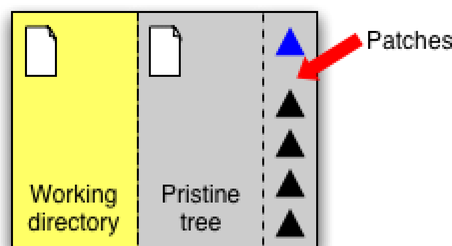
Understanding Darcs/Print Version

Getting Started

Anatomy of a darcs repository

Understanding how to use the darcs commands can be a lot easier if you have a rough idea how things work. We're not asking you to learn about patch commutation algebra (yet), but one thing you should at least be comfortable with is the anatomy of a darcs repository.

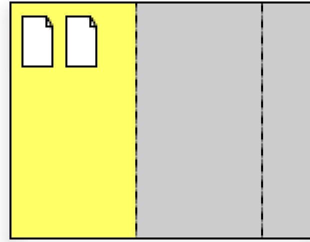


The idea of the diagram is as follows: the stuff on the left in yellow is you. That's what darcs calls the **working directory**. The stuff in grey, on the other hand, is part of that mysterious `_darcs` directory if you've played around with darcs before. This can be broken down some more. The **pristine tree** (middle) is exactly like your working directory, except only representing the last saved state. It's not essential to darcs's operations, but it makes things run more efficiently, and is perhaps useful for understanding how things work. Finally, the right-most portion is the set of **patches**. Patches are what makes darcs... well... darcs. Darcs thinks in patches. Almost every darcs operation somehow involves (darcs) juggling some patches around behind the scenes. Enthusiastic darcs users find that this makes life easier in a number of ways. For example, accepting patches to your code becomes an extremely natural thing to do - you just let darcs apply them and in they go. But before we get to that, let's tackle a small set of essential commands.

Essential commands

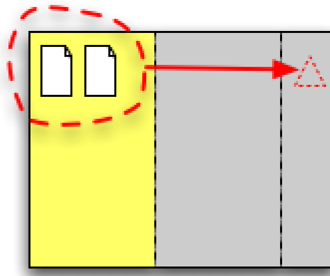
init

Say you have a directory with some files in it. When you run `darcs init` to initialise the repository, you get an empty new darcs repository. Your working directory might contain files, but darcs does not know about them yet.



add

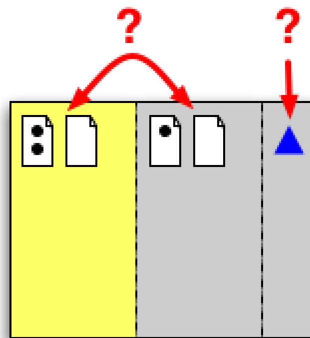
`darcs add` tells darcs to keep track of a file or directory that was only in your working directory. It creates or adds to a special temporary patch which we call the **pending patch** (will be represented in blue). Note that it does *not* affect your pristine tree whatsover! The idea is that we haven't saved your work (which is what the pristine tree is for). We've only told darcs that we might conceivably want to save it later on.



Note that the pending patch is different from all the other patches. It is really a representation of (some) things you have not yet converted into a real darcs patch.

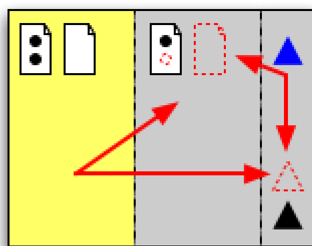
whatsnew

The `darcs whatsnew` compares the working directory against the pristine tree (theoretically, against the set of patches) and displays what has changed between them. If there is anything in the pending patch, it also displays the contents of that.



record

The `darcs record` command is how you save your work. It copies your chosen changes from the working directory to the pristine tree, and more importantly, creates a new patch representing those changes. Changes can also come from the pending patch, and these changes will also be propagated into the pristine tree.

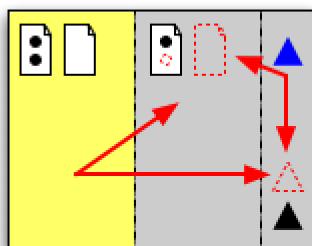


Making Changes

Editing files

`record` makes new stuff old

The **record** command takes the changes which only exist in your working directory (or the pending patch) and updates the pristine tree. The result of a record operation is a new patch.



`replace` for renaming tokens

The **replace** command is useful for explicitly telling darcs to replace one word with another (for example, a variable name, if you are a programmer).

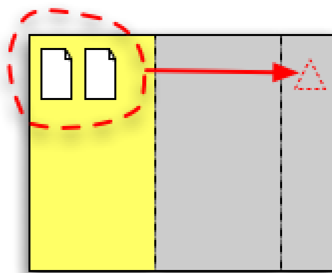
Note that because of the underlying patch theory, replace only works if the new word doesn't already exist in the file. Darcs will helpfully let you know if you try to replace something you cannot. Also, there is a switch for forcing the replacement, but the resulting patch is not a clean darcs-replace patch, but a combination of that, and what you would have gotten if you had edited the file in a text editor. In short, forcing darcs to replace when it really doesn't want to can lead to counter-intuitive results.

Playing with files

We saw **add** in the last chapter. What about the other file-related commands?

add tells darcs to pay attention

add, unsuprisingly, adds a file or directory to the list of files that darcs is paying attention to.



`mv` for moving or renaming

mv lets you rename a file or put it in a different directory. This command updates the pending patch with a move command.

you don't need `remove`

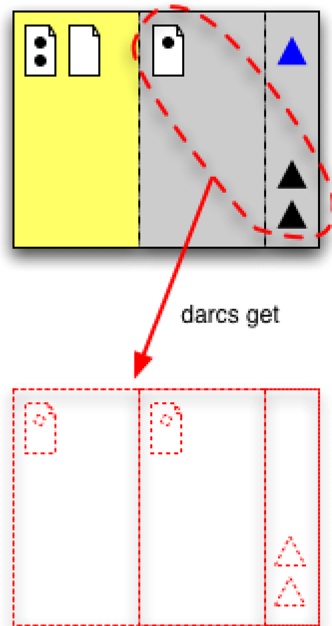
You might think that **remove** gets rid of a file, but actually all it does is to remove it from the list of files that darcs is paying attention to. You probably just want to delete the file instead (eg: with `rm`). Darcs will notice and record the change next time you `darcs record` that file. So what's the remove command good for? It might be handy if you want to just remove the file from darcs, without actually getting rid of your physical copy. This is most useful when you've accidentally used **darcs add** on a file you don't want darcs to pay attention to.

Working With Others

The commands

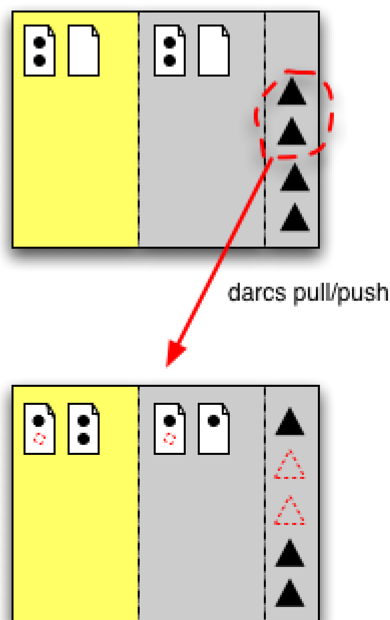
get

`darcs get` makes a copy of an entire darcs repository. Note that we only get the recorded patches (and pristine tree), not any of the pending stuff.



pull/push

`darcs pull` copies from some other repository patches that you do not have. The patches are applied to your pristine tree and working directory. This may cause changes to be merged. Note that `darcs push` does the same thing, but in the other direction.



send

Darcs `send` is sort of like `push`, only it doesn't actually apply the patches anywhere. Instead it generates a handy email to the person or people who own the repository. If it can't figure out who owns the repository, it will let you send to any email address you want. Note that you can also pass the `-o` command to "send" into a

file, rather than a mail.

apply

Apply is what you do to a patch that somebody darcs sends to you. You can also use it for the files you generated via send -o. Note: push is actually just apply in disguise, but with all the boring work of copying files over being done for you.

put

Put enables to copy a local repository to remote location (for instance, via ssh). Think of put as the opposite of get.

Dealing with conflicts

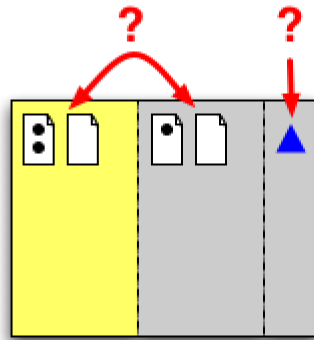
So you pulled a patch and you got a conflict. What do you do? See the chapter Dealing with conflicts

Reviewing Your Work

whatsnew

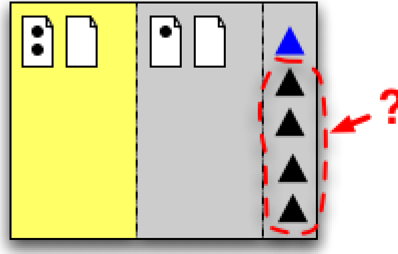
The whatsnew command allows you to get an overview of what unrecorded changes you have made on your working copy.

This way you can get an idea of what needs to be saved.



changes

Changes gives a changelog-style summary of the repo history



diff

Using this command you can get an output like that produced by the `diff` command. Allowing you to save changes in a plain old (yet very common) format.

Note that `darcs` does not depend on the `diff` binary.

Undoing Mistakes

There are many ways to get rid of things: remove, rollback, revert, obliterate, unpull, unrecord... One would almost think too many. Only three of these are very important. In order of gravity, they are `revert`, `unrecord`, `obliterate`. You can also see them as being part of this table of symmetries:

recency	task	anti-task
more recent	whatsnew (i.e: add, remove, mv, editing a file)	revert
recent	record	unrecord
less recent	pull	unpull (aka obliterate)

Don't worry too much about this table. Surely it will make sense later on.

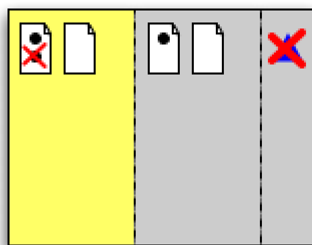
Anyway, here is a brief comparison of all the different ways you can get rid of stuff. Each one has its place, even the weird ones, like `rollback`.

The big three

`revert` only removes what's new

The simplest of these commands is **revert**. All reverting does is to get rid of stuff you have not recorded yet. You can think of revert as being the "opposite" of whatsnew. Revert gets rid of stuff that is new.

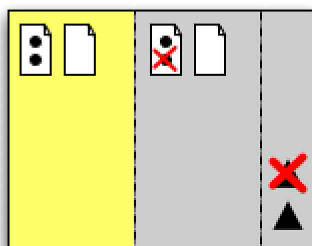
`reverted` something by mistake? Try `unrevert`



`unrecord` makes things new again

Unrecord does something quite different from revert. Whereas revert gets rid of stuff that is new, unrecord removes a patch, but here's the important part, *makes the stuff in it new again* so that you can choose to re-record or revert it at your leisure.

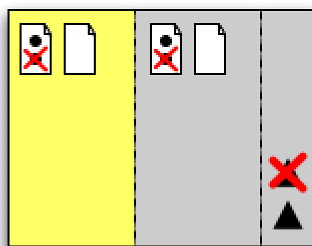
Unrecord should only be used if you recorded something, realised you made a mistake, and want to record it differently (note also amend-record). Note: only use unrecord if you are sure that your repository is the only one that has that patch in it!



Note that the picture above gives a somewhat more accurate depiction of what unrecord does - it removes a patch and the corresponding modifications from the pristine tree. The fact that something is new again is just a natural consequence of this fact.

`obliterate` is `unrecord` + `revert`

Obliterate was deliberately named to be scary. Obliterate can be seen as unrecording a patch (thus making its stuff new again) and then reverting it. In other words, obliterate totally wipes a patch out! Typically: you would use obliterate to go really far back in time. Say, "hmm, all that stuff I've been working on for the past three months was pretty stupid". Obliterate is the answer there.



Other ways to get rid of things

`unpull` is `obliterate`

Unpull and **obliterate** are exactly the same command and is only named this way to reflect its usefulness for undoing a pull.

`rollback`

Rollback is not a command that you can expect to use very often. The situation is this. You've got a patch you want to get rid of, but people have been telling you that you shouldn't obliterate or unrecord patches that are already in other people's repositories. So what do you do? One solution is to fire up your text editor and make exact opposite changes as the ones in the patch (and then record, etc). Another solution is to generate a rollback patch, which does the same thing. It creates a patch that does exactly the opposite of another patch.

Some users just find it easier to go the text-editor route.

`remove` doesn't belong here

Despite its getting-rid-of-things style name, `remove` is not really an undo kind of command. Its job is the opposite of `add`'s: it tells darcs not to pay attention to a file any longer. But as we mentioned in the previous chapter, most of the time you don't even need `darcs remove`. Simply telling your computer to get rid of the file in your working directory is good enough for darcs to notice it is gone.

Questions and objections

- But... but... I just want to go back to the state of my repository from two weeks ago!
  - `obliterate` is probably what you want. See above.

Dealing With Conflicts

Handle your conflicts at home

Resolving conflicts

For the moment, the best place to go for dealing with conflicts is the conflicts FAQ on darcs wiki (http://darcs.net/FAQ/Conflicts).

Introduction to Patch Theory

Math and computer science nerds only

(The occasional physicist will be tolerated)

Casual users be warned, the stuff you're about to read is not for the faint of heart! If you're a day-to-day darcs user, you probably do not need to read anything from this page on. However, if you are interested in learning how darcs *really* works, we invite you to roll up your sleeves, and follow us in this guided tour of the growing Theory of Patches.

What is the theory of patches?

The darcs patch formalism is the underlying "math" which helps us understand how darcs should behave when exchanging patches between repositories. It is implemented in the darcs engine as data structures for representing sequences of patches and Haskell functions equivalent to the operations in the formalism. This section is addressed at two audiences: curious onlookers and people wanting to participate in the development of the darcs core. My aim is to help you understand the intuitions behind all this math, so that you can get up to speed with current conflictors research as fast as possible and start making contributions. You should note that I myself am only starting to learn about patch theory and conflictors, so there may be mistakes ahead.

Why all this math?

One difference between centralized and distributed version control systems is that "merging" is something that we do practically all the time, so it is doubly important that we *get merging right*. Turning the problem of version control into a math problem has two effects: it lets us abstract all of the irrelevant implementation details away, and it forces us to make sure that whatever techniques we come up with are fundamentally sound, that they do not fall apart when things get increasingly complicated. Unfortunately, math can be difficult for people who do not make use of it on a regular basis, so what we attempt to do in this manual is to ease you into the math through the use of concrete, illustrated examples.

A word of caution though, "getting merging right" does not necessarily consist of having clever behaviour with respect to conflicts. We will begin by focusing on successful, non-conflicting merges and move on to the darcs approach to handling conflicts.

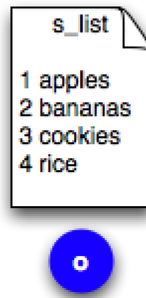
Context, patches and changes

Let us begin with a little shopping. Arjan is working to build a shopping list for the upcoming darcs hackathon. As we speak, his repository contains a single file *s_list* with the contents

```
1 apples
2 bananas
3 cookies
4 rice
```

Note:the numbers you see are just line numbers; they are not part of the file contents

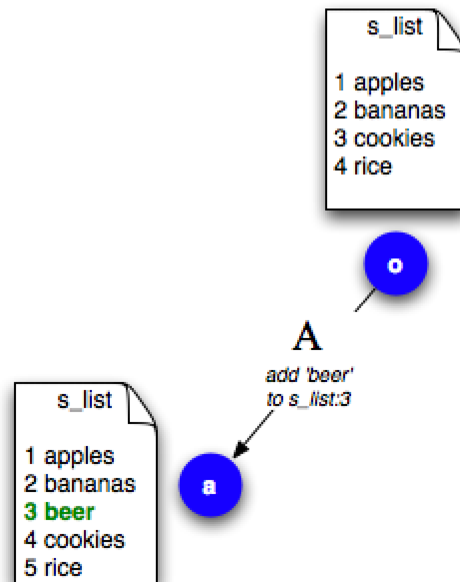
As we will see in this and other examples in this book, we will often need to assign a name to the state of the repository. We call this name a **context**. For example, we can say that Arjan's repository is a context ***o***, defined by there being a file s_list with the contents mentioned above.



Arjan makes a modification which consists of adding a line in *s_list*. His new file looks like this:

```
1 apples
2 bananas
3 beer
4 cookies
5 rice
```

When Arjan records this **change** (adding beer), we produce a patch which tells us not only what contents Arjan added ("beer") but where he added them, namely to line 3 of *s_list*. We can say that in his repository, we have moved from context o to context a via a **patch** A. We can write this using a compact notation like ${}^{o}A^{a}$ or using the graphical representation below:



Sequential patches

Starting from this context, Arjan might decide to make further changes. His new changes would be patches that apply to the context of the previous patches. So if Arjan makes a new patch B on top of this, it would take us from context a to some new context b. The next patch would take us from this context to yet another new context c, and so on and so forth. Patches which apply on top of each other like this are called **sequential patches**. We write them in *left to right* order as in the table below, either representing the contexts explicitly or leaving them out for brevity:

with context	sans context (shorthand)
$^{o}A^{a}$	A
$^{o}A^{a}B^{b}$	AB
$^{o}A^{a}B^{b}C^{c}$	ABC

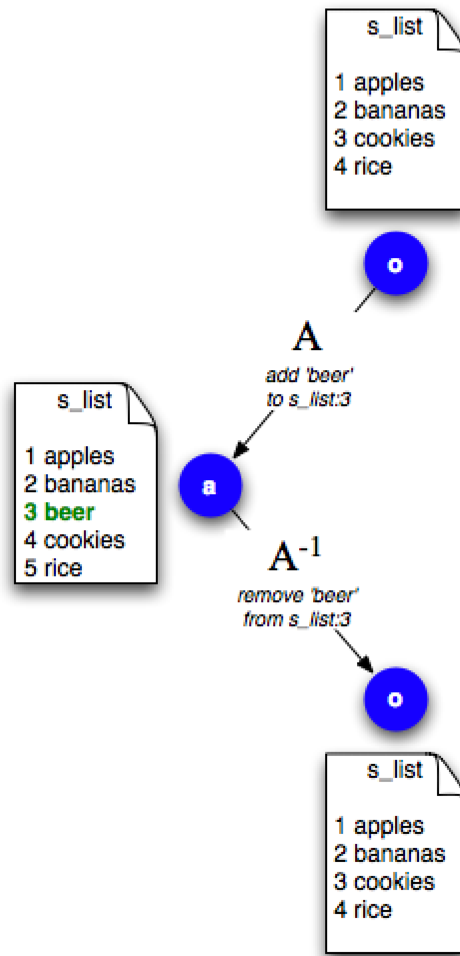
All darcs repositories are simply sequences of patches as above; however, when performing a complex operation such as an undo or exchanging patches with another user, it becomes absolutely essential that we have some mechanism for rearranging patches and putting them in different orders. Darcs patch theory is essentially about giving a precise definition to the ways in which patches and patch-trees can be manipulated and transformed while maintaining the coherence of the repository.

Inverses

Let's return to the example from the beginning of this module. Arjan has just added beer to our hackathon shopping list, but in a sudden fit of indecisiveness, he reconsiders that thought and wants to undo his change. In our example, this might consist of firing up his text editor and remove the offending line from the shopping list. But what if his changes were complex and hard to keep track of? The better thing to do would be to let darcs figure it out by itself. Darcs does this by computing an **inverse** patch, that is, a patch which makes the exact opposite change of some other patch:

> **Definition (Inverse of a patch)**:
>
> The Inverse of patch P is P^{-1}, which is **the** patch for which the composition PP^{-1} makes no changes to the context and for which the inverse of the inverse is the original patch.

So above, we said that Arjan has created a patch A which adds beer to the shopping list, passing from context o to a, or more compactly, $^{o}A^{a}$. Now we are going to create the inverse patch A^{-1}, which *removes* beer from the shopping list and brings us back to context o. In the compact context-patch notation, we would write this as $^{o}A^{a}A^{-1\,o}$. Graphically, we would represent the situation like this:

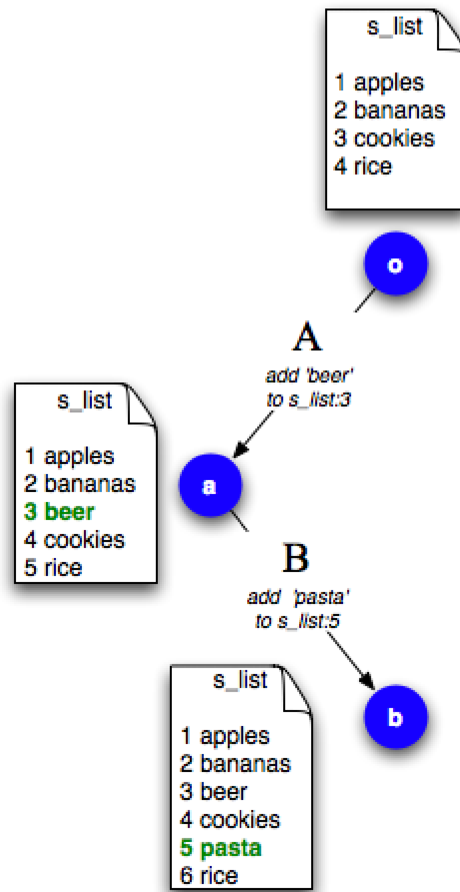


Patch inverses may seem trivial, but as we will see later on in this module, they are a fundamental operation and absolutely crucial to make some of the fancier stuff -- like merging -- work correctly. One of the rules we impose in darcs is that every patch must have an inverse. These rules are what we call **patch properties**. A patch property tells us things which must be true about a patch in order for darcs to work. People often like to dream up new kinds of patches to extend darcs's functionality, and defining these patch properties is how we know that their new patch types will behave properly under darcs. The first of these properties is dead simple:

Commutation

Arjan was lucky to realise that he wanted to undo his change as quickly as he did. But what happens if he was a little slower to realise his mistake? What if he makes some other changes before realising that he wants to undo the first change? Is it possible to undo his first change without undoing all the subsequent changes? It sometimes is, but to do so, we need to define an operation called **commutation**.

Consider a variant of the example above. As usual, Arjan adds beer to the shopping list. Next, he decides to add some pasta on line 5 of the file:



The question is how darcs should behave if Arjan now decides that he does not want beer on the shopping list after all. Arjan simply wants to remove the patch that adds the beer, without touching the one which adds pasta. The problem is that darcs repositories are simple, stupid sequences of patches. We can't just remove the beer patch, because then there would no longer be a context for the pasta patch! Arjan's first patch A takes us to context a like so: ${}^{o}A^{a}$, and his second patch takes us to context b, notably starting from the initial context a: ${}^{a}B^{b}$. Removing patch A would be pulling the rug out from under patch B. The trick behind this is to somehow *change the order* of patches A and B. This is precisely what commutation is for:

Why not keep our old patches?

To understand commutation, you should understand why we cannot keep our original patches, but are forced to rely on evil step sisters instead. It helps to work with a concrete example such as the beer and pasta one above. While we could write the sequence AB to represent adding beer and then pasta, simply writing BA for pasta and then beer would be a very foolish thing to do.

Put it this way: what would happen if we applied B before A? We add pasta to line 5 of the file:

```
1 apples
2 bananas
3 cookies
4 rice
5 pasta
```

Does something seem amiss to you? We continue by adding beer to line 3. If you pay attention to the contents of the end result, you might notice that the order of our list is subtly wrong. Compare the two lists to see why:

BA (wrong!)	AB (right)
1 apples 2 bananas 3 beer 4 cookies 5 rice 6 pasta	1 apples 2 bananas 3 beer 4 cookies 5 pasta 6 rice

It might not matter here because it is only a shopping list, but imagine that it was your PhD thesis, or your computer program to end world hunger. The error is all the more alarming because it is subtle and hard to pick out with the human eye.

The problem is one of context, specifically speaking, the context between A and B. In order for instructions like "add pasta to line 5 of s_list" to make any sense, they have to be in the correct context. Fortunately, commutation is easy to do, it produces two new patches B_1 and A_1 which perform the same change as A and B but with a different context in between.

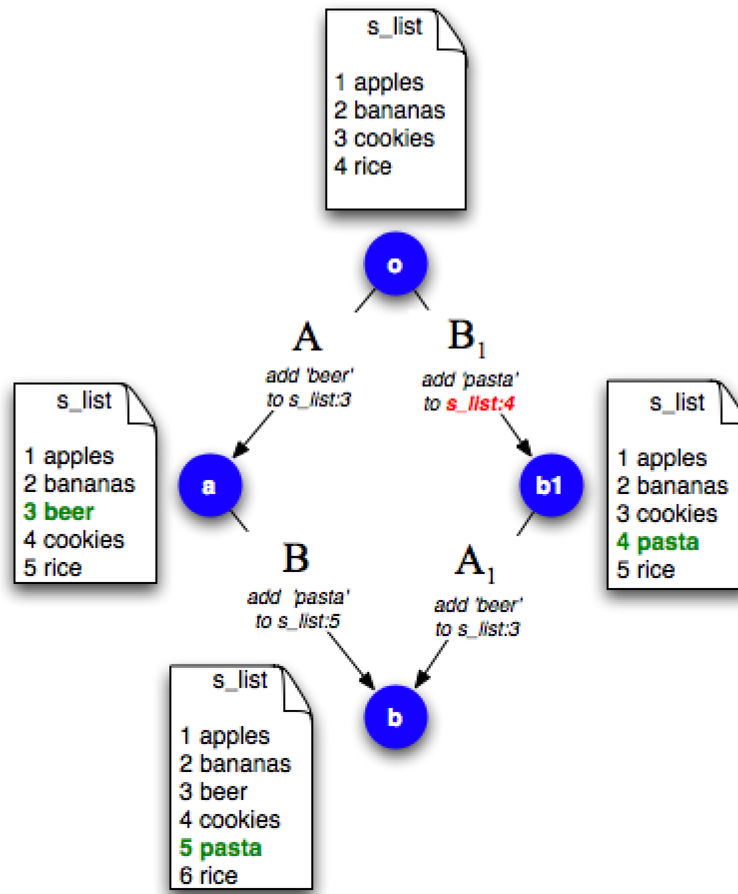
Exercises

Patch A_1 is identical to A. It adds "beer" to line 3 of the shopping list. But what should patch B_1 do?

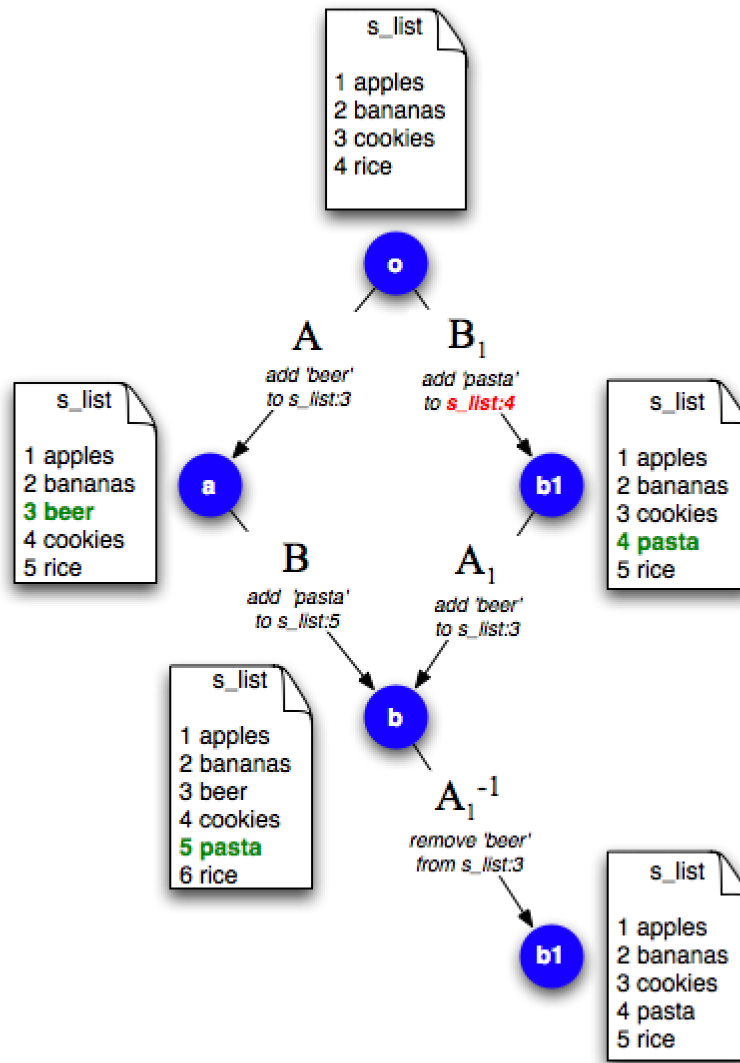
One more important detail to note though! We said earlier that getting the context right is the motivation behind commutation -- we can't simply apply patches AB in a different order, BA because that would get the context all wrong. But context does not have any effect on whether A and B can commute (or how they should commute). This is strictly a local affair. Conversely, the commutation of A and B does not have any effect either on the global context: the sequences AB and B_1A_1 (where the latter is the commutation of the former) start from the same context and end in the same context.

The complex undo revisited

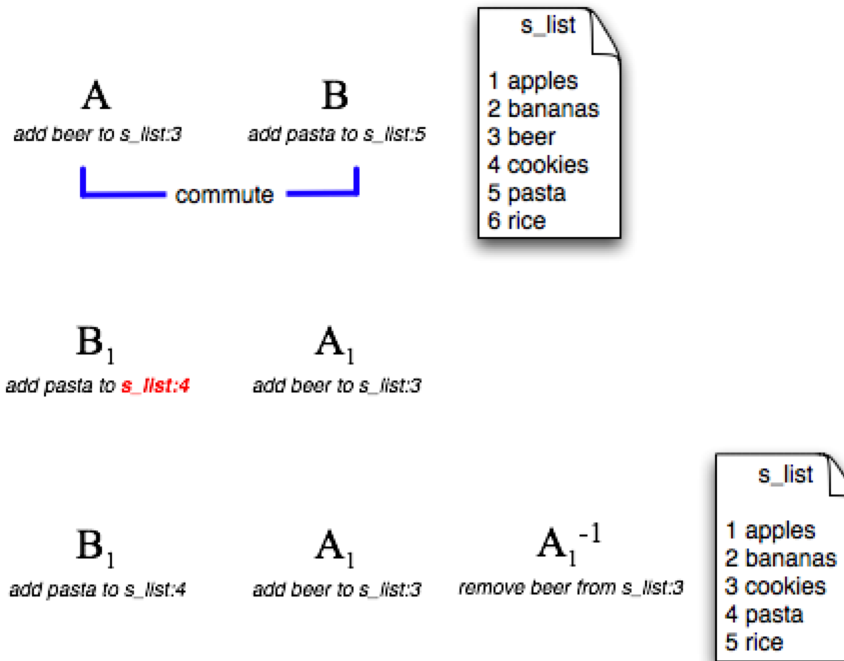
Now that we know what the commutation operation does, let's see how we can use it to undo a patch that is buried under some other patch. The first thing we do is commute Arjan's beer and pasta patches. This gives us an alternate route to the same context. But notice the small difference between B and B_1!



The purpose of commuting the patches is essentially to push patch A on to end of the list, so that we could simply apply its inverse. Only here, it is not the inverse of A that we want, but the inverse of its evil step sister A_1. This is what applying that inverse does: it walks us back to the context $b1$, as if we had only applied the pasta patch, but not the beer one.



And now the undo is complete. To sum up, when the patch we want to undo is buried under some other patch, we use commutation to squeeze it to the end of the patch sequence, and then compute the inverse of the commuted patch. For the more sequentially minded, this is what the general scheme looks like:



Exercises

Imagine the opposite scenario: Arjan had started by adding pasta to the list, and then followed up with the beer.

1. If there was no commutation, what concretely would happen if he tried to remove the pasta patch, and not the beer patch?
2. Work out how this undo would work using commutation. Pay attention to the line numbers.

Commutation and patches

Every time we define a type of patch, we have to define how it commutes with other patches. Most of time, it is very straightforward. When commuting two hunk patches, for instance, we simply adjust their line offset. For instance, we want to put something on line 3 of the file, but if we use patch Y to insert a single line before that, what used to be line 3 now becomes line 4! So patch X_1 inserts the line "x" into line 4, much like X inserts it into line 3.

Some patches cannot be commuted. For example, you can't commute the addition of a file with adding contents to it. But for now, we focus on patches which *can* commute.

Merging

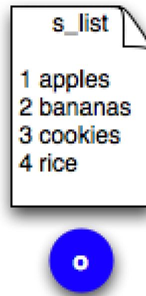
> *Note: this might be a good place to take a break. We are moving on to a new topic and new (but similar) examples*

We have presented two fundamental darcs operations: patch inverse and patch commutation. It turns out these two operations are almost all that we need to perform a darcs merge.

Arjan and Ganesh are working together to build a shopping list for the upcoming darcs hackathon. Arjan initialises the repository and adds a file *s_list* with the contents

```
1 apples
2 bananas
3 cookies
4 rice
```

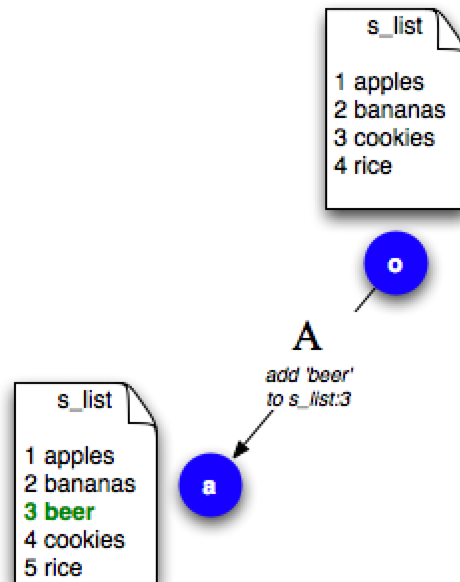
He then records his changes, and Ganesh performs a `darcs get` to obtain an identical copy of his repository. Notice that Arjan and Ganesh are starting from the same **context**



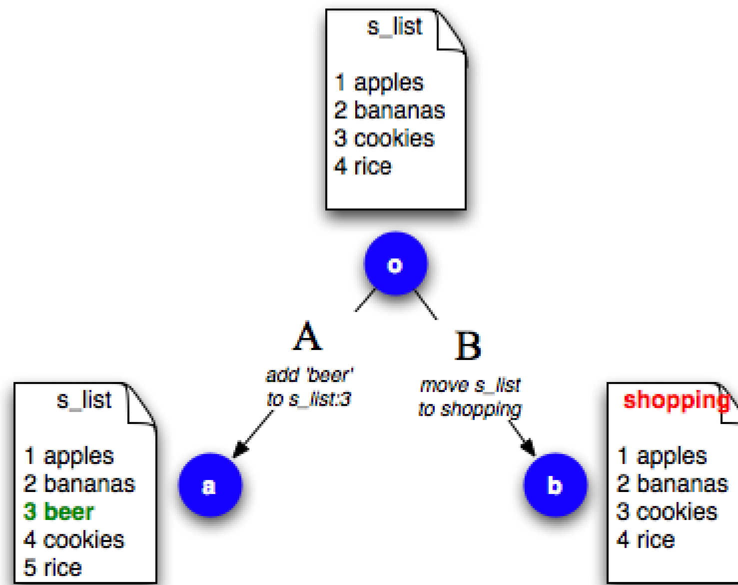
Arjan makes a modification which consists of adding a line in *s_list*. His new file looks like this:

```
1 apples
2 bananas
3 beer
4 cookies
5 rice
```

Arjan's patch brings him to a new context ***a***:

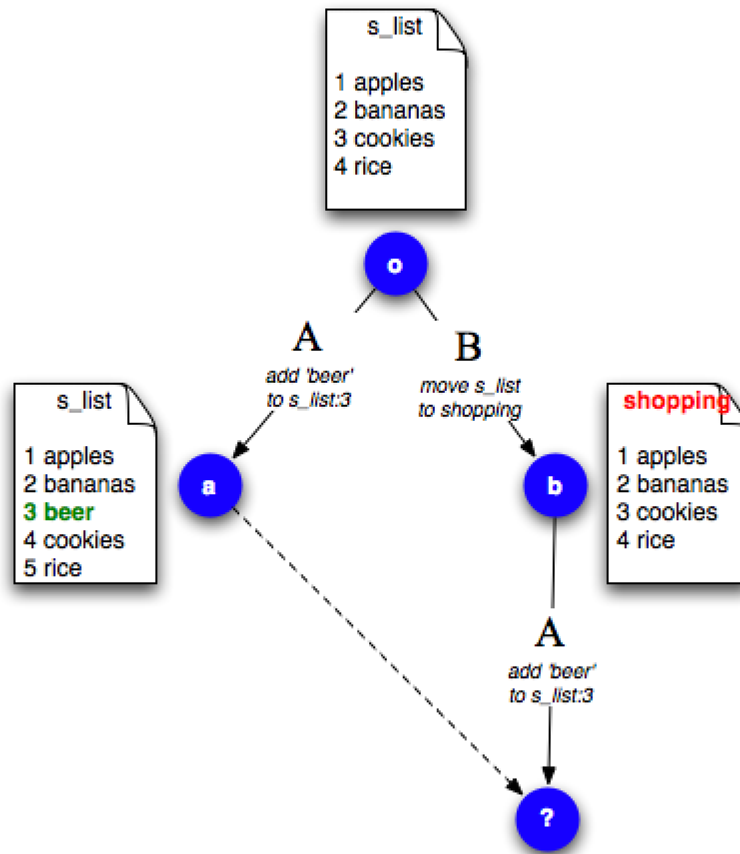


Now, in his repository, Ganesh also makes a modification; he decides that *s_list* is a little hard to decipher and renames the file to *shopping*. Remember, at this point, Ganesh has not seen Arjan's modifications. He's still starting from the original context ***o***, and has moved a new context ***b***, via his patch ***B***:



Parallel patches

At this point in time, Ganesh decides that it would be useful if he got a copy of Arjan's changes. Roughly speaking we would like to pull Arjan's patch A into Ganesh's repository B. But, there is a major problem! Namely, Arjan's patch takes us from context ***o*** to context ***a***. Pulling it into Ganesh's repository would involve trying to apply it to context ***b***, which we simply do not know how to do. Put another way: Arjan's patch tells us to add a line to file *s_list*; however, in Ganesh's repository, *s_list* no longer exists, as it has been moved to *shopping*. How are we supposed to know that Arjan's change (adding the line "beer") is supposed to apply to the new file *shopping* instead?

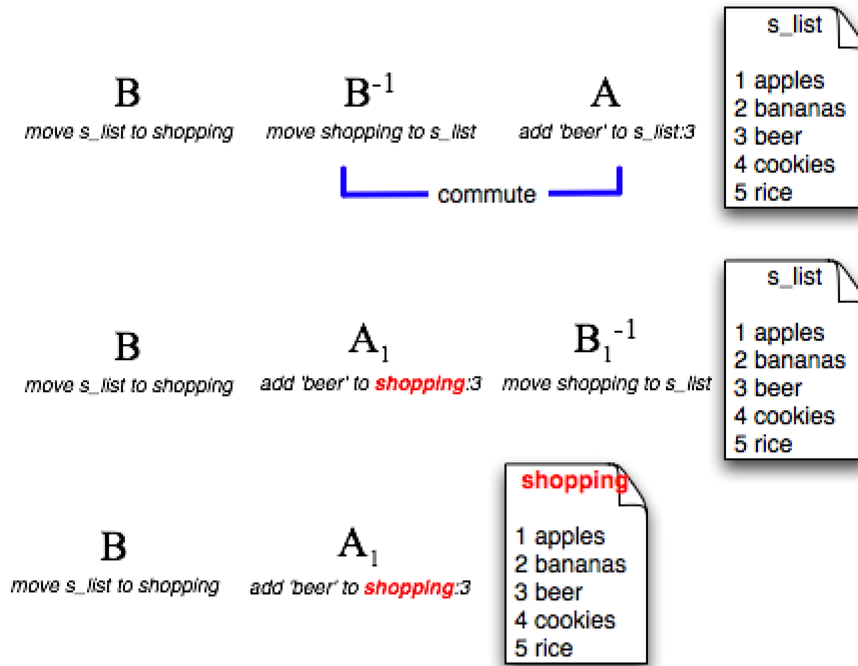


Arjan and Ganesh's patches start from the same context o and diverge to different contexts a and b. We say that their patches are **parallel** to each other, and write it as $A \vee B$. In trying to pull patches from Arjan's repository, we are trying to merge these two patches. The basic approach is to convert the parallel patches into the sequential patches BA_1, such that A_1 does essentially the same change as A does, but within the context of b. We want to produce the situation ${}^oB^bA_1^c$

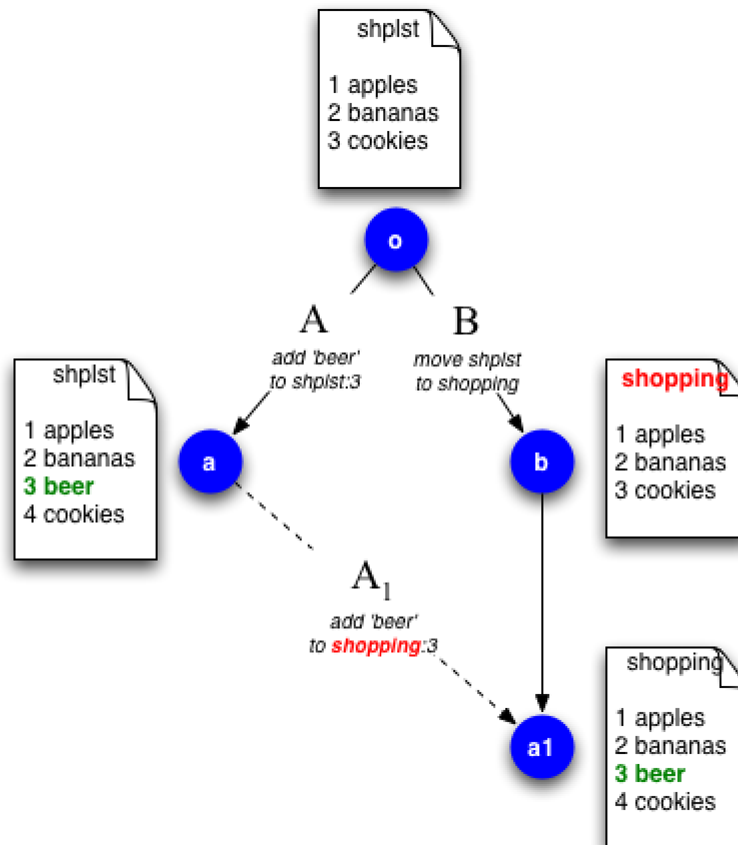
Performing the merge

Converting Arjan and Ganesh's parallel patches into sequential ones requires little more than the inverse and commutation operations that we described earlier in this module:

1. So we're starting out with just Ganesh's patch. In context notation, we are at ${}^oB^b$
2. We calculate the inverse patch B^{-1}. The sequence BB^{-1} consists of moving s_list to shopping and then back again. We've walked our way back to the original context: ${}^oB^bB^{-1o}$
3. Now we can apply Arjan's patch without worries: ${}^oB^bB^{-1o}A^a$, but the result does not look very interesting, because we've basically got the same thing Arjan has now, not a merge.
4. All we need to do is commute the last two patches, $B^{-1}A$, to get a new pair of patches $A_1{B_1}^{-1}$. Still, the end result doesn't seem to look very interesting since it results in exactly the same state as the last step: ${}^oB^bA_1{}^c{B_1}^{-1a}$
5. However, one crucial difference is that the second to last patch produces just the state we're looking for! All we now have to do to get at it is to ditch the ${B_1}^{-1}$ patch, which is only serving to undo Ganesh's precious work anyway. That is to say, by simply determining how to produce an A_1 which will commute with B, we have determined the version of A which will update Ganesh's repository.



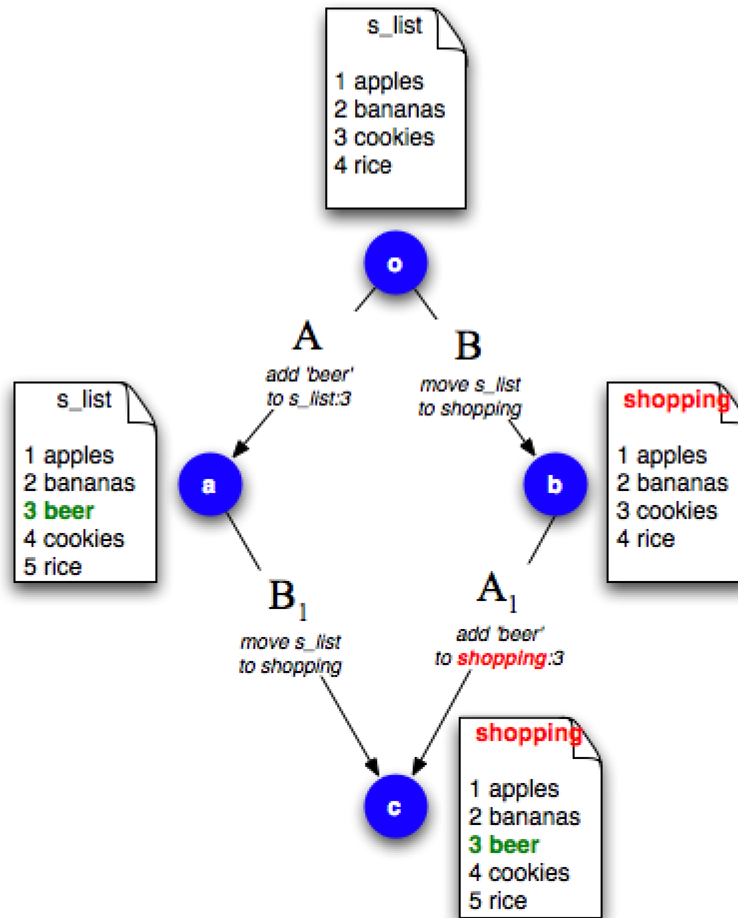
The end result of all this is that we have the patch we're looking for, A_1 and a successful merge.



Merging is symmetric

Concretely, we've talked about Ganesh pulling Arjan's patch into his repository, so what about the other way around? Arjan pulling Ganesh's patch into his repository would work the same exact way, only that he is looking for a commuted version of Ganesh's patch B_1 that would apply to his repository. If Ganesh can pull Arjan's patch in, then Arjan can pull Ganesh's one too, and the result would be exactly the same:

Merging is symmetric



Definition (Merge of two patches):

The result of a merge of two patches A and B is one of two patches A_1 and B_1, which satisfy the relationship $A \vee B \Longrightarrow BA_1 \leftrightarrow AB_1$

The merge definition describes what should happen when you combine two parallel patches into a patch sequence. The built-in symmetry is essential for darcs because a darcs repository is defined entirely by its patches. Put another way,

To be written

The commutation with inverse property

The definition of a merge tells us what we want merging to look like. How did we know how to actually perform that merge? The answer comes out of the following property of commutation and inverse: if you can commute the inverse of a patch A^{-1} with some other patch B, then you can also commute the patch itself against B_1.

Note how the left hand side of this property exactly matches the relationship demanded by the definition of a merge. To see why this all works,

To be written

Definitions and properties

definition of inverse	AA^{-1} has no effect
inverse of an inverse	$(A^{-1})^{-1} = A$
inverse composition property	$(AB)^{-1} = B^{-1}A^{-1}$
definition of commutation	$AB \leftrightarrow B_1A_1$
definition of a merge	$A \vee B \Longrightarrow BA_1 \leftrightarrow AB_1$
commutation with inverse property	$BA_1 \leftrightarrow AB_1$ if and only if $B_1{A_1}^{-1} \leftrightarrow A^{-1}B$

Intermediary Patch Theory

It's all patches

Before moving on, we would like to make one very minor point clear: in the day to day operation of darcs, we talk about pulling and pushing **patches**, and of recording and reverting **changes**. This is just a user interface convention. In patch theory terms, all of these are just patches. The patches which you pull and push are **named patches**, patches which contain a name and a list of unnamed patches. So in fact, when you pull a single named patch from somebody else's repository, you are pulling a sequence of potentially many primitive patches. What does this mean for merging?

*Prior to darcs 1.0.6, **changes** were also called **patches**, but we decided it was too confusing.*

Merging a sequence of patches

In the last chapter, we saw that dealing with simple, non-conflicting merges consists mainly of making an inverse patch and commuting that inverse with the other side's patches. Let us now explore a slightly more complicated scenario, where we have to merge against a non-conflicting sequence of patches. We do this with a variant of the darcs hackathon shopping list. As usual, Arjan and Ganesh are working together to write the shopping list. They both start from a common file *shplst* containing

```
apples
bananas
cookies
```

As before, Arjan inserts "beer" in line 3 of *shplst* and records the change. He then decides to add another item on the end of the list, this time, "pasta" and records his second change. In darcs notation, Arjan has brought us from an initial context o, to a new context a with beer in it, and then to yet another context c with pasta as well..

FIXME: will be fleshed out: i want to show what happens when Ganesh pulls two patches in

Sequences of patches

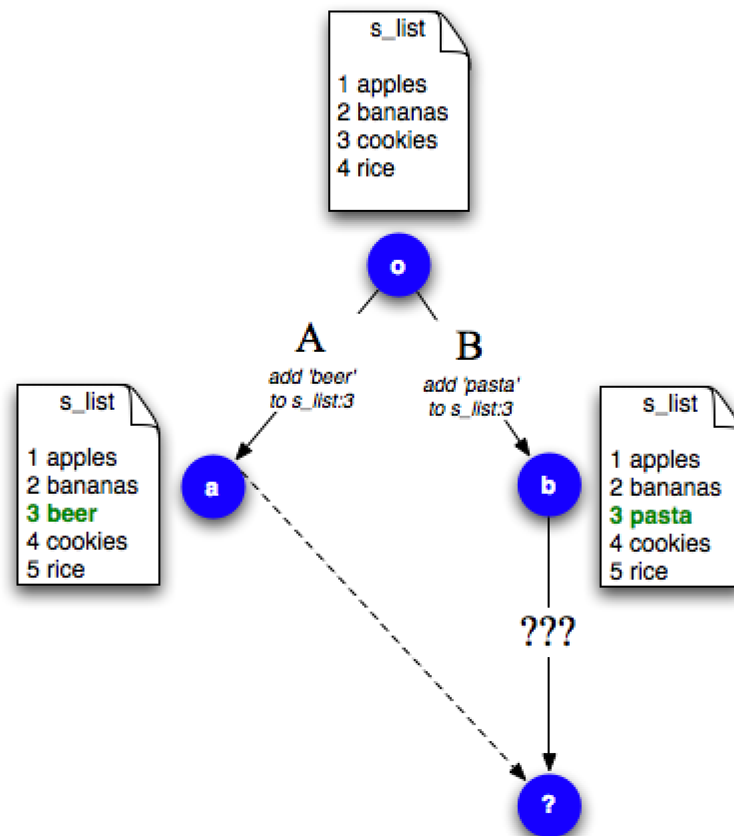
Permutivity

Patch Theory and Conflicts

Conflicts

Up to now, we have only dealt with merging patches that do not conflict with each other. The next question of interest is how darcs should behave when they do.

Consider the previous darcs hackathon example, where as usual, Arjan decides that the shopping list needs some beer. In this scenario, Ganesh decides that you can't live on apples and cookies alone and records a patch adding "pasta" to the *s_list* file. Now he wants to know what Arjan is up to, and so pulls the beer patch into his repository, but oh no! Arjan and Ganesh's patches conflict! How should darcs behave here?



The darcs answer is that both patches *cancel each other out* so that neither of them has any effect. The resulting shopping list has neither beer nor pasta. This might sound alarming, but it's not as bad as you might think. Darcs does not silently delete your code. After canceling the two patches, it adds a third patch into your working

directory which indicates both sides of the conflict so that you can select the one that you want. So any resolution you apply is a third patch which depends on the two conflicting ones. If you did `darcs whatsnew` on Ganesh's repository at this point, what you would get is something like this:

```
v v v v v v
beer
-----------
pasta
^ ^ ^ ^ ^ ^
```

How do we know we have a conflict?

It is intuitively obvious that Arjan's patch conflicts with Ganesh's, but intuition is useless if it does not translate into actual Haskell code. The first issue is thus that of knowing that we have a conflict in the first place.

All of this boils down to commutation. We have a conflict if commutation is not defined for the two patches. Let us briefly revisit the merge process described in the previous chapter. When Ganesh tries to pull Arjan's patch in, he tries to adapt the patch to his context by performing the following sequence: invert his own patch, apply Arjan's patch A, commute the inverted patch with Arjan's patch, and discard the evil step sister of his inverted patch. As we know, inverting patches is easy. Ganesh's patch is inverted into something which remove 'pasta' from line 3 of the s_list file. On the other hand, when we try to commute that against Arjan's patch, we have a failure.

Why? Simply because it is how we define commutation between the two types of patches. For instance, both Ganesh's and Arjan's patches are **hunk patches**. The commutation of two hunk patches of the same file is defined in darcs using Haskell code very similar to the following (simplified from PatchCommute.lhs):

```
commuteHunk :: FileName -> (FilePatchType, FilePatchType) -> Maybe (Patch, Patch)
commuteHunk f (p1@(Hunk line2 old2 new2), p2@(Hunk line1 old1 new1))
  | line1 + lengthnew1 < line2 = Just ...
  | line1 + lengthnew1 == line2 && nonZero = Just ...
  | line2 + lengthold2 < line1 = Just ...
  | line2 + lengthold2 == line1 && nonZero = Just ...
  | otherwise = Nothing
  where nonZero = lengthold2 /= 0 && lengthold1 /= 0 && lengthnew2 /= 0 && lengthnew1 /= 0
        lengthnew1 = length new1
        lengthnew2 = length new2
        lengthold1 = length old1
        lengthold2 = length old2
```

Only four cases are defined. The first two cases cover the situation where the `p1` occurs in an earlier part file than `p2` (even bumping up against it as in the second case). The latter two cases cover the reverse situation (`p2` is in earlier part of the file than `p1`). However, the case where `p1` and `p2` overlap simply does not fall into one of these possibilities. Thus we have a conflict on our hands.

Forced commutation

Now that we know we have a conflict, we now need to deal with this conflict in a sane manner. We not only want to deal with the conflict at hand, but deal with it in a way which allows the conflict to propagate cleanly across an entire sequence of patches. Well, darcs is based on commutation, so in order to keep things running smoothly, we need to make sure that things continue to commute. So, we're going to define a secondary **forced commutation** operation that we only use when there is a conflict.

Recall the definition of commutation from the previous chapter:

The forced commutation is going to do something similar, but with a very odd twist. Instead of patches Y_1 and X_1 performing the same change as their respective ancestors Y and X; forced commutation is going to give us patches, each of which makes the change that the *other* patch does. That is, normal commutation wants Y_1 to do roughly the same thing as Y, but forced commutation makes it do the same thing as X.

operation	effect of Y_1	effect of X_1
normal commutation	Y	X
forced commutation	X	Y

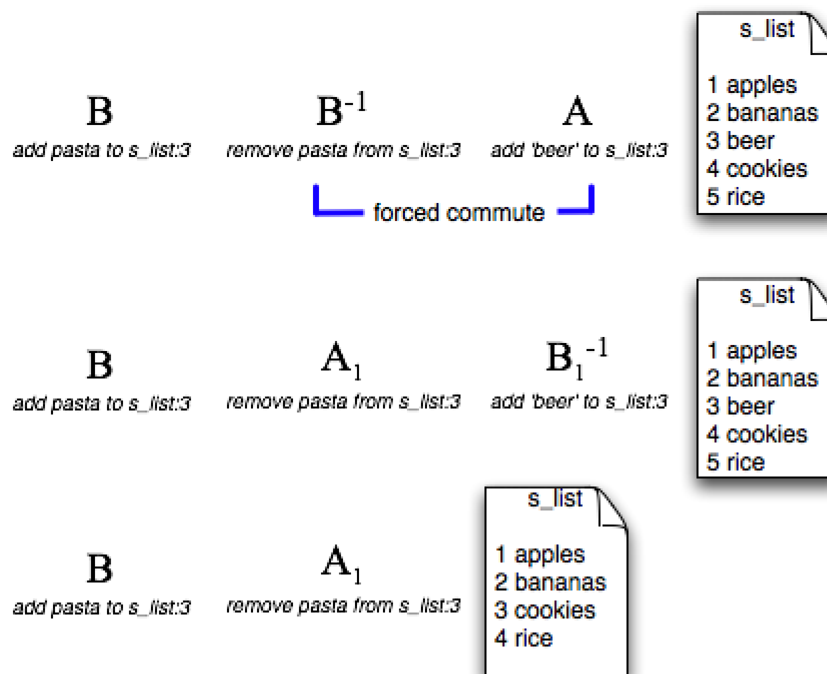
Effects

As a side note, we're going to need a little terminology to keep ourselves from tripping over our tongues. It's not very convenient to always talk about one patch making the same change as another patch, which is something we will be referring to a lot. So let us compress things a little bit. Instead of saying that patch Y_1 makes the same change as X, let us simply say that the **effect** of Y_1 is X. It is the same idea, but with slightly smoother terminology.

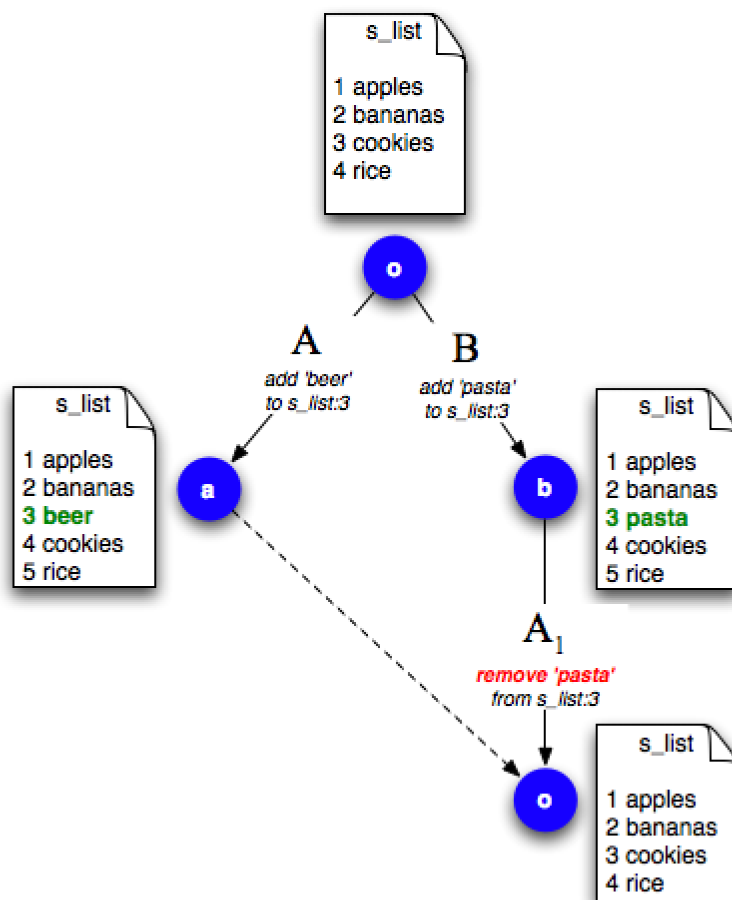
Forced commutation in merging

Let us see what the implications of this are for Ganesh and Arjan. We want to commute the inverse of Ganesh's patch (B^{-1}) against Arjan's patch. Since the two patches conflict, we have to resort to forced commutation, which produces two patches A_1 and ${B_1}^{-1}$ with the following bizarre properties:

- the effect of A_1 is B^{-1}; it removes Ganesh's "pasta" from the shopping list.
- likewise, the effect of ${B_1}^{-1}$ is A; it adds Arjan's "beer" to the shopping list.



This is all very convenient, because if I may remind you, what we're really after is cancelling out the patches. If we do the standard merging technique of simply removing ${B_1}^{-1}$ (so we don't add the beer after all), we will have successfully undone Ganesh's pasta patch. The merge is complete!



Marking the conflict

But wait! We can't just leave things undone. How is the poor developer supposed to know if there is a conflict, if darcs handles them by undoing things? The answer is that we're not going to stop here. Undoing the conflict is a very important first step, as we will see in further detail below. Look at it this way. We know there was a conflict, because of the way commutation was defined, and we know which patches were involved in the conflict. So whenever this happens, we *first* undo everything, and then inspect the contents of the conflicting patches, and use that to create a new conflict-marking patch.

FIXME:insert image here showing the conflict-marking patch

Darcs 2

:*TODO: introduce this section*

The exponential merge problem

Unfortunately, the darcs 1 merge algorithm has the property that certain merges -- merges that people have experienced in real life -- are exponential in time with respect to the size of conflict (in number of conflicting patches). This leads to the problem that some users have experienced where users would do a `darcs pull` and inexplicably, darcs would just sit there and hang...

So how does the new darcs 2 fix this problem? What's going on under the hood?

Conflictors

The notion of conflictors is essentially that we would special patches that contain a list of patches they conflict with

Use of Generalised Algebraic Datatypes to improve code safety

- See also Haskell/GADT and http://wiki.darcs.net/Ideas/GADTPlan

Current research

Retrieved from "https://en.wikibooks.org/w/index.php?title=Understanding_Darcs/Print_Version&oldid=3371490"

This page was last edited on 5 February 2018, at 12:30.

Text is available under the Creative Commons Attribution-ShareAlike License.; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.