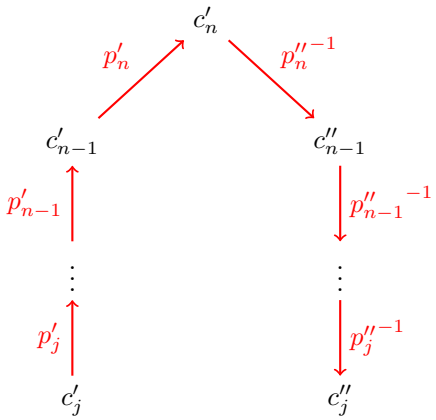
Proof. Denote the patches in the sequences as $S = (p_1, \ldots, p_n)$, $S' = (p_1', \ldots, p_n')$ and $S'' = (p_1'', \ldots, p_n'')$, and the contexts as $(c_0, \ldots, c_n)$, $(c_0', \ldots, c_n')$ and $(c_0'', \ldots, c_n'')$.

Since commuting two patches swaps their names, the permutation σ permutes the order of the names in the sequence: $\text{name}(p_i') = \text{name}(p_{\sigma(i)})$.

Since S' and S'' have the same permutation σ, they have the same sequence of names: $\text{name}(p_i') = \text{name}(p_i'')$ for all $i \in \{1, \ldots, n\}$. This property is sufficient for the rest of the proof.

Let $j \in \{0, \ldots, n\}$. We wish to show $c_j' = c_j''$. If we can show this for all j, it follows that the two patch sequences are the same, since a patch is fully identified by its name and starting and ending contexts (Definition 1) and we've already shown $\text{name}(p_j') = \text{name}(p_j'')$.

We'll follow a similar argument as in the proof of Lemma 3. Consider the path $(p_{j+1}', p_{j+1}', \ldots, p_n', {p_n''}^{-1}, {p_{n-1}''}^{-1}, \ldots, {p_{j+1}''}^{-1})$:

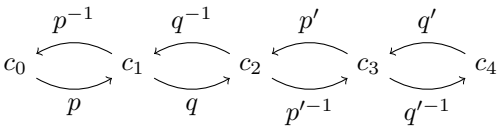


This is a name-balanced path, so its starting and ending contexts are the same: $c'_j = c''_j$. (Note: If $j = n$, then the path is empty, but the argument still works.) □

2.4 Possible equivalence to other formulations

It would be nice to be able to prove that other formulations of patch theory are equivalent to our formulation of a patch universe. Section 2.3 gives evidence that any patch universe satisfies properties required in other formulations. What about the other direction? Is it true that anything satisfying the properties in Section 2.3 is also a patch universe according to Definition 8?

Here's a possible counterexample. Create a patch graph that looks like this:



where name(p) = name(p') and name(q) = name(q'). Define the commute relation to be empty: there exists no combination of patches such that $(p_1, p_2) \leftrightarrow (p_3, p_4)$. The fact that nothing ever commutes means the properties described in Lemmas 2, 3 and 4 are trivially satisfied. However, this is not a patch universe according to Definition 8, because the name-balanced sequence (p, q, p'^{-1}, q'^{-1}) is not a cycle.

So, at first glance it seems that our definition is stronger than one derived just from Lemmas 2, 3 and 4. However, there's something strange about our counterexample: patches p and p' have the same name, but there's no reason for that. Someone using an implementation of our theory will never be able to apply a sequence of commuting operations to transform patch p into p'.

Conjecture 1. *There exists some reasonable condition about patches with the same names — for example, whenever any two patches have the same name, there's some sequence of commute operations that turns one into the other —*

such that that property and the properties described in Lemmas 2, 3 and 4 together imply that what we have is a patch universe as defined by Definition 8.

3 Context addresses

Definition 10 (Context address)**.** *A* context address *within a patch universe is a pair* (s, A) *where* s *is a context (called the sequence's* starting context*) and* A *is a signed name multiset (Definition 5).*

Definition 11 (Context pointed to by an address)**.** *A context address* (s, A) points to *a context* e *if there exists a patch sequence* S *with name signature* A *such that* $\text{start}(S) = s$ *and* $\text{end}(S) = e$*.*

Lemma 5. *A context address* (s, A) *points to at most one context.*

Proof. Suppose (s, A) points to context e and also points to context e'.

Then there exist patch sequences S and S' which both have name signature A and both have starting context s, and such that $\text{end}(S) = e$ and $\text{end}(S') = e'$.

Then make a new path by concatenating the reverse of S with S'. This new path starts at e and ends at e'. The path is name-balanced, because the name signature of the reverse of S is $-A$, which cancels out the name signature A of S'. Since the path is name-balanced, by the balance-respecting property of the patch universe, we have $e = e'$. □

4 Tree repositories for handling conflicts without conflictors

Sometimes patches *conflict*: we have patches p_1 and p_2 but there is no sequence of patches containing both. Darcs handles this by introducing a new kind of patch called a *conflictor*.

In this section, we try something else: we require the user to resolve the conflict by *deactivating* one or both of the conflicting patches, and optionally recording a new patch to replace them.

We begin with an informal overview in Section 4.1, then give a more complete description in the following sections.

4.1 Examples and motivation

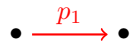
Informally, we think of a *tree repository* as a tree of patches rooted at empty_repo, with one path from the root selected as the "chosen resolution". The other branches store patches that were discarded due to conflicts.

4.1.1 Resolving a conflict

For example, suppose patch p_1 creates the following text file:

```
int x;
x = ask("Hwo old are you?");
print("That's ", 7 * x, " in dog years.");
```

Our repository starts out with just p_1:



Alice sees the spelling mistake on line 2, and creates patch p_2 to fix it. In Darcs notation:

```
hunk ./dog_years 2
-x = ask("Hwo old are you?");
+x = ask("How old are you?");
```

Meanwhile, Bob makes a different change to the same line, recording it as patch p_3:

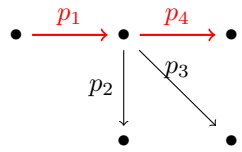
```
hunk ./dog_years 2
-x = ask("Hwo old are you?");
+x = ask("Hwo old are you in human years?");
```

When Alice pulls Bob's change, there's a problem: p_2 and p_3 conflict. In Darcs, this would be handled by creating a *conflictor* which returns the repository to a state before either patch was applied. Alice would then record a new patch to replace both changes.

With tree repositories, Alice can handle the conflict a few different ways.

Example 4.1 (Darcs-style conflict resolution)**.** One way is to *deactivate* both conflicting patches and create a replacement patch p_4 incorporating both changes:

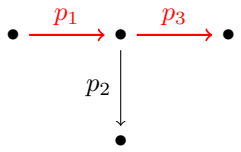
```
hunk ./dog_years 2
-x = ask("Hwo old are you?");
+x = ask("How old are you in human years?");
```



p_2 and p_3 stay in the repository in case we need them later, but we put them off to the side and draw them in a different colour to show they are deactivated.

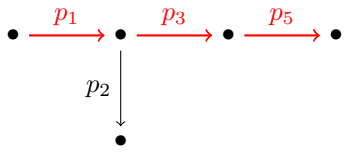
This way of resolving the conflict is similar to what happens with Darcs, except the new patch p_4 does not depend on either of the conflicting changes. (The full *combination* of changes Alice made — deactivating p_2 and p_3 and creating p_4 — does depend on both p_2 and p_3, so is very similar to a Darcs conflictor.)

Example 4.2 (Resolving a conflict by keeping one side)**.** Alice could also choose to deactivate p_2 and keep p_3 (or vice versa):



Example 4.3 (Rebasing)**.** Finally, Alice might decide to replace p_2 with a new patch p_5 to be applied after p_3:

```
hunk ./dog_years 2
-x = ask("Hwo old are you in human years?");
+x = ask("How old are you in human years?");
```



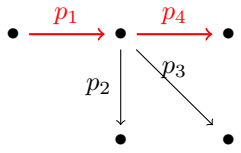
This is a bit like rebasing p_2.

4.1.2 Merging after a conflict resolution

After Alice resolves the conflict, she'll want to share her changes. In Darcs, merging two repositories results in a repository with all the patches from both (possibly re-ordered or transformed into conflictors). How should we merge tree repositories, which can have both active and deactivated patches?

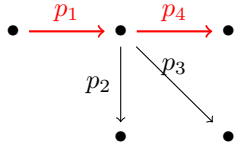
As our first example, suppose Alice has merged in Bob's changes, resolving the conflict as in Example 4.1, and Bob pulls her changes back. Since Bob hasn't made any additional changes of his own, his repository should end up in the same state as Alice's:

Example 4.4 (Merging back a conflict resolution)**.**
Merging



and

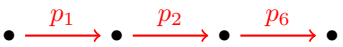
produces



(In general, if repository A has all the changes from B, then when B pulls from A their repository should end up in the same state.)

What kind of merge algorithm could result in this behaviour? Before we answer that, let's see another example. Suppose Carol cloned Alice's repository early on, when Alice only had p_1 and p_2. Carol then went on to record a new patch, p_6, which adds a new line to the end of the program:

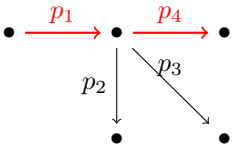
```
hunk ./dog_years 4
+print("Goodbye!");
```



Note that p_6 is independent of all the other patches we've seen (except p_1); that is, it can be commuted with any of them. What should happen when Carol pulls from Alice's merged repository from Example 4.1?

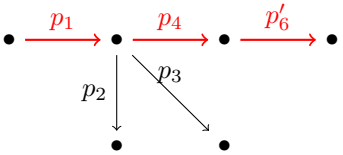
The final result shouldn't include p_2 — Alice deliberately deactivated that patch in order to resolve a conflict. On the other hand, there's no reason not to include p_6, if it can be commuted to co-exist with the remaining patches. So we should end up with:

Example 4.5.
Merging



and

produces



where p'_6 has the same name as p_6.

We aren't quite ready to describe the merge algorithm yet, but these examples illustrate an important principle:

Principle. A repository R can be identified by its set of *active* patch names $A(R)$ and *deactivated* patch names $D(R)$, in the sense that two repositories R and R' are *equivalent* iff $A(R) = A(R')$ and $D(R) = D(R')$. Merging should result in simple and predictable behaviour with respect to these sets.

We formally define equivalence in Definition 14.

Here's a rule we could follow which is consistent with all the examples seen so far:

A possible merge rule. When R_1 and R_2 are merged, producing R_3, all deactivated patches must stay deactivated: $D(R_3) \supseteq D(R_1) \cup D(R_2)$. (The particular choice is at the discretion of the user, as in the examples in Section 4.1.1.) The remaining patches from either repo will be active: $A(R_3) \supseteq (A(R_1) \cup A(R_2)) \setminus D(R_3)$, with new patches added at the discretion of the user.

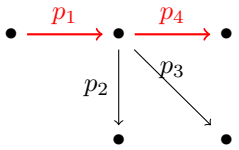
In other words: patches never disappear, but once a patch is deactivated, it never becomes active again.

But wouldn't it be useful to be able to re-activate a patch? It turns out we can allow this, as shown in the next section.

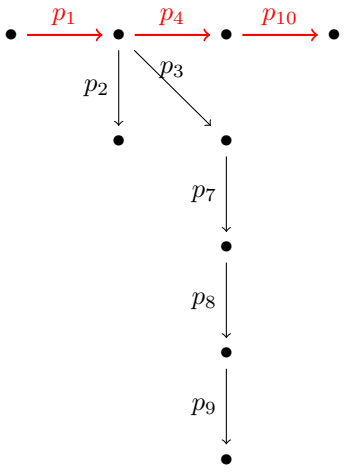
4.1.3 Re-activating a patch

Let's return to Example 4.1, where Alice resolved a conflict by de-activating one of Bob's patches. Suppose that in the meantime, Bob has recorded a large number of important changes which all depend on his original p_3:

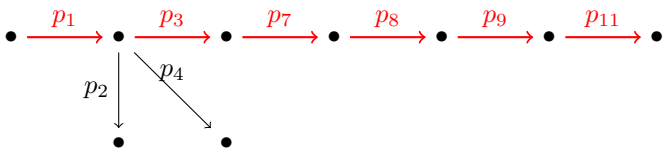
Recall that Alice's repository looks like this:



Now, when Alice pull's Bob's repository, she feels regret: by deactivating Bob's patch p_3, she has made it impossible for Bob's important work in patches p_7, p_8 and p_9 to be applied. One solution would be to write a new patch p_{10} which rebases all of Bob's work on top of p_4:



One problem with this solution is that if Bob continues his work on top of p_9 in the meantime, that will result in more conflicts to resolve. Alice would rather re-activate Bob's p_3 so that his later patches can remain active, and instead de-activate her own smaller change, p_4, and record a replacement p_{11}:



Fortunately, in tree repositories, it is possible to re-activate a patch in this way, *as long as some new patch in the repository depends on it.* To make sure this won't get us in any trouble, we define a partial order on repositories (Definition 16) that ensures that de-activation and re-activation are "non-destructive", meaning they are operations that result in larger repositories according to our order.

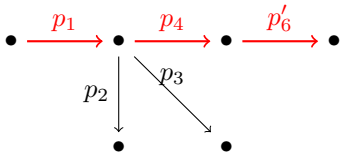
The "new patch" that permits an older patch to be re-activated does not itself need to be active. (Indeed, we must allow it to be inactive: otherwise, we would not be able to handle the situation where first a patch p_1 is re-activated by a patch p_2 that depends on it, and p_2 is later deactivated (but not p_1).) The ability to reactivate using inactive patches could be useful if the user wishes to reactivate a patch but does not wish to add any new patches. "Inactive tags" could be used for this, empty patches that do nothing more than depend on the patch to be reactivated, and are never active.

4.1.4 A tree repository is a set of tree patches

At this point, the reader may feel uneasy about the following two things.

First, the principle in Darcs that *a repository is a set of patches* has an benefits that we seem to have lost. It allows us to reason about repositories by comparing the patches that are in one, the other, or both. If Alice's repository is the patch sequence p_1, p_2 and Bob's is p_1, p_3, the situation is clear: Bob has one change Alice doesn't have, and vice-versa. Merging is simply the process of moving all the missing patches of one repository to the other. It also enables certain operations: to undo the last thing the user did, just remove the most recently-added patch; when merging repositories, to pull in some but not all changes, simply pull in the desired patches and ignore the others. The situation with tree repositories is less clear.

Second, our history is missing information. Take, for example, Carol's merged repository from Example 4.5:



Why were p_2 and p_3 deactivated? Which patch(es) replaced them? If you've been reading the story closely, it turns out they were deactivated as part of a conflict resolution, and p_4 is their replacement. But the tree view doesn't give us enough information to deduce that, so the historical record feels incomplete.

The solution to both of these problems is *tree patches.* A tree patch is a

collection of patch names together with, optionally, a set of *deactivation markers* causing other names to be deactivated. For example, Alice's conflict resolution from Example 4.1 would be represented as { deactivate n_2, deactivate n_3, add n_4 }, where n_2, n_3, n_4 are the names of p_2, p_3, p_4, respectively. Associating a tree repository with a set of tree patches lets us recover at least some of the benefit of the Darcs principle that a repository is a set of patches.

Tree patches are not essential to tree repositories, so we will start our development without them, and then show how to add them on in Section 4.2.11.

4.1.5 Application: keeping amendment history

The preceding examples show how tree repositories can be used to manage conflicts between patches. Another application that may be possible is keeping track of a history of amendments.

In Darcs, patches can be edited using the `darcs amend` command, but it is inadvisable to do this if any other repositories have a copy of your patch, because the command creates an entirely new patch which conflicts with the original.

Tree repositories might admit a simple mechanism for amending a patch while keeping the original in the repository: simply de-activate the old version of the patch and add the new one in its place. This would also have the advantage of preserving history: if an amendment is done in error, the original version of the patch could be recovered. An entire sequence of patches representing rough work toward a goal could similarly be "squashed" into a single tidy patch that replaces them, allowing the tidier representation and the sequence of rough work that led to it to coexist in the repository's history.

It may similarly be possible to deactivate a patch with no replacement, as a history-preserving way to remove it when it's no longer wanted.

There are many details that would need to be worked out, and it is beyond the scope of this document. We can't claim with certainty that it would be feasible.

This functionality might be compared to Mercurial's evolve extension.

4.2 Mathematical formulation and algorithms

In this section, we mathematically define tree repositories and operations that can be performed on them.

4.2.1 Setting and assumptions

We assume there is an underlying patch universe $G' = \mathrm{addinv}(G)$ (Definition 8). We will not use inverses, so within this section, "names" and "patches" refer to names and patches in G.

Let C, N, P be the contexts, names and patches in G; i.e. $G = (C, N, P)$.

We make the following additional assumption, which in short means that there are *conflict* and *dependency* relations between patch names which determine the possible orders for patches.

Assumption 4.1 (Simple dependencies)**.** There exist:

- a partial order required_by $\subseteq N \times N$; required_by(n_1, n_2) means n_2 *depends on* n_1.
- a relation conflict $\subseteq N \times N$; and
- a context empty_repo $\in C$ (which we will use as a universal starting context)

such that the following properties hold. (Note that we don't assume required_by and conflict can be computed just from the names. We only assume the relations exist.)

1. conflict is symmetric: $\forall n_1, n_2 \in N.\ \text{conflict}(n_1, n_2) \iff \text{conflict}(n_2, n_1)$.
2. required_by and conflict are consistent, in the following sense. For all names $n_1, n_2 \in N$, at most one of the following is true:
   - required_by(n_1, n_2)
   - required_by(n_2, n_1)
   - conflict(n_1, n_2)
3. required_by determines when patches commute: a length-two patch sequence p_1, p_2 commutes iff $\neg$ required_by(name(p_1), name(p_2)).
4. conflict determines when patches can be merged. That is, for any two patches p_1 and p_2 with start(p_1) = start(p_2), there exists a patch p_2' with name(p_2') = name(p_2)) and start(p_2') = end(p_1) iff $\neg$ conflict(name(p_1), name(p_2)).
5. Every patch sequence $S = p_1, \ldots, p_k$ with starting context start(S) = empty_repo and names $n_1, \ldots, n_k$ satisfies all of the following:
   - No name is repeated.
   - None of the names conflict: $\forall 1 \leq i < j \leq k.\ \neg\,\text{conflict}(n_i, n_j)$.
   - All dependencies are satisfied: for every name n_i in the sequence, and every other name $n \in N$, if required_by(n, n_i), then $n = n_j$ for some $j \leq i$.

Definition 12 (Dependency and reverse dependency sets)**.** *The* dependency set *of a name $n_1 \in N$, denoted* dependencies(n_1)*, is the set of all names $n_2 \in N$ such that* required_by(n_2, n_1).

For a set of names S, dependencies$(S) = \bigcup_{n \in S}$ dependencies(n).

The reverse dependency set *of $n_1 \in N$, denoted* reverse_dependencies(n_1)*, is the set of all names $n_2 \in N$ such that* required_by(n_1, n_2).

Notes:

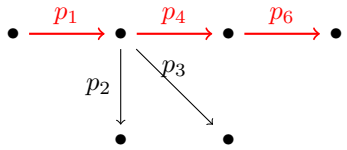
- A name is always in its own dependency set, since required_by is a partial order and therefore satisfies required_by(n, n) for all names n.
- This set is finite for all names that we see in practice: if it were infinite, the name could not appear in any patch sequence starting at empty_repo.

Aside. A weak converse of the last property of Assumption 4.1 follows from the assumption. Suppose $n_1, \ldots, n_k$ is a sequence of names with no repeats or conflicts and where dependencies are satisfied. Suppose further that each name n_i can be realized as a patch p_i appearing in a patch sequence starting at empty_repo (not necessarily the same sequence). Then using commute operations, it is possible to produce a patch sequence $p'_1, \ldots, p'_k$ with starting context empty_repo and names $n_1, \ldots, n_k$. We will not be using this converse directly, and do not include a proof.

4.2.2 Repositories and signatures

Definition 13. *A* tree repository *is a pair $R = (T, C)$, where T is a finite set of patch sequences with starting context* empty_repo *and C is a patch sequence with starting context* empty_repo*. C is called the* chosen resolution *of R.*

There are no explicit "trees" in this definition, but we can relate it to the tree-like presentation in Section 4.1 as follows. C is the highlighted, active path of patches in the tree. The rest of the tree is the union of the paths in T. For example, this repository:



could be represented as $(\{(p_1, p_2), (p_1, p_3)\}, (p_1, p_4, p_6))$. We use this representation instead of a tree to simplify the technical presentation in this section, but a practical implementation could store the patches in a tree to reduce redundancy.

A repository can be identified by its set of active patches and full set of patches.

Definition 14. *A* tree repository signature*, or* signature*, is a pair of sets of names (P, A) where $A \subseteq P$. Names in P are called* present *and names in A are called* active*.*

The signature of a tree repository $R = (T, C)$*, denoted* signature(R)*, is the pair (P, A) where P is all names that appear in T or C, and A is names that appear in C.*

Two tree repositories are equivalent *if they have the same signature.*

Note that not every "tree repository signature" is realizable as the signature of a tree repository: for example, a signature could contain active patches that

conflict with each other. For the most part, the signatures we look at will be realizable, but it will be useful to allow impossible signatures when reasoning about merges and tree patches (see Lemma 9).

For example, the signature of $(\{(p_1, p_2), (p_1, p_3)\}, (p_1, p_4, p_6))$ is $(\{n_1, n_2, n_3, n_4, n_6\}, \{n_1, n_4, n_6\})$, following the convention that n_i is the name of patch p_i.

Definition 15. *A signature (P, A) is* complete *if P and A are closed under dependencies. More precisely, for every $n \in P$,* dependencies$(n) \subseteq P$*, and for every $n \in A$,* dependencies$(n) \subseteq A$*.*

Lemma 6. *Every tree repository's signature is complete.*

Proof. Let (T, C) be a tree repository with signature (P, A).

Let $n_1 \in P$ and suppose required_by(n_2, n_1). We must show $n_2 \in P$. Indeed, n_1 appears in some patch sequence in T (by the definition of signature). By Assumption 4.1, n_2 appears earlier in that same sequence, and is therefore also in P.

Similarly, if $n_1 \in A$ and required_by(n_2, n_1), then n_1 appears in C and so n_2 also appears in C and is therefore also in A. □

4.2.3 Ordering signatures

Definition 16 (Partial order on signatures)**.** *We define a relation $\preceq$ on signatures as follows: $(P_1, A_1) \preceq (P_2, A_2)$ iff $P_1 \subseteq P_2$ and*

$$A_2 \subseteq A_1 \cup \bigcup_{n \in P_2 \setminus P_1} \text{dependencies}(n).$$

In other words, all names present in (P_1, A_1) are present in (P_2, A_2), and every active name in (P_2, A_2) was already active in (A_1, P_1), or is a dependency of a "newly present name, i.e. a name in $P_2 \setminus P_1$. (Recall that under Definition 12, a name is always in its own dependency set, i.e. $n \in$ dependencies(n).)

Theorem 1. *$\preceq$ is a partial order on signatures.*

Proof. Let (P_1, A_1), (P_2, A_2), (P_3, A_3) be signatures.

Reflexivity. It's straightforward to verify that $(P_1, A_1) \preceq (P_1, A_1)$.

Antisymmetry. Suppose $(P_1, A_1) \preceq (P_2, A_2)$ and $(P_2, A_2) \preceq (P_1, A_1)$. To prove the antisymmetry property of $\preceq$, we should show that $(P_1, A_1) = (P_2, A_2)$.

Since $P_1 \subseteq P_2$ and $P_2 \subseteq P_1$ we have $P_1 = P_2$ (antisymmetry of $\subseteq$).

Now, $P_2 \setminus P_1$ is empty, so the definition of $(P_1, A_1) \preceq (P_2, A_2)$ gives us

$$A_2 \subseteq A_1 \cup \bigcup_{n \in P_2 \setminus P_1} \text{dependencies}(n) = A_1.$$

$A_1 \subseteq A_2$ follows from a similar argument.

Transitivity. Suppose $(P_1, A_1) \preceq (P_2, A_2)$ and $(P_2, A_2) \preceq (P_3, A_3)$. We must show $(P_1, A_1) \preceq (P_3, A_3)$.

$P_1 \subseteq P_3$ follows from transitivity of $\subseteq$.

It remains to show

$$A_3 \subseteq A_1 \cup \bigcup_{n \in P_3 \setminus P_1} \text{dependencies}(n).$$

We know

$$A_2 \subseteq A_1 \cup \bigcup_{n \in P_2 \setminus P_1} \text{dependencies}(n)$$

and

$$A_3 \subseteq A_2 \cup \bigcup_{n \in P_3 \setminus P_2} \text{dependencies}(n);$$

combining these, we have

$$\begin{aligned} A_3 \subseteq & A_1 \cup \left(\bigcup_{n \in P_2 \setminus P_1} \text{dependencies}(n) \right) \cup \bigcup_{n \in P_3 \setminus P_2} \text{dependencies}(n) \\ = & A_1 \cup \bigcup_{n \in (P_2 \setminus P_1) \cup (P_3 \setminus P_2)} \text{dependencies}(n) \\ = & A_1 \cup \bigcup_{n \in P_3 \setminus P_1} \text{dependencies}(n) \end{aligned}$$

where $(P_2 \setminus P_1) \cup (P_3 \setminus P_2) = P_3 \setminus P_1$ follows from $P_1 \subseteq P_2 \subseteq P_3$. □

4.2.4 Destructive and non-destructive operations

In Darcs, patches can be added to a repository using the `darcs pull` and `darcs record` commands, and these changes are propagated using `darcs push`. A user who only uses these three commands will find that Darcs is *monotone*, in the sense that changes are only accumulated, and never lost. We call these changes *non-destructive*.

On the other hand, if a user destroys or changes patches using commands like `darcs obliterate` or `darcs amend`, those changes are not propagated when running `darcs push`: for example, `darcs push` will not obliterate a patch on the remote repository just because it was obliterated in the local one. We call these changes *destructive*.

A similar distinction can be defined for tree-like repositories using the partial ordering $\preceq$. We call any operation that makes a tree repository's signature *larger* (or equal) under the $\preceq$ relation *non-destructive*, and other operations *destructive*. In the following sections, we will ensure that operations like recording a new patch or merging changes from another repository and resolving conflicts are all non-destructive.

The merge procedure in Section 4.2.9 ensures that non-destructive changes are propagated when merging repositories, in the sense that if Alice and Bob start out with the same repository, then Alice makes a non-destructive change and then Bob pulls from Alice, Bob's repository ends up in the same state as Alice's. TODO: argue why that's true.

4.2.5 Recording a new patch

Suppose (T, C) is a tree repository, and the user would like to record a new patch p. Recall that C is the sequence of patches the user has chosen to include in the repository (the *chosen resolution*). Therefore, we assume $\text{start}(p) = \text{end}(C)$.

To record the patch p, we simply replace the repository with (T, Cp).

Lemma 7. *Recording a new patch (with name not present in the existing repository) is a non-destructive operation (as defined in Section 4.2.4).*

Proof. Let $(P, A) = \text{signature}((T, C))$ and $(P', A') = \text{signature}((T, Cp))$. We must show $(P, A) \preceq (P', A')$.

$P' = P \cup \{\text{name}(p)\}$, so $P \subseteq P'$ as required.

$A' = A \cup \{\text{name}(p)\}$, and since $\text{name}(p) \in P' \setminus P$, we have

$$\text{name}(p) \in \text{dependencies}(\text{name}(p)) \subseteq \bigcup_{n \in P' \setminus P} \text{dependencies}(n),$$

so

$$A' \subseteq A \cup \bigcup_{n \in P' \setminus P} \text{dependencies}(n)$$

as required. □

4.2.6 Deactivating a patch

Here we show how to deactivate a patch and the patches that depend on it. This is used in the merge algorithm in Section 4.2.9 to handle conflicts. A user might also want to apply this operation directly, when the effect of the patch is no longer desired.

Input. We're given a tree repository (T, C) and the index of a patch p in C.

Algorithm. We commute the patch p to the end of C and drop it. Along the way, we may also be forced to commute other patches that depend on p to the end too; we also drop those when we drop p. Details are omitted; this will be similar to how Darcs obliterates a patch and other patches that depend on it.

Let C' be the result of this operation. Then the final repository we return is $(T \cup \{C\}, C')$.

The operation is non-destructive. Let $(P, A) = \text{signature}((T, C))$ and $(P', A') = \text{signature}((T \cup \{C\}, C'))$. Then $P \subseteq P'$, because every name present in T is also in $T \cup \{C\}$. Also, $A' \subseteq A$, since every name in C' is also in C. So $(P, A) \preceq (P', A')$: the operation is non-destructive (Section 4.2.4).

Efficiency. The algorithm will perform $O(n^2)$ commute operations, where n is the length of C.

The repository will grow on disk, since it now needs to store the full original tree T, the original chosen resolution C, and the new chosen resolution C'. It may be possible to reduce the size by removing redundant patches. See also Section 4.2.10.

4.2.7 Reactivating a patch

As promised in Section 4.1.3, it is possible to re-activate a previously deactivated patch p.

First, it is likely the user will need to de-activate patches p conflicts with, as well as patches that conflict with p's dependencies if those are also being re-activated. Section 4.2.6 explains how to deactivate these patches and the patches that depend on them.

Once the tree repository's chosen resolution no longer has any patches that conflict with p or its dependencies, re-activating p consists of two steps:

- Insert p and its dependencies into the chosen resolution. We already have a copy of them somewhere: recall that a tree repository is a pair (T, C); T and C together contain all active and deactivated patches in the repository. Some commuting may be needed. (TODO: elabourate?)
- Create a new patch that depends on p. Otherwise, the resulting repo wouldn't be "larger" than the original under $\preceq$; in other words, this would be a destructive operation. If the user does not actually have changes depending on p that they wish to record, it is sufficient to add a new patch depending on p to one of the patch sequences in T. (For example, this could be a tag that depends on p and does nothing else.)

4.2.8 Merging as an abstract operation

As shown in Section 4.1.1, there are many ways to merge repositories when there are conflicts. In this section, we set a simple and very loose restriction on what counts as a "merge". In the next section, we will show how to produce such a merge.

Definition 17. *A tree repository R_3 is a* merge *of tree repositories R_1 and R_2 if* $\text{signature}(R_1) \preceq \text{signature}(R_3)$ *and* $\text{signature}(R_2) \preceq \text{signature}(R_3)$.

This definition ensures that merging is a non-destructive operation, and also that it propagates non-destructive operations as prescribed in Section 4.2.4.

4.2.9 Implementing merging

Here we will describe how to merge two repositories and allow the user to handle conflicts. This, together with recording new patches (Section 4.2.5), should be enough to create a usable version control system.

Input. We are given two tree-like repositories $R_1 = (T_1, C_1)$ and $R_2 = (T_2, C_2)$. (For exmaple, R_1 might be the user's local repository, and R_2 a repository they're pulling from.)

Step 1: compute signatures. Let $(P_1, A_1) = \text{signature}(R_1)$ and $(P_2, A_2) = \text{signature}(R_2)$. These are easily computed by scanning all the names in T_1, C_1, T_2 and C_2 respectively. (An implementation could also store a pre-computed signature with every repository.)

Step 2: find the new chosen resolution. The goal of Step 2 is to produce a sequence of patches C that will be the chosen resolution of our merged repository. If possible, we'd like C to include all the patch names from $A_1 \cup A_2$, but that may not be possible due to conflicts.

For this step, we will adapt the merging process described in Section 8 of I. Lynagh's *Camp Patch Theory* [2].

Step 2a: preparation. We begin by permuting C_1 and C_2 so that they begin with all the names that are active in both repositories, i.e. in $A_1 \cap A_2$).

By Lemma 6, A_1 and A_2 are both closed under dependencies, and therefore so is $A_1 \cap A_2$. That is, for any $n_1 \in A_1 \cap A_2$ and $n_2 \in \text{dependencies}(n_1)$, we have $n_2 \in A_1 \cap A_2$.

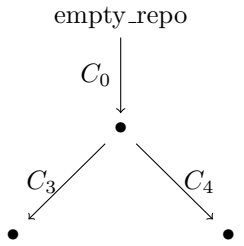
Start with the sequence C_1.

Let n_1 be the first name in $A_1 \cap A_2$ that appears in the sequence C_1. Then n_1 has no dependencies other than itself: $\text{dependencies}(n_1) = \{n_1\}$. Otherwise, say $\text{required\_by}(n', n_1)$, n' would come before n_1 in C_1 and so n_1 would not have been the first.) By Assumption 4.1, the patch with name n_1 can be commuted to the beginning of the sequence C_1 to produce a sequence C_1'.

Now, let n_2 be the next name after n_1 that appears in C_1'. By a similar argument, it has no dependencies other than possibly n_1 and n_2, and so C_1' can be commuted so it starts with names n_1 and n_2.

By repeating this process, we can transform C_1 into a sequence that starts with the names in $A_1 \cap A_2$, and by a similar process we can permute C_2 so that it starts with that same prefix.

Call the common prefix C_0, and the remaining parts C_3 and C_4. The are arranged like this:



In other words, we've permuted the two chosen resolutions to have the form C_0C_3 and C_0C_4, where C_3 and C_4 have no names in common.

Step 2b: deactivating. If a patch has been deactivated in one repository, we must make sure it is not active in the final repository, unless a patch that is only present in the other repository depends on it. (This is required in order for merging to be non-destructive.)

For every patch p in C_3, check if it has been deactivated in R_2: that is, if $\text{name}(p) \in P_2 \setminus A_2$. If so, we must compute whether any name in $P_1 \setminus P_2$ depends on $\text{name}(p)$. To do this, we try to commute p to the end of C_3, and also try to commute p to the end of every patch sequence in T_1 where it appears. If we are able to do so, or if the only patches blocking p from being commuted to the end also have names in $P_2 \setminus A_2$, then p (along with those other patches) must be dropped from the end of C_3.

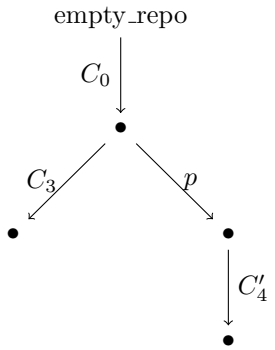
We follow the same process to drop patches from C_4.

We believe this can be done with $O(|P_1| + |P_2| + i_1|P_1| + i_2|P_2|)$ (attempted) commute operations, where $i_1 = |A_1 \cap (P_2 \setminus A_2)|$ is the number of potentially-deactivated patches from R_1, and similarly $i_2 = |A_2 \cap (P_1 \setminus |_1)|$.

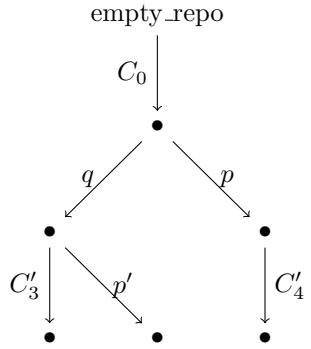
(TODO: explain better and verify the runtime.)

Step 2c: combining, and handling conflicts. If C_4 from the previous step is empty, then we are done with Step 2: we can take the patch sequence C_0C_3 as our chosen resolution, and it will include all names in $A_1 \cup A_2$, which is the ideal outcome.

Assume now that C_4 is not empty. Then $C_4 = pC_4'$ for some patch p; in a picture:

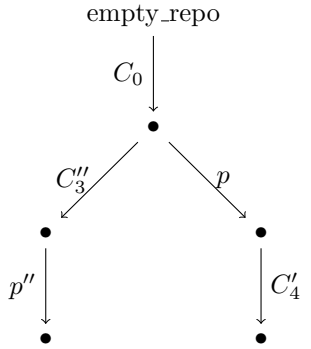


We then attempt to merge p with each patch in C_3 in turn. For example, if q is the first patch in C_3 ($C_3 = qC_3'$), successfully merging p and q would give:



Repeating this operation, we can attempt to move p all the way to the end of the sequence C_3.

This will fail if p conflicts with a patch in C_3. Every time we encounter such a conflicting patch, delete it from the sequence, along with every patch that depends on it, using the method described in Section 4.2.6. We end up with this:

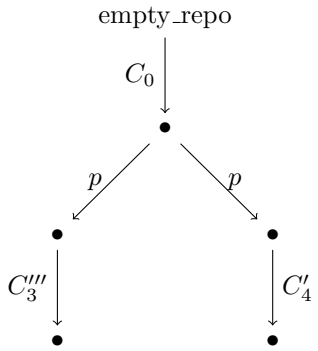


where C_3'' is C_3 with all patches that conflicted with p (and their dependencies) removed.

At this point, we show the user the list of patches that conflicted with p, and offer them a choice: they may either deactivate them (and keep C_3'') or they may instead deactivate p and all the patches in C_4 that depend on it. (If there were no conflicts, the choice can be skipped.)

If they choose to deactivate p, we follow the procedure in Section 4.2.6 to produce a sequence C_4'' without p.

If they choose to keep p, then we commute p back to the start of C_3'' (possible by Assumption 4.1), producing:



Now we can extend C_0 to $C_0 p$ and continue merging C_3''' and C_4'.

Step 3: Replacement patch and output Step 2 produces a patch sequence C_{merge} containing the patches the user has chosen to keep from C_1 and C_2.

At this point, if any patches were deactivated, the user may wish to record a new patch to replace their effects (see, for example, Example 4.1). This optional patch is added to the end of C_{merge}.

Finally, the algorithm outputs $R_3 = (T_1 \cup T_2 \cup \{C_1, C_2\}, C_{\text{merge}})$.

Non-destructive. Let $(P_1, A_1) = \text{signature}(R_1)$, $(P_2, A_2) = \text{signature}(R_2)$ and $(P_3, A_3) = \text{signature}(R_3)$.

By design, $T_1 \cup T_2 \cup \{C_1, C_2\}$ contains all the patch names present in either original repository, so $P_1 \subseteq P_3$ and $P_2 \subseteq P_3$.

The deactivations from Step 2b ensure that $A_3 \subseteq A_i \cup \bigcup_{n \in P_3 \cap P_i} \text{dependencies}(n)$ for $i \in \{1, 2\}$.

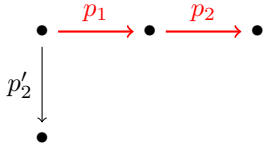
So $(P_1, A_1) \preceq (P_3, A_3)$ and $(P_2, A_2) \preceq (P_3, A_3)$: this procedure computes a merge (Definition 17).

4.2.10 Storing tree repositories efficiently

The operations in Sections 4.2.6 and 4.2.9 add new patch sequences to the T part of a tree repository (T, C). In a naïve implementation which stores each patch sequence separately, this would require storing an additional $O(n)$ patches after each operation, where n is the length of the chosen resolution C.

We can mitigate this by storing T as a tree instead of a set of sequences, as shown throughout the exposition in Section 4.1. This way, common prefixes would only need to be stored once.

Other simplifications may be possible: for example, in this tree repository:



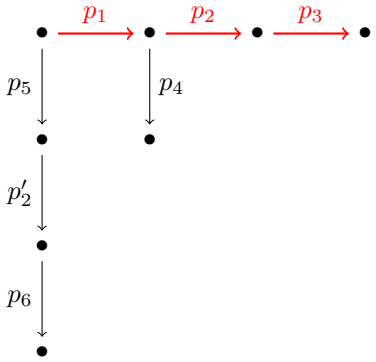
if $\text{name}(p_1') = \text{name}(p_1)$, there's no reason to keep p_1'. It can be simplified to:

These repositories are equivalent (Definition 14). In general, operations that result in equivalent repositories might be used to reduce the size.

It is tempting to try to reduce the tree so that no patch name appears twice:

Definition 18. *A tree repository is* non-redundant *if (when viewed as a tree) none of the patches in the tree have the same name.*

Unfortunately, it is not always possible to make a repository non-redundant. For example:



Suppose name(p_2) = name(p'_2), p_1 and p_5 conflict, p_2 and p_4 conflict; p_3 depends on p_1 and p_2; and p_6 depends on p_5 and p'_2. Then there is no tree with this set of patch names which doesn't contain any duplicates. (TODO: include proof; it's on the `darcs-users` list dated 2020-11-19.)

4.2.11 Tree patches

As explained in Section 4.1.4, it would be good to have a notion of a "tree patch" which can combine a set of changes to a tree repository, and such that each repository can be thought of as nothing more than a collection of tree patches.

Definition 19. *A* tree patch *is a pair* (P, D) *where* P *and* D *are sets of names and* $D \subseteq P$*.*

Informally, the effect of the tree patch (P, D) is to deactivate all the names in D, and to add or re-activate all the names in dependencies$(P) \setminus D$. Its precise effect is given by Definition 20.

(Definition 19 could be expanded to allow for metadata like a user-supplied description.)

Definition 20 (Signature of a set of tree patches)**.** *The* signature *of a set* S *of tree patches is* signature$(S) = (P, P \setminus D)$*, where*

$$P = \bigcup_{(P',D') \in S} P'$$

and

$$D = \bigcup_{(P',D')\in S} D' \cap R(P, P')$$

where $R(P, P') = \{n \in P \mid \text{reverse\_dependencies}(n) \cap P \subseteq \text{dependencies}(P')\}$.

In other words, a name n is active as long as there are no tree patches (P', D') that explicitly deactivate n and are also "aware" of all of the other patches that depend on n (in the sense that those patches are in P').

Defining D in this way allows successively applied tree patches to flip primitive patches back and forth between active and de-activated states, as long as every time a patch is re-activated, a new name n can be found to make it active. This will be useful in Lemma 8.

Tree patches supplement the information in a tree repository. We store both together:

Definition 21. *A* tree patch repository *is a pair* (R, S)*, where* R *is a tree repository,* S *is a set of tree patches, and* $\text{signature}(R) = \text{signature}(S)$.

Note that since a tree patch only has names, and not the content of patches, repositories cannot exchange information through tree patches alone: access to the underlying tree repository is still needed. However, the invariant that $\text{signature}(R) = \text{signature}(S)$ hints that the tree patches are encoding meaningful information about the repository, and give us hope:

- That a meaningful "partial pull" operation could be implemented: allow the user to pull a subset of the tree patches from another repository, and then update R so that its signature matches.
- Similarly, that a meaningful "obliterate" operation could be implemented.
- Finally, that it may be possible to design a tree patch format that includes the underlying patches, and can be used to exchange information between repositories.

The following lemmas and corollaries show that every non-destructive operation can be recorded with a corresponding tree patch, and that in particular, when merging two tree patch repositories, we can take the union of the tree patch sets (possibly adding a conflict resolution patch).

Lemma 8. *If* S_1 *is a set of tree patches and* (P_2, A_2) *is a complete (Definition 15) signature such that* $\text{signature}(S_1) \preceq (P_2, A_2)$*, then there exists a tree patch* t *such that* $\text{signature}(S_1 \cup \{t\}) = (P_2, A_2)$.

Proof. Let $(P_1, A_1) = \text{signature}(S_1)$.

Set $t = (P_t, D_t)$ where

$$D_t = P_2 \cap \text{reverse\_dependencies}(\text{dependencies}(P_2 \setminus P_1) \setminus A_2)$$

and $P_t = D_t \cup (P_2 \setminus P_1)$. In other words, we for every inactive dependency of a new name (dependencies$(P_2 \setminus P_1) \setminus A_2$), we explicitly disable it and all of its reverse dependencies.

Let $(P_2', A_2') = \text{signature}(S_1 \cup \{t\})$. We must show that $P_2 = P_2'$ and $A_2 = A_2'$.

Indeed,

$$\begin{aligned} P_2' = &\left(\bigcup_{(P',D') \in S_1} P' \right) \cup P_t \\ =&P_1 \cup D_t \cup (P_2 \setminus P_1) \\ =&P_2 \end{aligned}$$

where the last step uses the facts that $D_t \subseteq P_2$ and $P_1 \subseteq P_2$ since signature$(R_1) \preceq$ signature(R_2).

Now, let's compare A_2 and A_2'.

Suppose $n \in A_2$; we'll show $n \in A_2'$.

First, we'll show that $n \notin D_t$. Indeed, let n' be any dependency of n. Then since (P_2, A_2) is complete, $n' \in A_2$, so $n' \notin$ dependencies$(P_2 \setminus P_1) \setminus A_2$. Since none of n's dependencies are in that set, n is not in D_t.

Now, From the definition of $\preceq$ (Definition 16) we know $n \in A_1$ or $n \in$ dependencies$(P_2 \setminus P_1)$.

- **Case 1:** $n \in A_1$**.** In this case, since $(P_1, A_1) = \text{signature}(S_1)$, there is no $(P', D') \in S_1$ such that $n \in D' \cap R(P_1, P')$, taking $R(P_1, P')$ from Definition 14. Adding t to the set cannot add any new names to $R(P_2, P')$, so $n \notin D' \cap R(P_2, P')$. Finally, the new element $D_t \cap R(P_2, P_t)$ also does not contain n since $n \notin D_t$. So n is in the set D from Definition 14, so $n \in A_2'$.

- **Case 2:** $n \in$ dependencies$(P_2 \setminus P_1)$**.** Then reverse_dependencies(n) contains a name than is in P_2 but not in any of the sets P' from the signature S_1, so none of the sets $R(P_2, P')$ contain n for $(P', D') \in S_1$. Since $n \notin D_t$ we also have $n \notin D_t \cap R(P_2, P_t)$, and so n is in the set D from Definition 14, so $n \in A_2'$.

We have shown that $A_2 \subseteq A_2'$. Conversely, suppose $n \in P_2 \setminus A_2$. Then by the construction of $D_t \subseteq P_t$, $R(P_2, P_t)$ and D_t both contain n, so $n \notin A_2'$.

We have $A_2 \subseteq A_2'$ and $(P_2 \setminus A_2) \cap A_2' = \emptyset$; since $A_2' \subseteq P_2$ we conclude that $A_2' = A_2$ as required. □

Corollary 1. *If (R_1, S_1) is a tree patch repository and R_2 is a tree repository such that* signature$(R_1) \preceq$ signature(R_2)*, then there exists a tree patch t such that $(R_2, S_1 \cup \{t\})$ is a tree patch repository.*

Proof. R_2's signature is complete (Lemma 6), so we can apply Lemma 8 to find the patch t. □

The proof of Lemma 8 is constructive, so it could be used to construct the patch required for Corollary 1. However this could be computationally expensive. Probably it would be better to create specialized algorithms to create tree patches for specific operations.

For example, adding a new (primitive) patch (Section 4.2.5) p simply corresponds to the tree patch $(\{\text{name}(p)\}, \emptyset)$. Reactivating a patch (Section 4.2.7) is similarly easy in the case where no patches needed to be deactivated. Deactivating patches (Section 4.2.6) may be more difficult.

Now, we show that when merging tree patch repositories, we can keep the tree patches from both.

Lemma 9. *For any sets of tree patches S_1 and S_2,* signature$(S_1 \cup S_2)$ *is the smallest signature that is greater than both* signature(S_1) *and* signature(S_2) *(in order theory, it is the* join *of the two signatures).*

Proof. Let (P_1, A_1) and (P_2, A_2) by the signatures of S_1 and S_2. Let $(P_{1\cup2}, A_{1\cup2}) =$ signature$(S_1 \cup S_2)$.

Let (P_3, A_3) be any signature such that $(P_1, A_1) \preceq (P_3, A_3)$ and $(P_2, A_2) \preceq (P_3, A_3)$. We must show that $(P_{1\cup2}, A_{1\cup2}) \preceq (P_3, A_3)$.

It is simple to show that $P_{1\cup2} = P_1 \cup P_2$ and $P_1 \cup P_2 \subseteq P_3$, so $P_{1\cup2} \subseteq P_3$.

It remains to show that $A_3 \subseteq A_{1\cup2} \cup \text{dependencies}(P_3 \setminus P_{1\cup2})$.

Let n be any name in A_3. If $n \in \text{dependencies}(P_3 \setminus P_{1\cup2})$ we are done. Henceforth assume $n \notin \text{dependencies}(P_3 \setminus P_{1\cup2})$; our goal is to show that $n \in A_{1\cup2}$.

Since $(P_1, A_1) \preceq (P_3, A_3)$, we know $n \in A_1 \cup \text{dependencies}(P_3 \setminus P_1)$.

For the sake of contradiction, suppose (P', D') is an element of S_1 such that $n \in D' \cap R(P_{1\cup2}, P')$, taking R from Definition 20. Then reverse_dependencies$(n) \cap P_{1\cup2} \subseteq$ dependencies(P'). In particular, reverse_dependencies$(n) \cap P_1 \subseteq$ dependencies(P'), so $n \in D' \cap R(P_1, P')$, so $n \notin A_1$. Therefore we must have $n \in \text{dependencies}(P_3 \setminus P_1)$. Since by assumption $n \notin \text{dependencies}(P_3 \setminus P_{1\cup2})$, we have $n \in \text{dependencies}(P_2 \setminus P_1)$: there is some $n' \in P_2 \setminus P_1$ such that $n \in \text{dependencies}(n')$. Since P' came from S_1, we know $n \notin \text{dependencies}(P')$, and so $n' \in$ reverse_dependencies$(n) \cap P_{1\cup2} \setminus \text{dependencies}(P')$. So $n \notin R(P_{1\cup2}, P')$: a contradiction.

This argument by contradiction shows $n \notin D' \cap R(P_{1\cup2}, P')$ for every $(D', P') \in S_1$. A similar argument shows it for every $(D', P') \in S_2$, and so n is not in the set D, so $n \in A_{1\cup2}$. □

Corollary 2. *Let (R_1, S_1) and (R_2, S_2) be tree patch repositories. Suppose R_3 is a merge of R_1 and R_2 (Definition 17).*

Then there exists a tree patch t such that $(R_3, S_1 \cup S_2 \cup \{t\})$ is a tree patch repository. If R_3 is minimal *in the sense that* signature(R_3) *is the smallest signature greater than both* signature(S_1) *and* signature(S_2)*, then $(R_3, S_1 \cup S_2)$ is a tree patch repository.*

Proof. By Lemma 9, signature$(S_1 \cup S_2) \preceq$ signature(R_3), so Lemma 8 gives us the required tree patch.

If R_3 is minimal, then Lemma 9 also tells us that signature$(S_1 \cup S_2) =$ signature(R_3). □

Caveats and missing pieces.

- The dependency structure of tree patches is not as simple as for primitive patches. Primitive patches are straightforward: a patch can be added to a repository if the repository has all of its dependencies and no conflicting patches. For tree patches, we can say a set of tree patches S is *consistent* if there exists a tree repository R such that signature(R) = signature(S). It's possible, for example, to have a situation where tree patch A can be added to the set $\{B, C\}$ or the set $\{B, D\}$ but not to the set $\{B\}$: that is, in the presence of patch B, A depends on C *or* D. (Say A and B add conflicting primitive patches p_1 and p_2 and C and D both deactivate p_2.)

- The complexity of some of the operations hasn't been worked out. For example, computing the tree patch corresponding to deactivating a primitive patch, or computing the new tree repository after a "partial merge" where some but not all tree patches are pulled, or similarly, computing the effect of obliterating a tree patch.

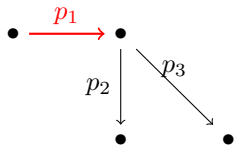
- Although Corollary 2 says that merging repositories doesn't require adding a new tree patch as long as the merge is "minimal", we haven't shown that merges ever are minimal. (We expect a merge with no conflicts is minimal, but this needs to be proved.)

- Tree patches cannot be used exclusively to exchange information between repositories, since they don't contain any actual (primitive) patches. Is there some way to add the necessary information?

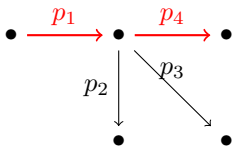
4.3 User interface considerations

What should the state of the repository be during a merge, before conflicts are fully resolved? Section 4.2.9 describes an interactive algorithm for merging which leads to the following question about the user interface.

Suppose Alice pulls from Bob's repository and resolves a conflict by recording a replacement patch, as in Example 4.1. During this process, the repository will reach the following intermediate state, where the conflicting patches have been deactivated but no new patch has been recorded:



If we are storing the repository as a tree patch repository as in Definition 21, then storing this repository will require adding a tree patch that disables p_2 and p_3. This is unfortunate, since this is only a stepping stone to what Alice really wants:



So, naïvely, the merge will result in two new tree patches being recorded instead of one. To avoid this, we suggest storing a special *pending tree patch* with the repository which records changes in progress, such as the deactivation of p_2 and p_3. The pending tree patch can be converted to a recorded patch once the user is finished.

This is similar to Darcs's pending operations like file renames, which behave as primitive patches that aren't yet recorded. One difference is that while in Darcs, the user is always free to discard all pending changes, it's not always possible here: Alice cannot simply discard the pending deactivation of p_2 and p_3, since that would leave the repository in an impossible state, with conflicting patches p_2 and p_3 active. This is a disadvantage of using tree patch repositories. It could be mitigated by saving a snapshot of the repository allowing Alice to revert to the state before the merge.

References

[1] Judah Jacobson. A formalization of darcs patch theory using inverse semigroups. Retrieved from ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf on 2020-06-21, orginally 2009.

[2] Ian Lynagh. Camp patch theory. Retrieved from https://archives.haskell.org/projects.haskell.org/camp/files/theory.pdf (linked from https://archives.haskell.org/projects.haskell.org/camp/index.shtml) 2020-06-21.