# Camp Patch Theory

Ian Lynagh

August 10, 2019

## Contents

## 1 Overview

Camp is a version control system, similar in philosophy to darcs. In this document we present a description of camp's patch theory.

This document is very much a work in progress. Some sections are more readable than others. Some coq proofs are starting to appear, and where they haven't there may be a (somewhat handwavey) hand proof.

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

In Section 3 we say define a set of unnamed patches, and explain what behaviour we require that they satisfy. We also introduce a concept of commutation for unnamed patches.

In Section 4 we define sequences of unnamed, and some properties that they must satisfy.

In Section 5 we define a subset of all the possible unnamed patch sequences, which we call sensible.

In Section 6 we introduce names.

Everything up to this point is an input to the paper. In other words, we require that we are provided with unnamed patches, unnamed patch commute and the notion of sensible unnamed patch sequences, which satisfy the properties that we specify. Likewise, we require that we are given a suitable set of names.

While it is possible to start from a lower level, and prove the correctness of the operations on unnamed patches, that is not the goal of this document.

In Section 7 we build named patches out of names and unnamed patches.

In Section **??** we build sequences from these named patches.

In Section 8 we take a break from the formal definition, and give a description of how merges (of non-conflicting changes) work.

In Section 9 we introduce the concept of *patch universes*. This is an algebraic structure that many kinds of patches are examples of.

This actually give you all that you need in order to make a version control system, provided that you never make conflicting changes that you later want to merge.

In Section 11 we introduce a "contexted patch" datastructure.

In Section 13 we define catches and repos, and make conjectures that they satisfy properties that we want them to satisfy.

In Section 14 we prove that catches form a patch universe.

The result is a version control system in which it is always possible to merge repos — which is vital for a distributed version control system.

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ notation

## 2 Notation

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

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

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

| | |
|---|---|
| Unnamed Patches | $\underline{p}$, $\underline{q}$, $\underline{r}$, $\underline{s}$, $\underline{t}$, $\underline{u}$, $\underline{v}$ |
| (Named) Patches | $p$, $q$, $r$, $s$, $t$, $u$, $v$ |
| Contexted patches | $w$, $x$, $y$, $z$ |
| Catches | $c$, $d$, $e$, $f$, $g$ |

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

We use _ to mean "some value that we do not wish to name". Multiple _s do not necessarily refer to the same value.

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

The coq code is typeset thus:

```
(*  Here comes the science bit. Concentrate! *)
```

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ unnamed_patches

## 3 Unnamed Patches

```
Module Export unnamed_patches.
```

We start by introducing *unnamed patches*, and the concept of *patch commutation*. For now, we will refer to unnamed patches as simply *patches*.

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

```
Reserved Notation "p ^u" (at level 10).
Reserved Notation "« p , q » <˜>u « q' , p' »" (at level 60, no associativity).
Reserved Notation "p <˜?˜>u q" (at level 60, no associativity).

Record UnnamedPatch : Type := mkUnnamedPatch {
    up_type : Set;
```

The underlining in the $\underline{\mathbf{P}}$ and $\underline{p}$ syntax is a bit heavy, but we prefer to reserve the cleaner syntax for named patches.

**Axiom 3.1 (unnamed-patch-inverse)**
$\forall \underline{p} \in \underline{\mathbf{P}} \cdot \underline{p}^{-1} \in \underline{\mathbf{P}}$.

```
(*
XXX inverse is an axiom in the latex. Maybe it should
be a definition?
*)
unnamedPatchInverse : up_type → up_type
    where "p ^u" := (unnamedPatchInverse p);
```

**Explanation**
*All patches have an inverse.*

```
unnamedPatchInvertInverse : ∀ (p : up_type), (p^u)^u = p;
```

**Definition 3.2 (unnamed-patch-commute)**
There is a $\leftrightarrow$ relation, pronounced "commutes to", such that:
$\forall \underline{p} \in \underline{\mathbf{P}}, \underline{q} \in \underline{\mathbf{P}} \cdot$
$\quad (\exists \underline{p'} \in \underline{\mathbf{P}}, \underline{q'} \in \underline{\mathbf{P}} \cdot \langle \underline{p}, \underline{q} \rangle \leftrightarrow \langle \underline{q'}, \underline{p'} \rangle)$
$\vee \ \langle \underline{p}, \underline{q} \rangle \leftrightarrow \mathrm{fail}$

```
unnamedPatchCommute : up_type → up_type → up_type → up_type → Prop
    where "« p , q » <~>u « q' , p' »" := (unnamedPatchCommute p q q' p');
```

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

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

*We can represent this diagramatically thus:*

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

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

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

*For example, $\langle$replace line 3 with "Hello", replace line 3 with "World"$\rangle \leftrightarrow \mathrm{fail}$*

**Axiom 3.2 (unnamed-patch-commute-unique)**
$\forall \underline{p} \in \underline{\mathbf{P}}, \underline{q} \in \underline{\mathbf{P}}, j \in (\underline{\mathbf{P}} \times \underline{\mathbf{P}}) \cup \{\mathrm{fail}\}, k \in (\underline{\mathbf{P}} \times \underline{\mathbf{P}}) \cup \{\mathrm{fail}\} \cdot$
$(\langle \underline{p}, \underline{q} \rangle \leftrightarrow j) \wedge (\langle \underline{p}, \underline{q} \rangle \leftrightarrow k) \Rightarrow j = k$

```
UnnamedPatchCommuteUnique :
        ∀ {p : up_type} {q : up_type}
                  {p' : up_type} {q' : up_type}
                  {p'' : up_type} {q'' : up_type},
        «p, q» <~>u «q', p'»
    → «p, q» <~>u «q'', p''»
    → (p' = p'') ∧ (q' = q'');
```

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

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

**Definition 3.3 (unnamed-patches-commutable)**
We define $\underset{?}{\leftrightarrow}$ to relate two patches if they are commutable, i.e. $\underline{p} \underset{?}{\leftrightarrow} \underline{q} \Leftrightarrow \exists \underline{p'}, \underline{q'} \cdot \langle \underline{p}, \underline{q} \rangle \leftrightarrow \langle \underline{q'}, \underline{p'} \rangle$

```
unnamedPatchCommutable : up_type → up_type → Prop
 := fun (p : up_type) ⇒ fun (q : up_type) ⇒
     (∃ q' : up_type, ∃ p' : up_type,
     «p, q» <~>u «q', p'»)
where "p <~?~>u q" := (unnamedPatchCommutable p q);

unnamedPatchCommutable_dec :
        ∀ (p : up_type) (q : up_type),
        { q' : up_type &
        { p' : up_type &
        «p, q» <~>u «q', p'» }}
        +
        {~(p <~?~>u q)};
```

**Axiom 3.3 (unnamed-patch-commute-self-inverse)**
$\forall \underline{p} \in \mathbf{P}, \underline{q} \in \mathbf{P}, \underline{p'} \in \mathbf{P}, \underline{q'} \in \mathbf{P} \cdot$
$(\langle \underline{p}, \underline{q} \rangle \leftrightarrow \langle \underline{q'}, \underline{p'} \rangle) \Leftrightarrow (\langle \underline{q'}, \underline{p'} \rangle \leftrightarrow \langle \underline{p}, \underline{q} \rangle)$

```
UnnamedPatchCommuteSelfInverse :
        ∀ (p : up_type) (q : up_type)
                  (p' : up_type) (q' : up_type),
        («p, q» <~>u «q', p'») ↔
        («q', p'» <~>u «p, q»);
```

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

**Axiom 3.4 (unnamed-patch-commute-square)**
$\forall \underline{p} \in \underline{\mathbf{P}}, \underline{q} \in \underline{\mathbf{P}}, \forall \underline{p'} \in \underline{\mathbf{P}}, \underline{q'} \in \underline{\mathbf{P}}\cdot$
$(\langle \underline{p}, \underline{q} \rangle \leftrightarrow \langle \underline{q'}, \underline{p'} \rangle) \Leftrightarrow (\langle \underline{q'}^{-1}, \underline{p} \rangle \leftrightarrow \langle \underline{p'}, \underline{q}^{-1} \rangle)$

```
UnnamedPatchCommuteSquare :
      ∀ (p : up_type) (q : up_type)
             (p' : up_type) (q' : up_type),
      «p, q» <~>u «q', p'» →
      « q'^u, p» <~>u «p', q^u»;

(*  XXX UnnamedPatchCommuteConsistent12 copy/pasted from
       unnamed_patch_sequences without accompanying text *)
UnnamedPatchCommuteConsistent1 :
      ∀ {p1 : up_type}
             {q1 : up_type}
             {r1 : up_type}
             {q2 : up_type}
             {r2 : up_type}
             {q3 : up_type}
             {r3 : up_type}
             {p3 : up_type}
             {p5 : up_type},
      «q1, r1» <~>u «r2, q2»
   → «p1, q1» <~>u «q3, p5»
   → «p5, r1» <~>u «r3, p3»
   → ∃ r4 : up_type,
      ∃ q4 : up_type,
      ∃ p6 : up_type,
      «q3, r3» <~>u «r4, q4» ∧
      «p1, r2» <~>u «r4, p6» ∧
      «p6, q2» <~>u «q4, p3»;

UnnamedPatchCommuteConsistent2 :
      ∀ {p3 : up_type}
             {q3 : up_type}
             {r3 : up_type}
             {q4 : up_type}
             {r4 : up_type}
             {q1 : up_type}
             {r1 : up_type}
             {p1 : up_type}
             {p5 : up_type},
      «q3, r3» <~>u «r4, q4»
   → «r3, p3» <~>u «p5, r1»
   → «q3, p5» <~>u «p1, q1»
   → ∃ r2 : up_type,
      ∃ q2 : up_type,
      ∃ p6 : up_type,
      «q1, r1» <~>u «r2, q2» ∧
      «q4, p3» <~>u «p6, q2» ∧
      «r4, p6» <~>u «p1, r2»
}.
```

**Explanation**
*Axiom 3.1 tells us that all patches have an inverse. Therefore we can augment our patch commutation diagram with inverse patches:*

$$\begin{tikzcd}
& \bullet \arrow[dl, bend left, "\underline{p}^{-1}"] \arrow[dr, bend left, "\underline{q}"] & \\
\bullet \arrow[ur, bend left, "\underline{p}"] \arrow[dr, bend right, "\underline{q}'"'] & & \bullet \arrow[ul, bend left, "\underline{q}^{-1}"'] \arrow[dl, bend right, "\underline{p}'^{-1}"'] \\
& \bullet \arrow[ul, bend right, "\underline{q}'^{-1}"'] \arrow[ur, bend right, "\underline{p}'"'] &
\end{tikzcd}$$

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

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

```
Notation "p ^u" := (unnamedPatchInverse _ p).
Notation "« p , q » <~>u « q' , p' »" := (unnamedPatchCommute _ p q q' p').
Notation "p <~?~>u q" := (unnamedPatchCommutable _ p q).
End unnamed_patches.
```

coqdoc

printing <~>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ unnamed_patch_sequences

## 4 Unnamed Patch Sequences

```
Module Export patch_sequences.
Require Import unnamed_patches.
```

**Definition 4.1 (unnamed-patch-sequences)**
We write the empty sequence of unnamed patches as $\epsilon$.

We compose unnamed patch sequences with juxtaposition.

**Definition 4.2 (unnamed-patch-sequence-commute)**
We extend $\leftrightarrow$ to work with combinations of patches and patch sequences in the natural way:

Commuting a patch with a sequence:
$\left\langle \underline{p}, \epsilon \right\rangle \leftrightarrow \left\langle \epsilon, \underline{p} \right\rangle$
$\left\langle \underline{p}, \underline{q\overline{r}} \right\rangle \leftrightarrow \left\langle \underline{q'\overline{r}'}, \underline{p}' \right\rangle$
  if $\left\langle \underline{p}, \underline{q} \right\rangle \leftrightarrow \left\langle \underline{q}', \underline{p}'' \right\rangle$
    $\left\langle \underline{p}'', \underline{\overline{r}} \right\rangle \leftrightarrow \left\langle \underline{\overline{r}'}, \underline{p}' \right\rangle$

Commuting a sequence with a patch:
$\left\langle \epsilon, \underline{p} \right\rangle \leftrightarrow \left\langle \underline{p}, \epsilon \right\rangle$
$\left\langle \underline{\overline{p}q}, \underline{r} \right\rangle \leftrightarrow \left\langle \underline{r'}, \underline{\overline{p}'q'} \right\rangle$
  if $\left\langle \underline{q}, \underline{r} \right\rangle \leftrightarrow \left\langle \underline{r''}, \underline{q'} \right\rangle$
    $\left\langle \underline{\overline{p}}, \underline{r''} \right\rangle \leftrightarrow \left\langle \underline{r'}, \underline{\overline{p}'} \right\rangle$

Commuting a sequence with another sequence:
$\left\langle \underline{\overline{p}}, \underline{\epsilon} \right\rangle \leftrightarrow \left\langle \epsilon, \underline{\overline{p}} \right\rangle$
$\left\langle \epsilon, \underline{\overline{p}} \right\rangle \leftrightarrow \left\langle \underline{\overline{p}}, \epsilon \right\rangle$
$\left\langle \underline{\overline{p}q}, \underline{r\overline{s}} \right\rangle \leftrightarrow \left\langle \underline{r''\overline{s}''}, \underline{\overline{p}''q''} \right\rangle$
  if $\left\langle \underline{q}, \underline{r} \right\rangle \leftrightarrow \left\langle \underline{r'}, \underline{q'} \right\rangle$
    $\left\langle \underline{\overline{p}}, \underline{r'} \right\rangle \leftrightarrow \left\langle \underline{r''}, \underline{\overline{p}'} \right\rangle$
    $\left\langle \underline{q'}, \underline{\overline{s}} \right\rangle \leftrightarrow \left\langle \underline{\overline{s}'}, \underline{q''} \right\rangle$
    $\left\langle \underline{\overline{p}'}, \underline{\overline{s}'} \right\rangle \leftrightarrow \left\langle \underline{\overline{s}''}, \underline{\overline{p}''} \right\rangle$

Note that the first two definitions are degenerate cases of the third.

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

**Axiom 4.1 (unnamed-patch-commute-preserves-commute)**
$\forall \underline{p} \in \underline{\mathbf{P}}, \underline{q} \in \underline{\mathbf{P}}, \underline{r} \in \underline{\mathbf{P}}, \underline{p'} \in \underline{\mathbf{P}}, \underline{q'} \in \underline{\mathbf{P}}, \underline{r'} \in \underline{\mathbf{P}} \cdot$
$(\langle \underline{p}, \underline{qr} \rangle \leftrightarrow \langle \underline{q'r'}, \underline{p'} \rangle) \Rightarrow \left( \left( \underline{q} \underset{?}{\leftrightarrow} \underline{r} \right) \Leftrightarrow \left( \underline{q'} \underset{?}{\leftrightarrow} \underline{r'} \right) \right)$

> **Explanation**
> *This axiom says that if two patches commute then they still commute if you commute something else past both of them.*
>
> *Likewise, if two patches do not commute, then they still do not commute after commuting something else past them.*

**Axiom 4.2 (unnamed-patch-commute-consistent)**
$\forall \underline{p} \in \underline{\mathbf{P}}, \underline{q} \in \underline{\mathbf{P}}, \underline{r} \in \underline{\mathbf{P}}, \underline{p'} \in \underline{\mathbf{P}}, \underline{q'} \in \underline{\mathbf{P}}, \underline{r'} \in \underline{\mathbf{P}} \cdot$
$(\langle \underline{q}, \underline{r} \rangle \leftrightarrow \langle \underline{r'}, \underline{q'} \rangle) \Rightarrow ((\langle \underline{p}, \underline{qr} \rangle \leftrightarrow \langle \_, \underline{p'} \rangle) \Leftrightarrow (\langle \underline{p}, \underline{r'q'} \rangle \leftrightarrow \langle \_, \underline{p'} \rangle))$

```
Axiom UnnamedPatchCommuteConsistent1 :
      ∀ {unnamedPatch : UnnamedPatch}
             {p1 : up_type unnamedPatch}
             {q1 : up_type unnamedPatch}
             {r1 : up_type unnamedPatch}
             {q2 : up_type unnamedPatch}
             {r2 : up_type unnamedPatch}
             {q3 : up_type unnamedPatch}
             {r3 : up_type unnamedPatch}
             {p3 : up_type unnamedPatch}
             {p5 : up_type unnamedPatch},
      «q1, r1» <~>u «r2, q2»
   → «p1, q1» <~>u «q3, p5»
   → «p5, r1» <~>u «r3, p3»
   → (∃ r4 : up_type unnamedPatch,
       (∃ q4 : up_type unnamedPatch,
        (∃ p6 : up_type unnamedPatch,
         «q3, r3» <~>u «r4, q4» ∧
```

```
             «p1, r2» <~>u «r4, p6» ∧
             «p6, q2» <~>u «q4, p3»))).

Axiom UnnamedPatchCommuteConsistent2 :
      ∀ {unnamedPatch : UnnamedPatch}
             {p3 : up_type unnamedPatch}
             {q3 : up_type unnamedPatch}
             {r3 : up_type unnamedPatch}
             {q4 : up_type unnamedPatch}
             {r4 : up_type unnamedPatch}
             {q1 : up_type unnamedPatch}
             {r1 : up_type unnamedPatch}
             {p1 : up_type unnamedPatch}
             {p5 : up_type unnamedPatch},
      «q3, r3» <~>u «r4, q4»
   → «r3, p3» <~>u «p5, r1»
   → «q3, p5» <~>u «p1, q1»
   → (∃ r2 : up_type unnamedPatch,
        (∃ q2 : up_type unnamedPatch,
          (∃ p6 : up_type unnamedPatch,
             «q1, r1» <~>u «r2, q2» ∧
             «q4, p3» <~>u «p6, q2» ∧
             «r4, p6» <~>u «p1, r2»))).
```

**Explanation**
*This axiom says that commuting a patch $p$ past two patches $qr$ gives you the same patch $p'$ regardless of whether $q$ and $r$ have been commuted or not.*

```
End patch_sequences.
```

## 5 Sensible Unnamed Patch Sequences

**Definition 5.1 (sensible-sequences)**
We classify some unnamed patch sequences as *sensible*.

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

**Axiom 5.1 (doing-nothing-is-sensible)**
$\epsilon$ is sensible.

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

**Axiom 5.2 (sensible-subsequences)**
If $\underline{\overline{pq}}$ is sensible then $\underline{\overline{p}}$ is sensible.

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

**Axiom 5.3 (commute-preserves-sensibility)**
If $\underline{\overline{pqrs}}$ is sensible and $\langle \underline{q}, \underline{r} \rangle \leftrightarrow \langle \underline{r'}, \underline{q'} \rangle$, then $\underline{\overline{pr'q's}}$ is sensible.

**Explanation**
*Commuting patches maintains sensibility.*

**Axiom 5.4 (sensible-inverse)**
If $\underline{\overline{pq}}$ is sensible then $\underline{\overline{pqq}}^{-1}$ is sensible.

**Explanation**
*Undoing patches takes us back to an earlier sensible situation.*

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ names

## 6 Names

```
Module Export names.

Require Import Coq.MSets.MSets.
Require Import Coq.Structures.Orders.
Require Import Coq.Structures.OrdersAlt.
```

**Definition 6.1 (names)**
We have a (possibly infinite) set of names **N**.

```
Parameter Name : Set.
Parameter Name_compare : Name → Name → comparison.
Parameter Name_compare_sym : ∀ {x y : Name},
                         Name_compare y x = CompOpp (Name_compare x y).
Parameter Name_compare_trans : ∀ {c : comparison}
                                {x y z : Name},
                          Name_compare x y = c
                        → Name_compare y z = c
                        → Name_compare x z = c.
Parameter Name_eq_leibniz : ∀ {s s' : Name},
                        Name_compare s s' = Eq
                      → s = s'.
Parameter Name_neq_leibniz : ∀ {s s' : Name},
                         Name_compare s s' ≠ Eq
                       → s ≠ s'.

Lemma Name_eq_dec (x y : Name)
               : {x = y} + {˜(x = y)}.
Proof with auto.
case_eq (Name_compare x y).
        intro.
        left.
        apply Name_eq_leibniz...
    intro.
    right.
    apply Name_neq_leibniz.
    intro.
    congruence.
intro.
right.
apply Name_neq_leibniz.
intro.
congruence.
Qed.
```

**Definition 6.2 (names-signed)**
We say that a name is either positive or negative.

```
Inductive Sign : Set
    := Positive | Negative.

Inductive SignedName : Set
    := MkSignedName : ∀ (s : Sign)
                         (n : Name),
                      SignedName.

Definition SignedName_compare (x y : SignedName) : comparison
    := match x, y with
       | MkSignedName Negative _, MkSignedName Positive _ ⇒ Lt
       | MkSignedName Positive _, MkSignedName Negative _ ⇒ Gt
       | MkSignedName Negative x', MkSignedName Negative y' ⇒ Name_compare y' x'
       | MkSignedName Positive x', MkSignedName Positive y' ⇒ Name_compare x' y'
       end.

Lemma SignedName_compare_sym : ∀ {x y : SignedName},
                               SignedName_compare y x = CompOpp (Signed-
Name_compare x y).
Proof with auto.
intros.
destruct x as [sx nx].
destruct y as [sy ny].
destruct sx; destruct sy;
unfold SignedName_compare; auto; apply Name_compare_sym.
Qed.

Lemma SignedName_compare_trans :
    ∀ c x y z,
    (SignedName_compare x y) = c
 → (SignedName_compare y z) = c
 → (SignedName_compare x z) = c.
Proof with auto.
intros.
destruct x as [sx nx].
destruct y as [sy ny].
destruct z as [sz nz].
destruct sx; destruct sy; destruct sz;
unfold SignedName_compare in *...
            apply (Name_compare_trans H H0).
        congruence.
    congruence.
```

```
apply (Name_compare_trans H0 H).
Qed.

Lemma SignedName_eq_leibniz :
    ∀ {x y : SignedName},
    SignedName_compare x y = Eq → x = y.
Proof with auto.
intros.
destruct x as [sx nx].
destruct y as [sy ny].
destruct sx; destruct sy;
unfold SignedName_compare in *.
            apply Name_eq_leibniz in H.
            subst...
        congruence.
    congruence.
apply Name_eq_leibniz in H.
subst...
Qed.

Lemma SignedName_neq_leibniz :
    ∀ {x y : SignedName},
    SignedName_compare x y ≠ Eq → x ≠ y.
Proof with auto.
intros.
destruct x as [sx nx].
destruct y as [sy ny].
destruct sx; destruct sy;
unfold SignedName_compare in *.
            apply Name_neq_leibniz in H.
            intro.
            inversion H0.
            congruence.
        congruence.
    congruence.
apply Name_neq_leibniz in H.
intro.
inversion H0.
congruence.
Qed.

Lemma SignedName_eq_dec (x y : SignedName)
                          : {x = y} + {~(x = y)}.
Proof with auto.
case_eq (SignedName_compare x y).
        intro.
        left.
        apply SignedName_eq_leibniz...
    intro.
    right.
    apply SignedName_neq_leibniz.
    intro.
    congruence.
intro.
```

```
right.
apply SignedName_neq_leibniz.
intro.
congruence.
Qed.
```

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

**Axiom 6.1 (names-invertible)**
$\forall j \in \mathbf{N} \cdot j^{-1} \in \mathbf{N}$.

```
Definition signedNameInverse (n : SignedName) : SignedName
    := match n with
        | MkSignedName Positive b ⇒ MkSignedName Negative b
        | MkSignedName Negative b ⇒ MkSignedName Positive b
        end.
Lemma nameInverseInverse : ∀ (sn : SignedName), signedNameInverse (signedNameInverse
sn) = sn.
Proof with auto.
intros.
destruct sn.
destruct s...
Qed.
Lemma namesInverseInjective: ∀ (sn1: SignedName) (sn2: SignedName),
                                        signedNameInverse sn1 = signedNameInverse sn2 → sn1 =
sn2.
Proof with auto.
intros.
destruct sn1;
destruct s;
destruct sn2;
destruct s;
unfold signedNameInverse in *;
inversion H...
Qed.
```

**Explanation**
*Names have an inverse.*

**Axiom 6.2 (name-inverse-sign)**
If $j \in \mathbf{N}$ is positive then $j^{-1}$ is negative. If $j \in \mathbf{N}$ is negative then $j^{-1}$ is positive.

```
(*  XXX TODO *)
```

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

```
Module NameOrderedTypeAlt.
Definition t := Name.
Definition compare := Name_compare.
Definition compare_sym := @Name_compare_sym.
Definition compare_trans := @Name_compare_trans.
End NameOrderedTypeAlt.

Module NameOrderedType := OT_from_Alt(NameOrderedTypeAlt).

Module NameOrderedTypeWithLeibniz.
Include NameOrderedType.
Definition eq_leibniz := @Name_eq_leibniz.
End NameOrderedTypeWithLeibniz.

Module NameSetMod := MSetList.MakeWithLeibniz(NameOrderedTypeWithLeibniz).

Notation NameSet := (NameSetMod.t).
Notation NameSetIn := (NameSetMod.In).
Notation NameSetAdd := (NameSetMod.add).
Notation NameSetRemove := (NameSetMod.remove).
Notation NameSetEqual := (NameSetMod.Equal).

Module NameSetDec := WDecide (NameSetMod).
Ltac nameSetDec := NameSetDec.fsetdec.

Definition NameSetEquality {s s' : NameSet}
                                      (H : NameSetEqual s s')
                                    : s = s'
      := NameSetMod.eq_leibniz H.

Lemma NameSet_eq_dec : ∀ (x y : NameSet),
                                  {x = y} + {x ≠ y}.
Proof with auto.
intros.
destruct (NameSetMod.eq_dec x y).
    left.
    apply NameSetEquality...
right.
intro.
subst.
elim n.
reflexivity.
Qed.

Module SignedNameOrderedTypeAlt.
Definition t := SignedName.
Definition compare := SignedName_compare.
Definition compare_sym := @SignedName_compare_sym.
Definition compare_trans := @SignedName_compare_trans.
End SignedNameOrderedTypeAlt.

Module SignedNameOrderedType := OT_from_Alt(SignedNameOrderedTypeAlt).

Module SignedNameOrderedTypeWithLeibniz.
Include SignedNameOrderedType.
Definition eq_leibniz := @SignedName_eq_leibniz.
End SignedNameOrderedTypeWithLeibniz.

Module SignedNameSetMod := MSetList.MakeWithLeibniz(SignedNameOrderedTypeWithLeibniz).

Notation SignedNameSet := (SignedNameSetMod.t).
Notation SignedNameSetIn := (SignedNameSetMod.In).
Notation SignedNameSetAdd := (SignedNameSetMod.add).
Notation SignedNameSetRemove := (SignedNameSetMod.remove).
Notation SignedNameSetEqual := (SignedNameSetMod.Equal).

Module SignedNameSetDec := WDecide (SignedNameSetMod).
Ltac signedNameSetDec := SignedNameSetDec.fsetdec.

Definition SignedNameSetEquality {s s' : SignedNameSet}
                                  (H : SignedNameSetEqual s s')
                                : s = s'
    := SignedNameSetMod.eq_leibniz H.

Lemma SignedNameSet_eq_dec : ∀ (x y : SignedNameSet),
                             {x = y} + {x ≠ y}.
Proof with auto.
intros.
destruct (SignedNameSetMod.eq_dec x y).
    left.
    apply SignedNameSetEquality...
right.
intro.
subst.
elim n.
reflexivity.
Qed.

End names.
```

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ named_patches

## 7 Named patches

```
Module Export named_patches.

Require Import Equality.
Require Import ProofIrrelevance.

Require Import util.
Require Import unnamed_patches.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import commute_square.
```

Rather than working directly with unnamed patches, we will work with named patches. A named patch is built out of a name and an unnamed patch. For the motivation for this decision, see Appendix B.

**Definition 7.1 (named-patches)**
If $j \in \mathbf{N}$ and $\underline{p} \in \underline{\mathbf{P}}$, then $[\![j, \underline{p}]\!]$ is a *named patch*.

We define a (possibly infinite) set of *named patches* $\mathbf{P}$ thus: $\forall j \in \mathbf{N}, \underline{p} \in \underline{\mathbf{P}} \cdot [\![j, p]\!] \in \mathbf{P}$.

```
Record NamedPatch (unnamedPatch : UnnamedPatch)
                  (from to : NameSet) : Type := mkNamedPatch {
    np_unnamedPatch : up_type unnamedPatch;
    np_name : SignedName;
    np_nameOK : patchNamesOK from to np_name
}.
Implicit Arguments np_unnamedPatch [unnamedPatch from to].
Implicit Arguments np_name [unnamedPatch from to].
Implicit Arguments np_nameOK [unnamedPatch from to].
Implicit Arguments mkNamedPatch [unnamedPatch from to].
```

From here on, the unqualified term "patches" will refer to named patches.

**Definition 7.2 (patch-name)**
We define $n$ to tell us the name of a patch, i.e. $n\left([\![j, \underline{p}]\!]\right) = j$.

**Definition 7.3 (unnamed-patch-of)**
We define $\lfloor p \rfloor$ to tell us the unnamed patch of a patch, i.e. $\left\lfloor [\![j, \underline{q}]\!] \right\rfloor = \underline{q}$.

**Definition 7.4 (named-patch-inverse)**
$[\![j, \underline{p}]\!]^{-1} = [\![j^{-1}, \underline{p}^{-1}]\!]$

**Explanation**
*The inverse of a named patch has the inverse name, and the inverse effect.*

```
(*  XXX Should this be elsewhere?: *)
Lemma patchNamesOKInverse : ∀ {from to : NameSet}
                                {n : SignedName}
                                (namesOK : patchNamesOK from to n),
                           patchNamesOK to from (signedNameInverse n).
Proof with auto.
intros.
destruct n.
destruct s.
    simpl.
    destruct namesOK.
    split...
simpl.
destruct namesOK.
split...
Qed.

Definition namedPatchInverse {unnamedPatch : UnnamedPatch}
                             {from to : NameSet}
                             (p : NamedPatch unnamedPatch from to)
                           : NamedPatch unnamedPatch to from
(*  XXX Shouldn't unnamedPatchInverse's first argument be implicit? *)
 := mkNamedPatch (unnamedPatchInverse _ (np_unnamedPatch p))
                 (signedNameInverse (np_name p))
                 (patchNamesOKInverse (np_nameOK p)).
Notation "p ^n" := (namedPatchInverse p)
```

```
    (at level 10).
```

**Definition 7.5 (named-patch-commute)**
We extend $\leftrightarrow$ to named patches thus:
$(\langle [\![j,\underline{p}]\!]\,,[\![k,\underline{q}]\!]\rangle \leftrightarrow \langle [\![k,\underline{q'}]\!]\,,[\![j,\underline{p'}]\!]\rangle) \Leftrightarrow (j \neq k) \wedge (j \neq k^{-1}) \wedge (\langle p,q\rangle \leftrightarrow \langle q',p'\rangle)$

```
Inductive namedPatchCommute {unnamedPatch : UnnamedPatch}
                            {from mid1 mid2 to : NameSet}
        : NamedPatch unnamedPatch from mid1
        → NamedPatch unnamedPatch mid1 to
        → NamedPatch unnamedPatch from mid2
        → NamedPatch unnamedPatch mid2 to
        → Prop
    := MkNamedPatchCommute : ∀ (up : up_type unnamedPatch) (uq : up_type unnamed-
Patch)
                                 (up' : up_type unnamedPatch) (uq' : up_type un-
namedPatch)
                                 (np : SignedName) (nq : SignedName)
                                 (not_equal : np ≠ nq)
                                 (not_inverse : np ≠ signedNameInverse nq)
                                 (pOK : patchNamesOK from mid1 np)
                                 (qOK : patchNamesOK mid1 to nq)
                                 (q'OK : patchNamesOK from mid2 nq)
                                 (p'OK : patchNamesOK mid2 to np),
               unnamedPatchCommute unnamedPatch up uq uq' up'
            → namedPatchCommute (mkNamedPatch up np pOK)
                                (mkNamedPatch uq nq qOK)
                                (mkNamedPatch uq' nq q'OK)
                                (mkNamedPatch up' np p'OK).
Notation "« p , q » <~>n « q' , p' »"
    := (namedPatchCommute p q q' p')
    (at level 60, no associativity).
```

**Explanation**
*Named patches do not commute with themself or their inverse. Other than that, they commute iff their unnamed patches commute.*

**Lemma 7.1 (named-patch-commute-unique)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, j \in (\mathbf{P} \times \mathbf{P}) \cup \{\text{fail}\}, k \in (\mathbf{P} \times \mathbf{P}) \cup \{\text{fail}\} \cdot$
$(\langle p,q\rangle \leftrightarrow j) \wedge (\langle p,q\rangle \leftrightarrow k) \Rightarrow j = k$

```
(*  We have to split this into 2 parts in the coq, as we can't
   directly say that q' = q'' as they have different types. *)
Lemma NamedPatchCommuteUnique :
      ∀ {unnamedPatch : UnnamedPatch}
             {from mid mid' mid'' to : NameSet}
             {p : NamedPatch unnamedPatch from mid} {q : NamedPatch unnamedPatch
mid to}
             {q' : NamedPatch unnamedPatch from mid'} {p' : NamedPatch unnamedPatch
mid' to}
             {q'' : NamedPatch unnamedPatch from mid''} {p'' : NamedPatch unnamed-
Patch mid'' to},
        «p, q» <~>n «q', p'»
    → «p, q» <~>n «q'', p''»
    → (mid' = mid'') ∧ (p' ~= p'') ∧ (q' ~= q'').
Proof with auto.
intros unnamedPatch from mid mid' mid'' to p q q' p' q'' p'' commute' commute''.
dependent destruction commute'.
dependent destruction commute''.
assert (mid' = mid'').
    apply NameSetEquality.
    clear - q'OK q'OK0.
    destruct nq.
    destruct s.
        unfold patchNamesOK in *.
        inversion_clear q'OK.
        inversion_clear q'OK0.
        nameSetDec.
    unfold patchNamesOK in *.
    admit.
    (*  nameSetDec. *)
split...
subst.
destruct (UnnamedPatchCommuteUnique unnamedPatch H H0).
subst.
proofIrrel p'OK p'OK0.
proofIrrel q'OK q'OK0.
split...
Qed.
```

**Explanation**
*This is Axiom 3.2 restated for named patches.*

**Proof**
If $n\,(p) = n\,(q)$ or $n\,(p) = n\,(q)^{\,-1}$ then $j = k =$ fail.

Otherwise the result follows from Axiom 3.2 applied to the commute of the unnamed patches. ∎

**Definition 7.6 (named-patches-commutable)**
We extend $\underset{?}{\leftrightarrow}$ to relate two named patches if they are commutable, i.e.

$p \underset{?}{\leftrightarrow} q \Leftrightarrow \exists p', q' \cdot \langle p, q\rangle \leftrightarrow \langle q', p'\rangle$

```
Definition namedPatchCommutable {unnamedPatch : UnnamedPatch}
                                {from mid1 to : NameSet}
                                (p : NamedPatch unnamedPatch from mid1)
                                (q : NamedPatch unnamedPatch mid1 to) : Prop
 := ∃ mid2 : NameSet,
    ∃ q' : NamedPatch unnamedPatch from mid2,
    ∃ p' : NamedPatch unnamedPatch mid2 to,
    «p, q» <~>n «q', p'».
Notation "p <~?~> q" := (namedPatchCommutable p q)
    (at level 60, no associativity).

Lemma namedPatchCommutable_dec :
```

```
          ∀ {unnamedPatch : UnnamedPatch}
                  {from mid to : NameSet}
                  (p : NamedPatch unnamedPatch from mid)
                  (q : NamedPatch unnamedPatch mid to),
          {mid2 : NameSet &
          { q' : NamedPatch unnamedPatch from mid2 &
          { p' : NamedPatch unnamedPatch mid2 to &
          «p, q» <~>n «q', p'» }}}
        + {~(p <~?~> q)}.
Proof with auto.
intros.
dependent destruction p.
dependent destruction q.
rename np_name0 into snp.
rename np_name1 into snq.
rename np_unnamedPatch0 into up.
rename np_unnamedPatch1 into uq.
destruct (SignedNameOrderedTypeWithLeibniz.eq_dec snp snq).
    (*  This is the case where the two patches have the same name *)
    right.
    intro.
    dependent destruction H.
    dependent destruction H.
    dependent destruction H.
    dependent destruction H.
    apply SignedName_eq_leibniz in e.
    congruence.
destruct (SignedNameOrderedTypeWithLeibniz.eq_dec snp (signedNameInverse snq)).
    (*  This is the case where the two patches have inverse names *)
    right.
    intro.
    dependent destruction H.
    dependent destruction H.
    dependent destruction H.
    dependent destruction H.
    apply SignedName_eq_leibniz in e.
    congruence.
(*  In the remaining cases, whether the named patches commute is
   determined by whether the unnamed patches commute *)
destruct (unnamedPatchCommutable_dec unnamedPatch up uq).
    (*  This is the case where the unnamed patches commute *)
    left.
    destruct s as [uq' [up' up_uq_commute_uq'_up']].
    destruct snq.
    rename s into sq, n1 into nq.
    (*  XXX Can we get coq to leave this for us?: *)
    set (snq := MkSignedName sq nq).
    destruct snp.
    rename s into sp, n1 into np.
    (*  XXX Can we get coq to leave this for us?: *)
    set (snp := MkSignedName sp np).
    destruct sq.
```

```
      (*  This is the case where q is a positive patch *)
      remember (NameSetAdd nq from) as oq.
      ∃ oq.
      assert (q'OK : patchNamesOK from oq snq).
          destruct sp.
              simpl in *.
              split.
                  nameSetDec.
              nameSetDec.
          simpl in *.
          subst. (*  XXX Shouldn't be needed *)
          split.
              admit.
              (*  nameSetDec. *)
          nameSetDec.
      ∃ (mkNamedPatch uq' snq q'OK).
      assert (p'OK : patchNamesOK oq to snp).
          destruct sp.
              simpl in *.
              subst. (*  XXX Shouldn't be needed *)
              split.
                  clear - np_nameOK0 n.
                  admit. (*  nameSetDec. *)
              clear - np_nameOK0 np_nameOK1.
              nameSetDec.
          simpl in *.
          subst. (*  XXX Shouldn't be needed *)
          split.
              clear - np_nameOK0 np_nameOK1 n0.
              admit. (*  nameSetDec. *)
          clear - np_nameOK0 np_nameOK1 n0.
          nameSetDec.
      ∃ (mkNamedPatch up' snp p'OK).
      apply MkNamedPatchCommute...
          admit.
      admit.
  (*  This is the case where q is a negative patch *)
  remember (NameSetRemove nq from) as oq.
  ∃ oq.
  assert (q'OK : patchNamesOK from oq snq).
      destruct sp.
          simpl in *.
          subst. (*  XXX Shouldn't be needed *)
          split.
              nameSetDec.
          admit. (*  nameSetDec. *)
      simpl in *.
      subst. (*  XXX Shouldn't be needed *)
      split.
          nameSetDec.
      admit. (*  nameSetDec. *)
  ∃ (mkNamedPatch uq' snq q'OK).
```

```
        assert (p'OK : patchNamesOK oq to snp).
            destruct sp.
                simpl in *.
                subst. (*  XXX Shouldn't be needed *)
                split.
                    nameSetDec.
                destruct np_nameOK0 as [? HX1].
                destruct np_nameOK1 as [? HX2].
                destruct q'OK as [? HX3].
                assert (HX4 : NameSetIn nq from).
                    clear - HX3.
                    nameSetDec.
                clear - HX1 HX2 HX4 n0 H1.
                admit. (*  nameSetDec. *)
            simpl in *.
            subst. (*  XXX Shouldn't be needed *)
            split.
                nameSetDec.
            destruct np_nameOK0 as [? HX1].
            destruct np_nameOK1 as [? HX2].
            (*
            XXX
            clear - HX1 HX2 H0 H1 n.
            *)
            admit. (*  nameSetDec. *)
        ∃ (mkNamedPatch up' snp p'OK).
        apply MkNamedPatchCommute...
            admit.
        admit.
    (*  This is the case where the unnamed patches do not commute *)
    right.
    intro.
    dependent destruction H.
    dependent destruction H.
    dependent destruction H.
    dependent destruction H.
    assert (up <~?~>u uq).
        unfold unnamedPatchCommutable.
        ∃ uq'.
        ∃ up'...
    congruence.
    Qed.
```

```
Instance NamedPartPatchUniverse (unnamedPatch : UnnamedPatch)
         : PartPatchUniverse (NamedPatch unnamedPatch) (NamedPatch unnamedPatch)
    := mkPartPatchUniverse
           (NamedPatch unnamedPatch)
           (NamedPatch unnamedPatch)
           (@namedPatchCommute _)
           (@namedPatchCommutable_dec _)
```

```
            (@NamedPatchCommuteUnique _).
```

**Lemma 7.2 (named-patch-commute-self-inverse)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, p' \in \mathbf{P}, q' \in \mathbf{P}\cdot$
$(\langle p,q\rangle \leftrightarrow \langle q',p'\rangle) \Leftrightarrow (\langle q',p'\rangle \leftrightarrow \langle p,q\rangle)$

```
Lemma NamedPatchCommuteSelfInverse :
      ∀ {unnamedPatch : UnnamedPatch}
            {from mid1 mid2 to : NameSet}
            {p : NamedPatch unnamedPatch from mid1}
            {q : NamedPatch unnamedPatch mid1 to}
            {q' : NamedPatch unnamedPatch from mid2}
            {p' : NamedPatch unnamedPatch mid2 to}
            (namedCommute : «p, q» <~> «q', p'»),
      («q', p'» <~> «p, q»).
Proof with auto.
intros.
destruct namedCommute.
apply UnnamedPatchCommuteSelfInverse in H.
simpl commute.
apply MkNamedPatchCommute...
intro.
elim not_inverse.
apply namesInverseInjective.
rewrite nameInverseInverse...
Qed.
```

**Explanation**
*This is Axiom 3.3 restated for named patches.*

**Proof**
If $n(p) = n(q)$ or $n(p) = n(q)^{-1}$ then both commutes fail, so the result holds.

Otherwise the result follows from Axiom 3.3 applied to the commute of the unnamed patches. ■

**Lemma 7.3 (named-patch-commute-square)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, \forall p' \in \mathbf{P}, q' \in \mathbf{P}\cdot$
$(\langle p,q\rangle \leftrightarrow \langle q',p'\rangle) \Leftrightarrow \left(\langle q'^{-1},p\rangle \leftrightarrow \langle p',q^{-1}\rangle\right)$

**Explanation**
*This is Axiom 3.4 restated for named patches.*

**Proof**
If $n(p) = n(q)$ or $n(p) = n(q)^{-1}$ then both commutes fail, so the result holds.

Otherwise the result follows from Axiom 3.4 applied to the commute of the unnamed patches. ■

```
(*  XXX Move this somewhere more general *)
Lemma monotonicNeq : ∀ {t u : Set}
                              (f : t → u)
```

```
                                        (x y : t),
                                (f x ≠ f y)
                             → (x ≠ y).
Proof.
intros.
intro.
elim H.
subst.
auto.
Qed.

Instance NamedPatchUniverseInv
                {unnamedPatch : UnnamedPatch}
                (namedPartPatchUniverse : PartPatchUniverse
                                                    (NamedPatch unnamedPatch)
                                                    (NamedPatch unnamedPatch))
              : PatchUniverseInv (NamedPartPatchUniverse unnamedPatch)
                                 (NamedPartPatchUniverse unnamedPatch)
    := mkPatchUniverseInv
            (NamedPatch unnamedPatch)
            (NamedPatch unnamedPatch)
            (NamedPartPatchUniverse unnamedPatch)
            (NamedPartPatchUniverse unnamedPatch)
            (@NamedPatchCommuteSelfInverse unnamedPatch).
```

```
(*  XXX NamedPatchCommuteConsistent12 copy/pasted from
    patch_sequences without the accompanying text *)
Lemma NamedPatchCommuteConsistent1 :
      ∀ {unnamedPatch : UnnamedPatch}
              {o op opq opqr opr oq oqr : NameSet}
              {p1 : NamedPatch unnamedPatch o op}
              {q1 : NamedPatch unnamedPatch op opq}
              {r1 : NamedPatch unnamedPatch opq opqr}
              {q2 : NamedPatch unnamedPatch opr opqr}
              {r2 : NamedPatch unnamedPatch op opr}
              {q3 : NamedPatch unnamedPatch o oq}
              {r3 : NamedPatch unnamedPatch oq oqr}
              {p3 : NamedPatch unnamedPatch oqr opqr}
              {p5 : NamedPatch unnamedPatch oq opq},
      «q1, r1» <~> «r2, q2»
   → «p1, q1» <~> «q3, p5»
   → «p5, r1» <~> «r3, p3»
   → (∃ or : NameSet,
        (∃ r4 : NamedPatch unnamedPatch o or,
         (∃ q4 : NamedPatch unnamedPatch or oqr,
          (∃ p6 : NamedPatch unnamedPatch or opr,
           «q3, r3» <~> «r4, q4» ∧
           «p1, r2» <~> «r4, p6» ∧
           «p6, q2» <~> «q4, p3»)))).
Proof with auto.
```

```
intros unnamedPatch o op opq opqr opr oq oqr p1 q1 r1 q2 r2 q3 r3 p3 p5
       q1_r1_commutes_r2_q2
       p1_q1_commutes_q3_p5
       p5_r1_commutes_r2_p3.
simpl commute in *.
dependent destruction q1_r1_commutes_r2_q2.
dependent destruction p1_q1_commutes_q3_p5.
dependent destruction p5_r1_commutes_r2_p3.
rename nq into nr.
rename np into nq.
rename np0 into np.
destruct np as [sp np].
destruct nq as [sq nq].
destruct nr as [sr nr].
(*  XXX Can we get coq to leave these for us?: *)
set (snr := MkSignedName sr nr).
set (snq4 := MkSignedName sq nq).
set (snp6 := MkSignedName sp np).
destruct (UnnamedPatchCommuteConsistent1 _ H H0 H1)
  as [r4 [q4 [p6 [H3 [H4 H5]]]]].
destruct sr.
    (*  This is the case where r is a positive patch *)
    remember (NameSetAdd nr o) as or.
    ∃ or.
    assert (snrPatchNamesOK : patchNamesOK o or snr).
        subst.
        simpl in *.
        split.
            clear - q'OK pOK0 not_equal1 not_inverse1.
            destruct sp.
                nameSetDec.
            admit. (*  nameSetDec. *)
        clear.
        nameSetDec.
    ∃ (mkNamedPatch r4 snr snrPatchNamesOK).
    assert (q4PatchNamesOK : patchNamesOK or oqr snq4).
        subst.
        simpl in *.
        destruct sq.
            split.
                clear - not_equal q'OK0.
                admit. (*  nameSetDec. *)
            clear - q'OK1 q'OK0.
            nameSetDec.
        split.
            clear - not_inverse q'OK1 q'OK0.
            admit. (*  nameSetDec. *)
        clear - not_inverse q'OK1 q'OK0.
        nameSetDec.
    ∃ (mkNamedPatch q4 snq4 q4PatchNamesOK).
    assert (p6PatchNamesOK : patchNamesOK or opr snp6).
        subst.
        simpl in *.
        destruct sp.
            split.
                clear - pOK0 not_equal1.
                admit. (*  nameSetDec. *)
            clear - q'OK pOK0.
            nameSetDec.
        split.
            clear - pOK0 q'OK not_inverse1.
            admit. (*  nameSetDec. *)
        clear - pOK0 q'OK not_inverse1.
        nameSetDec.
    ∃ (mkNamedPatch p6 snp6 p6PatchNamesOK).
    split.
        apply MkNamedPatchCommute...
    split.
        apply MkNamedPatchCommute...
    apply MkNamedPatchCommute...
(*  This is the case where r is a negative patch *)
remember (NameSetRemove nr o) as or.
∃ or.
assert (snrPatchNamesOK : patchNamesOK o or snr).
    subst.
    simpl in *.
    split.
        clear.
        nameSetDec.
    assert (HX : NameSetIn nr o).
        clear - q'OK pOK0 not_equal1 not_inverse1.
        destruct sp.
            admit. (*  nameSetDec. *)
        nameSetDec.
    clear - HX.
    admit. (*  nameSetDec. *)
∃ (mkNamedPatch r4 snr snrPatchNamesOK).
assert (q4PatchNamesOK : patchNamesOK or oqr snq4).
    subst.
    simpl in *.
    destruct sq.
        split.
            clear - q'OK0.
            nameSetDec.
        clear - not_inverse q'OK1 q'OK0.
        admit. (*  nameSetDec. *)
    split.
        clear - q'OK1 q'OK0.
        nameSetDec.
    clear - q'OK1 q'OK0.
    admit. (*  nameSetDec. *)
∃ (mkNamedPatch q4 snq4 q4PatchNamesOK).
assert (p6PatchNamesOK : patchNamesOK or opr snp6).
    subst.
    simpl in *.
    destruct sp.
        split.
            clear - pOK0.
            nameSetDec.
        clear - q'OK pOK0 not_inverse1.
        admit. (* nameSetDec. *)
    split.
        clear - pOK0 q'OK.
        nameSetDec.
    clear - pOK0 q'OK.
    admit. (* nameSetDec. *)
∃ (mkNamedPatch p6 snp6 p6PatchNamesOK).
split.
    apply MkNamedPatchCommute...
split.
    apply MkNamedPatchCommute...
apply MkNamedPatchCommute...
Qed.
Lemma NamedPatchCommuteConsistent2 :
      ∀ {unnamedPatch : UnnamedPatch}
              {o op opq opqr or oq oqr : NameSet}
              {q3 : NamedPatch unnamedPatch o oq}
              {r3 : NamedPatch unnamedPatch oq oqr}
              {p3 : NamedPatch unnamedPatch oqr opqr}
              {q4 : NamedPatch unnamedPatch or oqr}
              {r4 : NamedPatch unnamedPatch o or}
              {p1 : NamedPatch unnamedPatch o op}
              {q1 : NamedPatch unnamedPatch op opq}
              {r1 : NamedPatch unnamedPatch opq opqr}
              {p5 : NamedPatch unnamedPatch oq opq},
      «q3, r3» <~> «r4, q4»
   → «r3, p3» <~> «p5, r1»
   → «q3, p5» <~> «p1, q1»
   → (∃ opr : NameSet,
        (∃ r2 : NamedPatch unnamedPatch op opr,
         (∃ q2 : NamedPatch unnamedPatch opr opqr,
          (∃ p6 : NamedPatch unnamedPatch or opr,
           «q1, r1» <~> «r2, q2» ∧
           «q4, p3» <~> «p6, q2» ∧
           «r4, p6» <~> «p1, r2»)))).
Proof with auto.
intros.
simpl commute in *.
dependent destruction H.
dependent destruction H0.
dependent destruction H1.
rename nq into nr.
rename np into nq.
rename nq0 into np.
destruct np as [sp np].
destruct nq as [sq nq].
destruct nr as [sr nr].
(*  XXX Can we get coq to leave this for us?: *)
set (snr2 := MkSignedName sr nr).
set (snq2 := MkSignedName sq nq).
set (snp6 := MkSignedName sp np).
destruct (UnnamedPatchCommuteConsistent2 _ H H0 H1)
  as [ur2 [uq2 [up6 [? [? ?]]]]].
destruct sr.
    (*  This is the case where r is a positive patch *)
    remember (NameSetAdd nr op) as opr.
    ∃ opr.
    assert (snrPatchNamesOK : patchNamesOK op opr snr2).
        subst.
        simpl in *.
        split.
            clear - q'OK q'OK1 not_equal0 not_inverse0.
            destruct sp.
                admit. (*  nameSetDec. *)
            nameSetDec.
        clear.
        nameSetDec.
    ∃ (mkNamedPatch ur2 snr2 snrPatchNamesOK).
    assert (q2PatchNamesOK : patchNamesOK opr opqr snq2).
        subst.
        simpl in *.
        destruct sq.
            split.
                clear - not_equal p'OK1.
                admit. (*  nameSetDec. *)
            clear - p'OK0 p'OK1.
            nameSetDec.
        split.
            clear - not_inverse p'OK1 p'OK0.
            admit. (*  nameSetDec. *)
        clear - not_inverse p'OK1 p'OK0.
        nameSetDec.
    ∃ (mkNamedPatch uq2 snq2 q2PatchNamesOK).
    assert (p6PatchNamesOK : patchNamesOK or opr snp6).
        subst.
        simpl in *.
        destruct sp.
            split.
                clear - q'OK q'OK1 not_equal0.
                admit. (*  nameSetDec. *)
            clear - q'OK q'OK1.
            nameSetDec.
        split.
            clear - q'OK1 not_inverse0.
            admit. (*  nameSetDec. *)
        clear - q'OK q'OK1 not_inverse0.
        nameSetDec.
    ∃ (mkNamedPatch up6 snp6 p6PatchNamesOK).
    split.
        apply MkNamedPatchCommute...
    split.
        apply MkNamedPatchCommute...
    apply MkNamedPatchCommute...
(*  This is the case where r is a negative patch *)
remember (NameSetRemove nr op) as opr.
∃ opr.
assert (snrPatchNamesOK : patchNamesOK op opr snr2).
    subst.
    simpl in *.
    split.
        clear.
        nameSetDec.
    assert (HX : NameSetIn nr op).
        clear - q'OK1 q'OK not_equal0 not_inverse0.
        destruct sp.
            nameSetDec.
        admit. (*  nameSetDec. *)
    clear - HX.
    admit. (*  nameSetDec. *)
∃ (mkNamedPatch ur2 snr2 snrPatchNamesOK).
assert (q2PatchNamesOK : patchNamesOK opr opqr snq2).
    subst.
    simpl in *.
    destruct sq.
        split.
            clear - p'OK1.
            nameSetDec.
        clear - p'OK0 p'OK1 not_inverse.
        admit. (*  nameSetDec. *)
    split.
        clear - p'OK1 p'OK0.
        nameSetDec.
    clear - p'OK1 p'OK0.
    admit. (*  nameSetDec. *)
∃ (mkNamedPatch uq2 snq2 q2PatchNamesOK).
assert (p6PatchNamesOK : patchNamesOK or opr snp6).
    subst.
    simpl in *.
    destruct sp.
        split.
            clear - q'OK q'OK1.
            nameSetDec.
        clear - q'OK q'OK1 not_inverse0.
        admit. (*  nameSetDec. *)
    split.
        clear - q'OK1.
        nameSetDec.
    assert (HX : NameSetIn nr op).
        clear - snrPatchNamesOK.
            nameSetDec.
    clear - q'OK q'OK1 HX.
    admit. (*  nameSetDec. *)
∃ (mkNamedPatch up6 snp6 p6PatchNamesOK).
split.
    apply MkNamedPatchCommute...
split.
    apply MkNamedPatchCommute...
apply MkNamedPatchCommute...
Qed.

(*  XXX unused: *)
Lemma NamedPatchContexts : ∀ {unnamedPatch : UnnamedPatch}
                                  {from to : NameSet}
                                  (p : NamedPatch unnamedPatch from to),
                           patchNamesOK from to (np_name p).
Proof with auto.
intros.
destruct p.
unfold np_name.
unfold patchNamesOK in *.
destruct np_name0.
destruct s; split; nameSetDec.
Qed.

Lemma NamedPatchCommuteNames :
      ∀ {unnamedPatch : UnnamedPatch}
             {from mid1 mid2 to : NameSet}
             {p : NamedPatch unnamedPatch from mid1} {q : NamedPatch unnamedPatch
mid1 to}
             {q' : NamedPatch unnamedPatch from mid2} {p' : NamedPatch unnamedPatch
mid2 to},
      « p , q » <~> « q' , p' »
   → (np_name p = np_name p') ∧
      (np_name q = np_name q') ∧
      (np_name p ≠ np_name q).
Proof with auto.
intros.
destruct H.
unfold np_name...
Qed.

Instance NamedPatchUniverse {unnamedPatch : UnnamedPatch}
                               {namedPartPatchUniverse : PartPatchUniverse (NamedPatch
unnamedPatch) (NamedPatch unnamedPatch)}
                               (namedPatchUniverseInv : PatchUniverseInv
                                                             (NamedPartPatchUniverse
unnamedPatch)
                                                             (NamedPartPatchUniverse
unnamedPatch))
 : PatchUniverse (NamedPatchUniverseInv namedPartPatchUniverse)
    := mkPatchUniverse
          (NamedPatch unnamedPatch)
          (NamedPartPatchUniverse unnamedPatch)
          (NamedPatchUniverseInv namedPartPatchUniverse)
```

```
            (@np_name _)
            (@NamedPatchCommuteConsistent1 _)
            (@NamedPatchCommuteConsistent2 _)
            (@NamedPatchCommuteNames _).
Lemma NamedPatchInvertInverse :
      ∀ {unnamedPatch : UnnamedPatch}
             {from to : NameSet}
             (p : NamedPatch unnamedPatch from to),
      (p ^n) ^n = p.
Proof with auto.
intros.
destruct p.
unfold namedPatchInverse.
simpl.
rewrite unnamedPatchInvertInverse.
destruct np_name0.
destruct s.
    simpl.
    f_equal.
    apply proof_irrelevance.
simpl.
f_equal.
apply proof_irrelevance.
Qed.
Lemma NamedPatchNameOfInverse
          {unnamedPatch : UnnamedPatch}
          {from to : NameSet}
          (p : NamedPatch unnamedPatch from to)
      : np_name (p ^n) = signedNameInverse (np_name p).
Proof.
auto.
Qed.
Instance NamedInvertiblePatchlike (unnamedPatch : UnnamedPatch)
 : InvertiblePatchlike (NamedPatch unnamedPatch)
    := mkInvertiblePatchlike
            (NamedPatch unnamedPatch)
            (@namedPatchInverse _)
            (@NamedPatchInvertInverse _).
Lemma NamedPatchCommuteSquare :
      ∀ {unnamedPatch : UnnamedPatch}
             {o op oq opq : NameSet}
             {p : NamedPatch unnamedPatch o op}
             {q : NamedPatch unnamedPatch op opq}
             {q' : NamedPatch unnamedPatch o oq}
             {p' : NamedPatch unnamedPatch oq opq},
      «p, q» <~> «q', p'» →
      « q'^, p» <~> «p', q^».
Proof with auto.
intros.
destruct H.
apply UnnamedPatchCommuteSquare in H.
```

```
constructor.
        simpl...
    simpl.
    apply (monotonicNeq signedNameInverse).
    rewrite nameInverseInverse.
    rewrite nameInverseInverse...
simpl...
Qed.

Instance NamedCommuteSquare
               (unnamedPatch : UnnamedPatch)
 : CommuteSquare (NamedPatch unnamedPatch) (NamedPatch unnamedPatch)
    := mkCommuteSquare
           (NamedPatch unnamedPatch)
           (NamedPatch unnamedPatch)
           _
           _
           _
           (@NamedPatchCommuteSquare _).

Instance NamedInvertiblePatchUniverse
               {unnamedPatch : UnnamedPatch}
               {namedPartPatchUniverse : PartPatchUniverse (NamedPatch unnamedPatch)
(NamedPatch unnamedPatch)}
               {namedPatchUniverseInv : PatchUniverseInv (NamedPartPatchUniverse un-
namedPatch) (NamedPartPatchUniverse unnamedPatch)}
 : InvertiblePatchUniverse
        (NamedPatchUniverse namedPatchUniverseInv)
        (NamedInvertiblePatchlike unnamedPatch)
    := mkInvertiblePatchUniverse
           (NamedPatch unnamedPatch)
           (NamedPartPatchUniverse unnamedPatch)
           (NamedPatchUniverseInv namedPartPatchUniverse)
           (NamedPatchUniverse namedPatchUniverseInv)
           (NamedInvertiblePatchlike unnamedPatch)
           (@NamedPatchNameOfInverse _).

End named_patches.
```

## 8 Patch Merge

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

We will give two descriptions on how merging works. The first, sequence-wise merge, is much simpler, but the second, patch-wise merge, is closer to what we have to do once conflicts are involved. Whichever merge algorithm is chosen, we first do merge preparation.

### 8.1 Merge preparation

Suppose we have two patch sequences (which could be entire repos) $\overline{p}$ and $\overline{q}$. They start from the same state (if they are repos, then the empty state), and then apply their respective patch sequences:

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

Then before we can merge these two repositories, we first need to commute their common patches to a prefix, i.e. we commute the patches within $\overline{p}$ and $\overline{q}$ to create:

$$\begin{tikzcd}
 & \bullet \arrow[d, "\overline{r}"'] & \\
 & \bullet \arrow[dl, bend right, "\overline{s}"'] \arrow[dr, bend left, "\overline{t}"] & \\
\bullet & & \bullet
\end{tikzcd}$$

where $\overline{p} \leftrightarrow^* \overline{rs}$, $\overline{q} \leftrightarrow^* \overline{rt}$, and $N\left(\overline{s}\right) \cap N\left(\overline{t}\right) = \emptyset$.

We will prove that it is always possible to commute $\overline{p}$ and $\overline{q}$ into such a state.

Then the actual merging is done on $\overline{s}$ and $\overline{t}$.

## 8.2 Sequence-wise merge

Suppose we have two sequences $\overline{p}$ and $\overline{q}$, where $N\left(\overline{p}\right) \cap N\left(\overline{q}\right) = \emptyset$:

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

Our goal is to find $\overline{p'}$ and $\overline{q'}$ such that $\overline{pq'} \leftrightarrow^* \overline{qp'}$, and is "morally equivalent" to applying both $\overline{p}$ and $\overline{q}$.

In order to achieve this, we use the fact that all patches are invertible, along with the notion of patch commute that we already have. To start with, let's add the sequence $\overline{p^{-1}}$ to our diagram:

$$\begin{tikzcd}
 & \bullet \arrow[dl, bend right, "\overline{p}"'] \arrow[dr, bend left, "\overline{q}"] & \\
\bullet \arrow[ur, bend right, "\overline{p}^{-1}"'] & & \bullet
\end{tikzcd}$$

Now $\overline{p}^{-1}$ and $\overline{q}$ are in sequence, so we can commute them:

$$\begin{tikzcd}
 & \bullet \arrow[dl, bend right, "\overline{p}"'] \arrow[dr, bend left, "\overline{q}"] & \\
\bullet \arrow[ur, bend right, "\overline{p}^{-1}"'] \arrow[dr, bend right, "\overline{q'}"'] & & \bullet \\
 & \bullet \arrow[ur, bend left, "\overline{p'}^{-1}"] &
\end{tikzcd}$$

Invert $\overline{p'}^{-1}$ again and throw away the inverted sequences, and we have the merge result:

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

But does this satisfy the "moral equivalence" requirement?

Consider starting with the sequence $\overline{pp}^{-1}\overline{q}$, which has the effect of just $\overline{q}$. Then we commuted $\overline{p}^{-1}$ and $\overline{q}$ giving us $\overline{pq'p'}^{-1}$, which must still have the effect of just $\overline{q}$. The "moral equivalence" guaranteed by commute tells us that $\overline{p'}^{-1}$ must be "morally equivalent" to $\overline{p}^{-1}$, and thus $\overline{p'}^{-1}$ removes the "moral effect" of $\overline{p}$. Therefore, if we take our sequence $\overline{pq'p'}^{-1}$ and throw away $\overline{p'}^{-1}$, leaving just $\overline{pq'}$, we must have the "moral effects" of $\overline{p}$ and $\overline{q}$.

That paragraph was very hand-wavey, but that's because our notion of "moral equivalence" is very hand-wavey.

## 8.3 Patch-wise merge

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

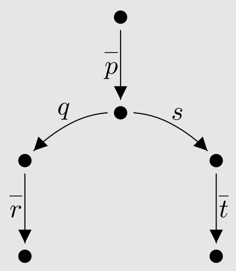
$$\begin{tikzcd}
& \bullet \arrow[d, "\overline{p}"'] & \\
& \bullet \arrow[dl, bend right, "q"'] \arrow[dr, bend left, "r"] & \\
\bullet & & \bullet
\end{tikzcd}$$

In order to merge the two sequences, we want a single sequence of patches incorporating the effects of both $q$ and $r$. We do this in the same way that we handled sequences in sequence-wise merge, by completing the bottom of our diagram to form a commutation square:

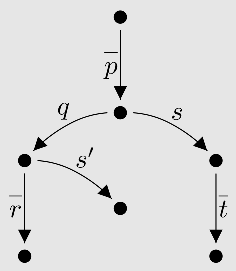
$$\begin{tikzcd}
& \bullet \arrow[d, "\overline{p}"'] & \\
& \bullet \arrow[dl, bend right, "q"'] \arrow[dr, bend left, "r"] & \\
\bullet \arrow[dr, bend right, "r'"'] & & \bullet \arrow[dl, bend left, "q'"] \\
& \bullet &
\end{tikzcd}$$

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

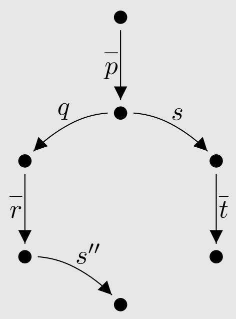
Now let's generalise to arbitrarily large merges. In one sequence we have $\overline{p}q\overline{r}$, and in the other we have $\overline{p}s\overline{t}$, where $N\left(q\overline{r}\right) \cap N\left(s\overline{t}\right) = \emptyset$:



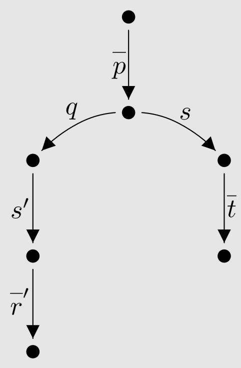
We start off by merging $q$ and $s$:



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



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



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

$$\begin{tikzcd}
& \bullet \arrow[d, "\overline{p}"'] & \\
& \bullet \arrow[dl, bend right, "s"'] \arrow[dr, bend left, "s"] & \\
\bullet \arrow[d, "q'"'] & & \bullet \arrow[d, "\overline{t}"] \\
\bullet \arrow[d, "\overline{r}'"'] & & \bullet \\
\bullet & &
\end{tikzcd}$$

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

$$\begin{tikzcd}
& \bullet \arrow[d, "\overline{p}"'] & \\
& \bullet \arrow[d, "s"'] & \\
& \bullet \arrow[dl, bend right, "q'"'] \arrow[dr, bend left, "\overline{t}"] & \\
\bullet \arrow[d, "\overline{r}'"'] & & \bullet \\
\bullet & &
\end{tikzcd}$$

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

$$\begin{tikzcd}
\bullet \arrow[d, "\overline{p}"'] \\
\bullet \arrow[d, "s"'] \\
\bullet \arrow[d, "\overline{t}"'] \\
\bullet \arrow[d, "q''"'] \\
\bullet \arrow[d, "\overline{r}''"'] \\
\bullet
\end{tikzcd}$$

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ patch_universes

## 9 Patch Universes

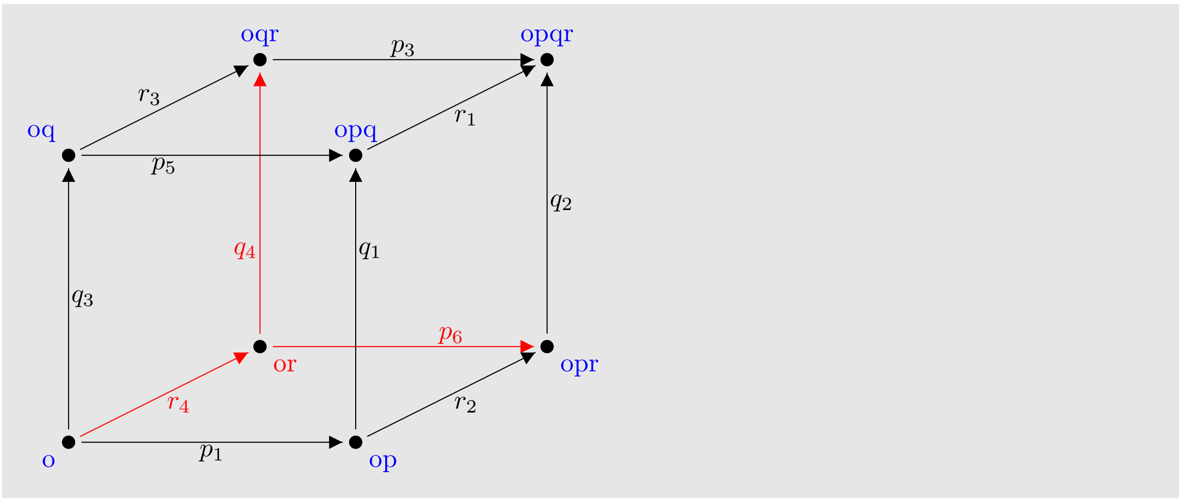
```
Module Export patch_universes.

Require Import Equality.
Require Import names.
Require Import util.
```

In this section we will introduce *patch universes*. Provided a patch type, and the operations on it, satisfies certain axioms, a set of sequences of patches forms a patch universe if it satisfies certain invariants.

In this section we will use notation normally used for named patches, but to refer to any patch type that can form a patch universe. Likewise, the term "patch" refers to the patch type of the particular patch universe being used. We use 𝕌 to denote the entire universe.

```
Definition patchNamesOK (from to : NameSet)
                          (sn : SignedName)
                        : Prop
    := match sn with
       | MkSignedName Positive n ⇒ (˜ NameSetIn n from)
                                    ∧ (NameSetEqual to (NameSetAdd n from))
       | MkSignedName Negative n ⇒ (˜ NameSetIn n to)
                                    ∧ (NameSetEqual from (NameSetAdd n to))
       end.

Reserved Notation "« p , q » <˜> « q' , p' »"
    (at level 60, no associativity).
Reserved Notation "p <˜?˜> q"
    (at level 60, no associativity).
Class PartPatchUniverse (pu_type1 pu_type2 : (NameSet → NameSet → Type))
                          : Type := mkPartPatchUniverse {
    commute : ∀ {from mid1 mid2 to : NameSet},
                  pu_type1 from mid1 → pu_type2 mid1 to
               → pu_type2 from mid2 → pu_type1 mid2 to → Prop
        where "« p , q » <˜> « q' , p' »" := (commute p q q' p');

    commutable : ∀ {from mid1 to : NameSet},
                          pu_type1 from mid1 → pu_type2 mid1 to → Prop
        := fun {from mid1 to : NameSet}
                (p : pu_type1 from mid1) (q : pu_type2 mid1 to) ⇒
             ∃ mid2 : NameSet,
             ∃ q' : pu_type2 from mid2,
             ∃ p' : pu_type1 mid2 to,
             «p, q» <˜> «q', p'»
        where "p <˜?˜> q" := (commutable p q);

    commutable_dec : ∀ {from mid to : NameSet}
                              (p : pu_type1 from mid)
                              (q : pu_type2 mid to),
                         {mid2 : NameSet &
                         { q' : pu_type2 from mid2 &
                         { p' : pu_type1 mid2 to &
```

```
                           «p, q» <~> «q’, p’» }}}
                         + {~(p <~?~> q)};

    commuteUnique : ∀ {from mid mid’ mid’’ to : NameSet}
                              {p : pu_type1 from mid} {q : pu_type2 mid to}
                              {q’ : pu_type2 from mid’} {p’ : pu_type1 mid’ to}
                              {q’’ : pu_type2 from mid’’} {p’’ : pu_type1 mid’’ to},
                          «p, q» <~> «q’, p’»
                        → «p, q» <~> «q’’, p’’»
                        → (mid’ = mid’’) ∧ (p’ ~= p’’) ∧ (q’ ~= q’’)
}.
Notation "« p , q » <~> « q’ , p’ »" := (commute p q q’ p’).
Notation "p <~?~> q" := (commutable p q).

Ltac doCommuteUnique H1 H2 :=
    let HmidsEqHmidsEq := fresh "HmidsEqHmidsEq" in
    let HpsEq := fresh "HpsEq" in
    let HqsEq := fresh "HqsEq" in
    destruct (commuteUnique H1 H2) as [HmidsEq [HpsEq HqsEq]];
    subst;
    subst.

Class PatchUniverseInv {pu_type1 pu_type2 : (NameSet → NameSet → Type)}
                       (ppu1 : PartPatchUniverse pu_type1 pu_type2)
                       (ppu2 : PartPatchUniverse pu_type2 pu_type1)
                     : Type := mkPatchUniverseInv {
    commuteSelfInverse : ∀ {from mid1 mid2 to : NameSet}
                                  {p : pu_type1 from mid1}
                                  {q : pu_type2 mid1 to}
                                  {q’ : pu_type2 from mid2}
                                  {p’ : pu_type1 mid2 to},
        («p, q» <~> «q’, p’»)
      → («q’, p’» <~> «p, q»)
}.

Class PatchUniverse {pu_type : (NameSet → NameSet → Type)}
                    {ppu : PartPatchUniverse pu_type pu_type}
                    (pui : PatchUniverseInv ppu ppu)
                  : Type := mkPatchUniverse {
    pu_nameOf : ∀ {from to : NameSet}, pu_type from to → SignedName;
```



```
commuteConsistent1 : ∀ {o op opq opqr opr oq oqr : NameSet}
                                       {p1 : pu_type o op}
                                       {q1 : pu_type op opq}
                                       {r1 : pu_type opq opqr}
                                       {q2 : pu_type opr opqr}
                                       {r2 : pu_type op opr}
                                       {q3 : pu_type o oq}
                                       {r3 : pu_type oq oqr}
                                       {p3 : pu_type oqr opqr}
                                       {p5 : pu_type oq opq},
                              «q1, r1» <~> «r2, q2»
                           → «p1, q1» <~> «q3, p5»
                           → «p5, r1» <~> «r3, p3»
                           → ∃ or : NameSet,
                              ∃ r4 : pu_type o or,
                              ∃ q4 : pu_type or oqr,
                              ∃ p6 : pu_type or opr,
                              «q3, r3» <~> «r4, q4» ∧
                              «p1, r2» <~> «r4, p6» ∧
                              «p6, q2» <~> «q4, p3»;

commuteConsistent2 : ∀ {o op opq opqr or oq oqr : NameSet}
                                       {q3 : pu_type o oq}
                                       {r3 : pu_type oq oqr}
                                       {p3 : pu_type oqr opqr}
                                       {q4 : pu_type or oqr}
                                       {r4 : pu_type o or}
                                       {p1 : pu_type o op}
                                       {q1 : pu_type op opq}
                                       {r1 : pu_type opq opqr}
                                       {p5 : pu_type oq opq},
                              «q3, r3» <~> «r4, q4»
                           → «r3, p3» <~> «p5, r1»
                           → «q3, p5» <~> «p1, q1»
                           → ∃ opr : NameSet,
                              ∃ r2 : pu_type op opr,
                              ∃ q2 : pu_type opr opqr,
                              ∃ p6 : pu_type or opr,
                              «q1, r1» <~> «r2, q2» ∧
                              «q4, p3» <~> «p6, q2» ∧
                              «r4, p6» <~> «p1, r2»;

(*
This isn't true for conflictors, but happily turned out not to be used
in the proofs so far:
    pu_contexts : forall {from to : NameSet} (p : pu_type from to),
                  patchNamesOK from to (pu_nameOf p);
*)
```

We will write sequences of patches as $\overline{p}$, and we require that patches $p$ have names $n(p)$. The set of names in a sequence of patches, or in a set of sequences of patches, is $N(p)$; the set of names used in the whole universe is therefore $N(\mathbb{U})$. The commute relations $\leftrightarrow$, $\underset{?}{\leftrightarrow}$ and $\leftrightarrow^*$ work on patches and sequences of patches in the normal way.

```
    commuteNames : ∀ {from mid1 mid2 to : NameSet}
                         {p : pu_type from mid1} {q : pu_type mid1 to}
                         {q' : pu_type from mid2} {p' : pu_type mid2 to},
                   « p , q » <~> « q' , p' »
                 → (pu_nameOf p = pu_nameOf p') ∧
                   (pu_nameOf q = pu_nameOf q') ∧
                   (pu_nameOf p ≠ pu_nameOf q)
}.
(*
Lemma commuteOKOfInverse : forall {o op op' opq o' oq oq' oqp : NameSet}
                                  (commuteOK : CommuteOK),
                           CommuteOK o' oq oq' oqp o op op' opq.
Proof.
intros.
constructor.
            remember commuteFromOK.
            clear - e.
            nameSetDec.
        remember commuteMid2OK.
        clear - e.
        nameSetDec.
    remember commuteMid1OK.
    clear - e.
    nameSetDec.
remember commuteToOK.
clear - e.
nameSetDec.
Qed.
*)
Inductive SequenceBase {pu_type : NameSet → NameSet → Type}
                       {ppu : PartPatchUniverse pu_type pu_type}
```

```
                              {pui : PatchUniverseInv ppu ppu}
                              (patchUniverse : PatchUniverse pui)
                            : NameSet → NameSet → Type
    := Nil : ∀ {cxt : NameSet},
                  SequenceBase patchUniverse cxt cxt
     | Cons : ∀ {from mid to : NameSet}
                            (p : pu_type from mid)
                            (qs : SequenceBase patchUniverse mid to),
                   SequenceBase patchUniverse from to.
Implicit Arguments Nil [pu_type ppu pui patchUniverse cxt].
Implicit Arguments Cons [pu_type ppu pui patchUniverse from mid to].
Fixpoint sequenceBaseContents {pu_type : NameSet → NameSet → Type}
                                       {ppu : PartPatchUniverse pu_type pu_type}
                                       {pui : PatchUniverseInv ppu ppu}
                                       {pu : PatchUniverse pui}
                                       {from to : NameSet}
                                       (s : SequenceBase pu from to)
                                     : SignedNameSet
    := match s with
       | Nil _ ⇒ SignedNameSetMod.empty
       | Cons _ _ _ p qs ⇒ SignedNameSetMod.add (pu_nameOf p) (sequenceBaseContents qs)
       end.
Fixpoint SequenceNoDupes {pu_type : NameSet → NameSet → Type}
                                  {ppu : PartPatchUniverse pu_type pu_type}
                                  {pui : PatchUniverseInv ppu ppu}
                                  {pu : PatchUniverse pui}
                                  {from to : NameSet}
                                  (s : SequenceBase pu from to)
                                : Prop
    := match s with
       | Nil _ ⇒ True
       | Cons _ _ _ p qs ⇒ ¬SignedNameSetMod.In (pu_nameOf p) (sequenceBaseContents qs)
                                  ∧ SequenceNoDupes qs
       end.
Lemma SequenceNoDupesCons
    {pu_type : NameSet → NameSet → Type}
    {ppu : PartPatchUniverse pu_type pu_type}
    {pui : PatchUniverseInv ppu ppu}
    {pu : PatchUniverse pui}
    {from mid to : NameSet}
    {x : pu_type from mid}
    {ys : SequenceBase pu mid to}
  : SequenceNoDupes (Cons x ys) = ¬ SignedNameSetIn (pu_nameOf x) (sequenceBaseContents
ys)
                                                  ∧ SequenceNoDupes ys.
Proof.
auto.
Qed.
Class SequenceOK {pu_type : NameSet → NameSet → Type}
                         {ppu : PartPatchUniverse pu_type pu_type}
                         {pui : PatchUniverseInv ppu ppu}
```

```
                    {pu : PatchUniverse pui}
                    {from to : NameSet}
                    (s : SequenceBase pu from to)
    := {
            sequenceNoDupes : SequenceNoDupes s
       }.
Inductive Sequence {pu_type : NameSet → NameSet → Type}
                   {ppu : PartPatchUniverse pu_type pu_type}
                   {pui : PatchUniverseInv ppu ppu}
                   (pu : PatchUniverse pui)
                   (from to : NameSet)
                 : Type
    := MkSequence : ∀ (s : SequenceBase pu from to)
                        {sequenceOK : SequenceOK s},
                   Sequence pu from to.
Implicit Arguments MkSequence [pu_type ppu pui pu from to].
Definition sequenceContents {pu_type : NameSet → NameSet → Type}
                            {ppu : PartPatchUniverse pu_type pu_type}
                            {pui : PatchUniverseInv ppu ppu}
                            {pu : PatchUniverse pui}
                            {from to : NameSet}
                            (s : Sequence pu from to)
                          : SignedNameSet
    := match s with
       | MkSequence s _ ⇒ sequenceBaseContents s
       end.
Class ConsOK {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {from mid to : NameSet}
             (head : pu_type from mid)
             (tail : Sequence pu mid to)
    := {
            consNotAlreadyThere : ¬ SignedNameSetIn (pu_nameOf head) (sequenceContents
tail)
       }.
Implicit Arguments ConsOK [pu_type ppu pui pu from mid to].
Program Definition nilSeq {pu_type : NameSet → NameSet → Type}
                          {ppu : PartPatchUniverse pu_type pu_type}
                          {pui : PatchUniverseInv ppu ppu}
                          {pu : PatchUniverse pui}
                          (cxt : NameSet)
                        : Sequence pu cxt cxt
    := MkSequence Nil _.
Next Obligation.
constructor.
constructor.
Qed.
Program Definition consSeq {pu_type : NameSet → NameSet → Type}
                           {ppu : PartPatchUniverse pu_type pu_type}
```

```
                                {pui : PatchUniverseInv ppu ppu}
                                {pu : PatchUniverse pui}
                                {from mid to : NameSet}
                                (p : pu_type from mid)
                                (qs : Sequence pu mid to)
                                (consOK : ConsOK p qs)
                              : Sequence pu from to
    := match qs with
        | MkSequence s _ ⇒
              MkSequence (Cons p s) _
        end.
Next Obligation.
constructor.
constructor.
    destruct consOK.
    unfold sequenceContents in consNotAlreadyThere0.
    auto.
destruct wildcard'.
auto.
Qed.

Notation "s :> t" := (consSeq s t _)
    (at level 60, right associativity).
Notation "[]" := (nilSeq _)
    (no associativity).

Lemma NilSeqToNil {pu_type : NameSet → NameSet → Type}
                  {ppu : PartPatchUniverse pu_type pu_type}
                  {pui : PatchUniverseInv ppu ppu}
                  {pu : PatchUniverse pui}
                  {cxt : NameSet}
                  (nilSequenceOK : SequenceOK Nil)
                : MkSequence Nil nilSequenceOK = nilSeq cxt.
Proof with auto.
unfold nilSeq.
f_equal.
apply proof_irrelevance.
Qed.

Lemma ConsSeqToCons {pu_type : NameSet → NameSet → Type}
                    {ppu : PartPatchUniverse pu_type pu_type}
                    {pui : PatchUniverseInv ppu ppu}
                    {pu : PatchUniverse pui}
                    {from mid to : NameSet}
                    (p : pu_type from mid)
                    (ps : SequenceBase pu mid to)
                    {pPsSequenceOK : SequenceOK (Cons p ps)}
                  : { psSequenceOK : SequenceOK ps &
                    { consOK : ConsOK p (MkSequence ps psSequenceOK) &
                    MkSequence (Cons p ps) pPsSequenceOK = p :> (MkSequence ps psSe-
quenceOK)}}.
Proof with auto.
solveExists.
    constructor.
    destruct pPsSequenceOK as [pPsSequenceNoDupes].
    destruct pPsSequenceNoDupes...
solveExists.
    constructor.
    destruct pPsSequenceOK as [pPsSequenceNoDupes].
    destruct pPsSequenceNoDupes.
    unfold sequenceContents...
unfold consSeq.
f_equal.
apply proof_irrelevance.
Qed.

Ltac consSeqToCons p ps :=
    let psSequenceOK := fresh "psSequenceOK" in
    let pPsConsOK := fresh "pPsConsOK" in
    let pPsEquality := fresh "pPsEquality" in
    destruct (ConsSeqToCons p ps) as [psSequenceOK [pPsConsOK pPsEquality]];
    rewrite pPsEquality in *;
    clear pPsEquality.

Lemma ConsEqNil {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {patchUniverse : PatchUniverse pui}
                {cxt mid : NameSet}
                {p : pu_type cxt mid}
                {ps : Sequence patchUniverse mid cxt}
                {consOK : ConsOK p ps}
                (consEqNil : (p :> ps) = [])
              : False.
Proof.
destruct ps.
unfold consSeq in consEqNil.
inversion consEqNil.
Qed.

Program Fixpoint appendBase
                {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {patchUniverse : PatchUniverse pui}
                {from mid to : NameSet}
                (ps : SequenceBase patchUniverse from mid)
                (qs : SequenceBase patchUniverse mid to)
        : SequenceBase patchUniverse from to
    := match ps with
        | Nil _ ⇒
            qs
        | Cons _ midP _ p ps' ⇒
            Cons p (appendBase (from := midP) (to := to) ps' qs)
        end.

Lemma SequenceContentsBaseNil {pu_type : NameSet → NameSet → Type}
                              {ppu : PartPatchUniverse pu_type pu_type}
                              {pui : PatchUniverseInv ppu ppu}
```

```
                                        {patchUniverse : PatchUniverse pui}
                                        {cxt : NameSet}
                                      : sequenceBaseContents (Nil (cxt := cxt))
                                      = SignedNameSetMod.empty.
Proof with auto.
unfold sequenceBaseContents...
Qed.
Lemma SequenceContentsBaseCons {pu_type : NameSet → NameSet → Type}
                                 {ppu : PartPatchUniverse pu_type pu_type}
                                 {pui : PatchUniverseInv ppu ppu}
                                 {patchUniverse : PatchUniverse pui}
                                 {from mid to : NameSet}
                                 (p : pu_type from mid)
                                 (ps : SequenceBase patchUniverse mid to)
                               : sequenceBaseContents (Cons p ps)
                               = SignedNameSetMod.add (pu_nameOf p) (sequenceBaseCon-
tents ps).
Proof with auto.
unfold sequenceBaseContents at 1.
fold (sequenceBaseContents (from := mid) (to := to))...
Qed.
Lemma AppendBaseCons {pu_type : NameSet → NameSet → Type}
                      {ppu : PartPatchUniverse pu_type pu_type}
                      {pui : PatchUniverseInv ppu ppu}
                      {patchUniverse : PatchUniverse pui}
                      {from mid1 mid2 to : NameSet}
                      (p : pu_type from mid1)
                      (ps : SequenceBase patchUniverse mid1 mid2)
                      (qs : SequenceBase patchUniverse mid2 to)
                    : appendBase (Cons p ps) qs
                    = Cons p (appendBase ps qs).
Proof with auto.
unfold appendBase at 1.
fold (appendBase (from := mid1) (mid := mid2) (to := to))...
Qed.
Lemma AppendBaseNil {pu_type : NameSet → NameSet → Type}
                     {ppu : PartPatchUniverse pu_type pu_type}
                     {pui : PatchUniverseInv ppu ppu}
                     {patchUniverse : PatchUniverse pui}
                     {from to : NameSet}
                     (ps : SequenceBase patchUniverse from to)
                   : appendBase Nil ps
                   = ps.
Proof with auto.
simpl...
Qed.
Lemma AppendBaseNil2 {pu_type : NameSet → NameSet → Type}
                      {ppu : PartPatchUniverse pu_type pu_type}
                      {pui : PatchUniverseInv ppu ppu}
                      {patchUniverse : PatchUniverse pui}
                      {from to : NameSet}
```

```
                         (ps : SequenceBase patchUniverse from to)
                       : appendBase ps Nil
                       = ps.
Proof with auto.
induction ps...
simpl.
rewrite IHps...
Qed.

Lemma SequenceContentsBaseAppend {pu_type : NameSet → NameSet → Type}
                                     {ppu : PartPatchUniverse pu_type pu_type}
                                     {pui : PatchUniverseInv ppu ppu}
                                     {patchUniverse : PatchUniverse pui}
                                     {from mid to : NameSet}
                                     (ps : SequenceBase patchUniverse from mid)
                                     (qs : SequenceBase patchUniverse mid to)
                                   : sequenceBaseContents (appendBase ps qs)
                                   = SignedNameSetMod.union (sequenceBaseContents ps) (se-
quenceBaseContents qs).
Proof with auto.
induction ps.
    rewrite SequenceContentsBaseNil.
    rewrite AppendBaseNil.
    apply SignedNameSetEquality.
    signedNameSetDec.
specialize (IHps qs).
rewrite AppendBaseCons.
rewrite SequenceContentsBaseCons.
rewrite SequenceContentsBaseCons.
rewrite IHps.
apply SignedNameSetEquality.
signedNameSetDec.
Qed.

Lemma SequenceContentsConsSeq
            {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {patchUniverse : PatchUniverse pui}
            {from mid to : NameSet}
            {p : pu_type from mid}
            {qs : SequenceBase patchUniverse mid to}
            {sequenceOK : SequenceOK (Cons p qs)}
          : sequenceContents (MkSequence (Cons p qs) sequenceOK) = SignedNameSetMod.add
(pu_nameOf p) (sequenceBaseContents qs).
Proof with auto.
unfold sequenceContents.
unfold sequenceBaseContents.
fold (sequenceBaseContents (from := mid) (to := to))...
Qed.

Lemma SequenceContentsNil
            {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
```

```
             {pui : PatchUniverseInv ppu ppu}
             {patchUniverse : PatchUniverse pui}
             {cxt : NameSet}
           : sequenceContents (nilSeq cxt) = SignedNameSetMod.empty.
Proof with auto.
unfold nilSeq.
unfold sequenceContents.
unfold sequenceBaseContents...
Qed.

Lemma SequenceContentsCons
             {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {patchUniverse : PatchUniverse pui}
             {from mid to : NameSet}
             {p : pu_type from mid}
             {qs : Sequence patchUniverse mid to}
             {consOK : ConsOK p qs}
           : sequenceContents (p :> qs) = SignedNameSetMod.add (pu_nameOf p) (sequenceCon-
tents qs).
Proof with auto.
unfold consSeq.
destruct qs.
apply SequenceContentsConsSeq.
Qed.

Class AppendOK {pu_type : NameSet → NameSet → Type}
               {ppu : PartPatchUniverse pu_type pu_type}
               {pui : PatchUniverseInv ppu ppu}
               {patchUniverse : PatchUniverse pui}
               {from mid to : NameSet}
               (ps : Sequence patchUniverse from mid)
               (qs : Sequence patchUniverse mid to)
     := {
           appendNoIntersection : SignedNameSetMod.Empty
                                       (SignedNameSetMod.inter (sequenceContents ps) (se-
quenceContents qs))
        }.

Program Definition append
                 {pu_type : NameSet → NameSet → Type}
                 {ppu : PartPatchUniverse pu_type pu_type}
                 {pui : PatchUniverseInv ppu ppu}
                 {patchUniverse : PatchUniverse pui}
                 {from mid to : NameSet}
                 (ps : Sequence patchUniverse from mid)
                 (qs : Sequence patchUniverse mid to)
                 (appendOK : AppendOK ps qs)
        : Sequence patchUniverse from to
     := match ps with
        | MkSequence psBase psOK ⇒
              match qs with
              | MkSequence qsBase qsOK ⇒
```

```
                    MkSequence (appendBase psBase qsBase) _
              end
        end.
Next Obligation.
induction psBase.
    rewrite AppendBaseNil.
    auto.
destruct psOK as [psNoDupes].
destruct psNoDupes.
solveFirstIn IHpsBase.
    constructor.
    auto.
specialize (IHpsBase qsBase).
specialize (IHpsBase qsOK).
destruct appendOK as [appendNoIntersection].
unfold sequenceContents.
unfold sequenceContents in appendNoIntersection.
solveFirstIn IHpsBase.
    clear IHpsBase.
    rewrite SequenceContentsBaseCons in appendNoIntersection.
    constructor.
    unfold sequenceContents.
    (*  coq bug 2699 *)
    remember (sequenceBaseContents psBase) as ps.
    remember (sequenceBaseContents qsBase) as qs.
    signedNameSetDec.
rewrite AppendBaseCons.
constructor.
constructor.
    rewrite SequenceContentsBaseCons in appendNoIntersection.
    rewrite SequenceContentsBaseAppend.
    (*  coq bug 2699 *)
    remember (sequenceBaseContents psBase) as ps.
    remember (sequenceBaseContents qsBase) as qs.
    signedNameSetDec.
destruct IHpsBase.
auto.
Qed.

Notation "ps :+> qs" := (append ps qs _)
    (at level 60, right associativity).

Lemma SequenceContentsAppend {pu_type : NameSet → NameSet → Type}
                              {ppu : PartPatchUniverse pu_type pu_type}
                              {pui : PatchUniverseInv ppu ppu}
                              {patchUniverse : PatchUniverse pui}
                              {from mid to : NameSet}
                              (ps : Sequence patchUniverse from mid)
                              (qs : Sequence patchUniverse mid to)
                              {appendOK : AppendOK ps qs}
                            : sequenceContents (ps :+> qs)
                            = SignedNameSetMod.union (sequenceContents ps) (sequenceCon-
tents qs).
```

```
Proof with auto.
destruct ps.
destruct qs.
unfold sequenceContents.
unfold append.
apply SequenceContentsBaseAppend.
Qed.

Lemma AppendNilSeq {pu_type : NameSet → NameSet → Type}
                   {ppu : PartPatchUniverse pu_type pu_type}
                   {pui : PatchUniverseInv ppu ppu}
                   {patchUniverse : PatchUniverse pui}
                   {from to : NameSet}
                   (ps : Sequence patchUniverse from to)
                   {sequenceOK : SequenceOK Nil}
                   {appendOK : AppendOK (MkSequence Nil sequenceOK) ps}
                 : MkSequence Nil sequenceOK :+> ps = ps.
Proof with auto.
destruct ps.
unfold append.
f_equal.
apply proof_irrelevance.
Qed.

Lemma AppendNil {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {patchUniverse : PatchUniverse pui}
                {from to : NameSet}
                (ps : Sequence patchUniverse from to)
                {appendOK : AppendOK [] ps}
              : [] :+> ps = ps.
Proof with auto.
unfold nilSeq.
apply AppendNilSeq.
Qed.

Lemma AppendConsSeq {pu_type : NameSet → NameSet → Type}
                    {ppu : PartPatchUniverse pu_type pu_type}
                    {pui : PatchUniverseInv ppu ppu}
                    {patchUniverse : PatchUniverse pui}
                    {from mid1 mid2 to : NameSet}
                    (p : pu_type from mid1)
                    (ps : SequenceBase patchUniverse mid1 mid2)
                    (qs : Sequence patchUniverse mid2 to)
                    {ppsOK : SequenceOK (Cons p ps)}
                    {append1OK : AppendOK (MkSequence (Cons p ps) ppsOK) qs}
                  : ∃ psOK : SequenceOK ps,
                    ∃ append2OK : AppendOK (MkSequence ps psOK) qs,
                    ∃ cons1OK : ConsOK p ((MkSequence ps psOK) :+> qs),
                        (MkSequence (Cons p ps) ppsOK :+> qs)
                      = p :> ((MkSequence ps psOK) :+> qs).
Proof with auto.
destruct ppsOK as [ppsNoDupes].
destruct append1OK as [append1NoIntersection].
solveExists.
    unfold SequenceNoDupes in ppsNoDupes.
    fold (SequenceNoDupes (from := mid1) (to := mid2)) in ppsNoDupes.
    constructor.
    destruct ppsNoDupes...
solveExists.
    constructor.
    rewrite SequenceContentsConsSeq in append1NoIntersection.
    destruct qs.
    unfold sequenceContents.
    unfold sequenceContents in append1NoIntersection.
    (*  coq bug 2699 *)
    remember (sequenceBaseContents ps) as psContents.
    remember (sequenceBaseContents s) as sContents.
    signedNameSetDec.
solveExists.
    constructor.
    destruct qs.
    rewrite SequenceContentsConsSeq in append1NoIntersection.
    destruct ppsNoDupes.
    unfold sequenceContents.
    unfold append.
    rewrite SequenceContentsBaseAppend.
    unfold sequenceContents in append1NoIntersection.
    (*  coq bug 2699 *)
    remember (sequenceBaseContents ps) as psContents.
    remember (sequenceBaseContents s) as sContents.
    signedNameSetDec.
destruct qs.
simpl.
f_equal.
apply proof_irrelevance.
Qed.

Lemma AppendCons {pu_type : NameSet → NameSet → Type}
                 {ppu : PartPatchUniverse pu_type pu_type}
                 {pui : PatchUniverseInv ppu ppu}
                 {patchUniverse : PatchUniverse pui}
                 {from mid1 mid2 to : NameSet}
                 (p : pu_type from mid1)
                 (ps : Sequence patchUniverse mid1 mid2)
                 (qs : Sequence patchUniverse mid2 to)
                 {cons1OK : ConsOK p ps}
                 {append1OK : AppendOK (p :> ps) qs}
               : ∃ append2OK : AppendOK ps qs,
                 ∃ cons2OK : ConsOK p (ps :+> qs),
                     (p :> ps) :+> qs
                   = p :> ps :+> qs.
Proof with auto.
destruct append1OK as [append1NoIntersection].
solveExists.
    constructor.
    rewrite SequenceContentsCons in append1NoIntersection.
    (*  coq bug 2699 *)
    remember (sequenceContents ps) as psContents.
    remember (sequenceContents qs) as qsContents.
    signedNameSetDec.
solveExists.
    constructor.
    rewrite SequenceContentsCons in append1NoIntersection.
    destruct cons1OK.
    rewrite SequenceContentsAppend.
    (*  coq bug 2699 *)
    remember (sequenceContents ps) as psContents.
    remember (sequenceContents qs) as qsContents.
    signedNameSetDec.
destruct ps.
destruct qs.
simpl.
f_equal.
apply proof_irrelevance.
Qed.
```

There is also a merge relation $\oplus$. Given two patches $p$ and $q$, either there exists $p'$ and $q'$ such that $N(p) = N(p')$, $N(q) = N(q')$, $p \oplus q = \langle q', p' \rangle$ and $pq' \leftrightarrow^* qp'$; or $p \oplus q = \text{fail}$.

**Axiom 9.1 (patch-universe-commute-preserves-commute)**
$(\langle p, qr\rangle \leftrightarrow \langle q'r', p'\rangle) \Rightarrow \left(\left(q \underset{?}{\leftrightarrow} r\right) \Leftrightarrow \left(q' \underset{?}{\leftrightarrow} r'\right)\right)$

```
  (*
      commute : pu_type -> pu_type -> pu_type -> pu_type -> Prop
          where "« p , q » <~> « q' , p' »" := (commute p q q' p');
  *)
```

**Explanation**
*This is Lemma ?? restated for patch universes.*

**Axiom 9.2 (patch-universe-commute-consistent)**
$(\langle q, r\rangle \leftrightarrow \langle r', q'\rangle) \Rightarrow ((\langle p, qr\rangle \leftrightarrow \langle \_, p'\rangle) \Leftrightarrow (\langle p, r'q'\rangle \leftrightarrow \langle \_, p'\rangle))$

**Explanation**
*This is Lemma ?? restated for named patches.*

Those are the operations and axioms that your patch type must provide. We now explain what can be done in a patch universe, and what invariants must be satisfied by the operations while they are done.

The operations you can do in a patch universe are:

**Record** Given a sequence of patches $\overline{p}$ we can record a patch with a fresh name, giving us $\overline{pq}$.

**Unrecord** Given a sequence of patches $\overline{pq}$ we can unpull $q$ leaving us $\overline{p}$.

**Commute** Given $\overline{p}$, we can make $\overline{q}$ where $\overline{p} \leftrightarrow^* \overline{q}$.

**Merge** Given $\overline{pq}$ and $\overline{pr}$, if $\overline{q} \oplus \overline{r} = \left\langle \overline{r'}, \overline{q'} \right\rangle$ then we can make $\overline{pqr'}$ and $\overline{prq'}$.

XXX and unrecord

XXX The remainder of this section is proofs of things

**Lemma 9.1 (commute-in-sequence-consistent)**
Suppose $\overline{p}qr\overline{s} \leftrightarrow^* \overline{t}q'r'\overline{u}$ where $n(q) = n(q')$ and $n(r) = n(r')$. Then $\left(q \underset{?}{\leftrightarrow} r\right) \Leftrightarrow \left(q' \underset{?}{\leftrightarrow} r'\right)$.

**Explanation**
*Suppose we have adjacent two patches $q$ and $r$ in a sequence. No amount of commuting within this sequence can alter whether or not $q$ and $r$ commute.*

**Proof**
Suppose, for the purpose of contradiction, that we have a counter example of minimal sequence length. Without loss of generality, assume that $q \underset{?}{\leftrightarrow} r$ holds but $q' \underset{?}{\leftrightarrow} r'$ does not.

If $\overline{p} = \epsilon$ and $\overline{s} = \epsilon$ then we trivially have a contradiction. We will assume that $\overline{p}$ is non-empty, but an analogous argument covers the case where only $\overline{s}$ is non-empty.

So $\overline{p} = v\overline{p'}$, for some patch $v$ and sequence $\overline{p'}$. We will first show, by induction on the structure of $\leftrightarrow^*$, that at each step $v$ can be commuted directly to the start of the sequence.

Trivially that is true for the initial sequence $v\overline{p'}qr\overline{s}$. Suppose that it is true for a sequence $\overline{a_1}a_2a_3\overline{a_4}$ where $\langle a_2, a_3\rangle \leftrightarrow \langle a'_3, a'_2\rangle$; then we must show that it is true for $\overline{a_1}a'_3a'_2\overline{a_4}$. If $v$ is in $\overline{a_1}$ then it is trivially true, using the same sequence of commutes as before. If $v$ is $a_2$ then it is true, by first commuting with $a'_3$ and then continuing as before. If $v$ is $a_3$ then it is true, by skipping the first commute of the previous sequence. The must interesting case is when $v$ is in $\overline{a_4}$. We know that $v$ can be commuted past everything else in $\overline{a_4}$ until we have $\overline{a_1}a_2a_3v'\overline{a'_4}$, then past $a_2a_3$ giving us $\overline{a_1}v''a''_2a''_3\overline{a'_4}$, and finally past $\overline{a_1}$, giving us $v'''\overline{a'_1}a''_2a''_3\overline{a'_4}$. But Axiom 9.2 tells us that $\langle a'_3a'_2, v'\rangle \leftrightarrow \langle v'', a'''_3a'''_2\rangle$, so it is also true for this case.

Now, consider the final sequence. If $v$ is in $\overline{t}$ then trivially whether $q'$ and $r'$ commute is unchanged. If $v$ is in $\overline{u}$ then Axiom 9.1 tells us that whether $q'$ and $r'$ commute is unchanged.

Therefore we can construct a shorter counter example, by commuting $v$ to the left of the sequence at each stage, and removing it. ■

```
(*
XXX
Lemma NamePersistsInSequenceNameSet :
      forall {patchUniverse : PatchUniverse}
             {o op opq opqr : NameSet}
             (p : (pu_type patchUniverse) o op)
             (qs : Sequence (pu_type patchUniverse) op opq)
             (rs : Sequence (pu_type patchUniverse) opq opqr),
      NameSetIn (pu_nameOf patchUniverse p) opq
   -> NameSetIn (pu_nameOf patchUniverse p) opqr.
Proof with auto.
intros.
dependent induction rs...
specialize (IHrs patchUniverse).
specialize (IHrs p).
specialize (IHrs mid).
specialize (IHrs (qs :+> (p0 :> ))).
assert (p_in_mid : NameSetIn (pu_nameOf patchUniverse p) mid).
```

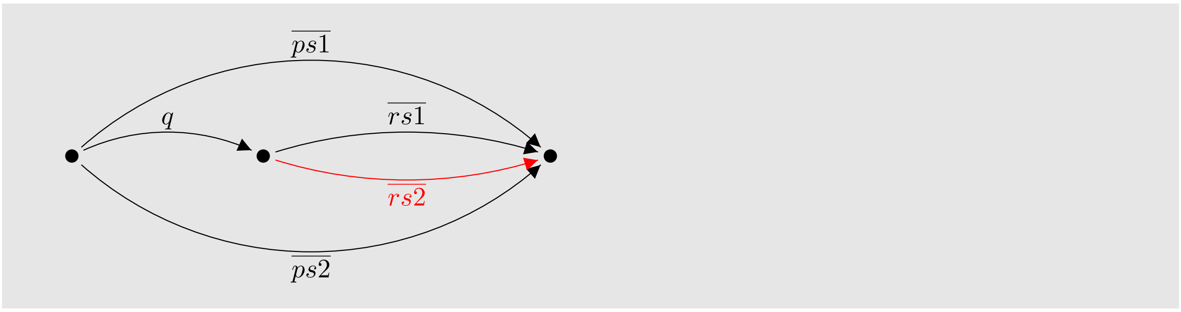
```
    destruct (pu_contexts patchUniverse p0).
    fsetdec.
specialize (IHrs p_in_mid).
specialize (IHrs opqr refl refl)...
Qed.

Lemma NamePersistsInSequence :
      forall {patchUniverse : PatchUniverse}
             {o op opq : NameSet}
             (p : (pu_type patchUniverse) o op)
             (qs : Sequence (pu_type patchUniverse) op opq),
      NameSetIn (pu_nameOf patchUniverse p) opq.
Proof with auto.
intros.
assert (p_in_op : NameSetIn (pu_nameOf patchUniverse p) op).
    destruct (pu_contexts patchUniverse p).
    fsetdec.
set (H1 := NamePersistsInSequenceNameSet p  qs p_in_op)...
Qed.

Lemma NameInSequenceUnique :
      forall {patchUniverse : PatchUniverse}
             {o op opq opqr : NameSet}
             (p : (pu_type patchUniverse) o op)
             (qs : Sequence (pu_type patchUniverse) op opq)
             (r : (pu_type patchUniverse) opq opqr),
      (pu_nameOf patchUniverse p <> pu_nameOf patchUniverse r).
Proof with auto.
intros.
set (p_in_opq := NamePersistsInSequence p qs).
clearbody p_in_opq.
destruct (pu_contexts patchUniverse r).
congruence.
Qed.
*)
Reserved Notation "« ps » <~~> « qs »"
    (at level 60, no associativity).
Inductive CommuteRelates {pu_type : NameSet → NameSet → Type}
                         {ppu : PartPatchUniverse pu_type pu_type}
                         {pui : PatchUniverseInv ppu ppu}
                         {patchUniverse : PatchUniverse pui}
                         {from to : NameSet}
                       : Sequence patchUniverse from to
                      → Sequence patchUniverse from to
                      → Prop
    := SwapNow : ∀ {op opq oq : NameSet}
                        {p : pu_type from op}
                        {q : pu_type op opq}
                        {q' : pu_type from oq}
                        {p' : pu_type oq opq}
                        {rs : Sequence patchUniverse opq to}
                        (qRsConsOK : ConsOK q rs)
```

```
                              (qQRsConsOK : ConsOK p (q :> rs))
                              (pRsConsOK : ConsOK p' rs)
                              (qPRsConsOK : ConsOK q' (p' :> rs)),
                       «p, q» <~> «q', p'»
                    → «p :> q :> rs» <~~> «q' :> p' :> rs»
      | SwapLater : ∀ {mid : NameSet}
                                (p : pu_type from mid)
                                {qs : Sequence patchUniverse mid to}
                                {rs : Sequence patchUniverse mid to}
                                (pQsOK : ConsOK p qs)
                                (pRsOK : ConsOK p rs),
                         CommuteRelates (from := mid) qs rs
                      → CommuteRelates (p :> qs)
                                       (p :> rs)
where "« ps » <~~> « qs »"
    := (@CommuteRelates _ _ _ _ _ _ _ ps qs).

(*
Lemma CommuteRelatesSameContentsBase
          {pu_type : NameSet -> NameSet -> Type}
          {ppu : PartPatchUniverse pu_type pu_type}
          {pui : PatchUniverseInv ppu ppu}
          {patchUniverse : PatchUniverse pui}
          {from to : NameSet}
          {s t : SequenceBase patchUniverse from to}
          {sSequenceOK : SequenceOK s}
          {tSequenceOK : SequenceOK t}
          (commuteRelates : « MkSequence s sSequenceOK » <~~> « MkSequence t tSequenceOK »)
        : sequenceBaseContents s = sequenceBaseContents t.
Proof.
dependent destruction commuteRelates.
*)

Lemma CommuteRelatesSameContents {pu_type : NameSet → NameSet → Type}
                                 {ppu : PartPatchUniverse pu_type pu_type}
                                 {pui : PatchUniverseInv ppu ppu}
                                 {patchUniverse : PatchUniverse pui}
                                 {from to : NameSet}
                                 {s t : Sequence patchUniverse from to}
                                 (commuteRelates : « s » <~~> « t »)
                               : sequenceContents s = sequenceContents t.
Proof with auto.
induction commuteRelates as [ ? ? ? ? ? ? ? ? ? ? ? ? ? ? commute | ? ? ? ? ? ? ? ? ?
qs_rs_same_contents ].
    rewrite SequenceContentsCons.
    rewrite SequenceContentsCons.
    rewrite SequenceContentsCons.
    rewrite SequenceContentsCons.
    destruct (commuteNames commute) as [? [? ?]].
    apply SignedNameSetEquality.
    signedNameSetDec.
rewrite SequenceContentsCons.
rewrite SequenceContentsCons.
rewrite qs_rs_same_contents...
Qed.

Reserved Notation "« p » <~~>* « q »"
    (at level 60, no associativity).
(*  XXX Would this be better with 3 constructors, Same, Swap and Trans? *)
Inductive TransitiveCommuteRelates {pu_type : NameSet → NameSet → Type}
                                  {ppu : PartPatchUniverse pu_type pu_type}
                                  {pui : PatchUniverseInv ppu ppu}
                                  {patchUniverse : PatchUniverse pui}
                                  {from to : NameSet}
                        : Sequence patchUniverse from to
                        → Sequence patchUniverse from to
                        → Prop
    := Same : ∀ (s : Sequence patchUniverse from to),
                TransitiveCommuteRelates s s
     | Swap : ∀ {s : Sequence patchUniverse from to}
                       {t : Sequence patchUniverse from to}
                       {u : Sequence patchUniverse from to},
                «s» <~~> «t»
             → «t» <~~>* «u»
             → «s» <~~>* «u»
where "« p » <~~>* « q »"
    := (@TransitiveCommuteRelates _ _ _ _ _ _ p q).

Lemma TransitiveCommuteRelatesSameContents
          {pu_type : NameSet → NameSet → Type}
          {ppu : PartPatchUniverse pu_type pu_type}
          {pui : PatchUniverseInv ppu ppu}
          {patchUniverse : PatchUniverse pui}
          {from to : NameSet}
          {s t : Sequence patchUniverse from to}
          (transitiveCommuteRelates : « s » <~~>* « t »)
        : sequenceContents s = sequenceContents t.
Proof with auto.
induction transitiveCommuteRelates as [? | ? ? ? s_commute_t ? t_u_same_contents]...
rewrite ← t_u_same_contents.
rewrite (CommuteRelatesSameContents s_commute_t)...
Qed.

Reserved Notation "« p , qs » <~ « rs »"
    (at level 60, no associativity).
Inductive commuteOutLeft {pu_type : NameSet → NameSet → Type}
                         {ppu : PartPatchUniverse pu_type pu_type}
                         {pui : PatchUniverseInv ppu ppu}
                         {patchUniverse : PatchUniverse pui}
                         {from mid to : NameSet}
                        : Sequence patchUniverse from to
                        → pu_type from mid
                        → Sequence patchUniverse mid to
                        → Prop
    := commuteOutLeftDone : ∀ (p : pu_type from mid)
                                  (qs : Sequence patchUniverse mid to)
                                  (p_qsConsOK : ConsOK p qs),
                                «p, qs» <~ «p :> qs»
      | commuteOutLeftSwap : ∀ {mid_q mid_r : NameSet}
                                        {p : pu_type mid_r mid_q}
                                        {qs : Sequence patchUniverse mid_q to}
                                        {rs : Sequence patchUniverse mid_r to}
                                        {p' : pu_type from mid}
                                        {q : pu_type mid mid_q}
                                        {r : pu_type from mid_r}
                                        (q_qsConsOK : ConsOK q qs)
                                        (r_rsConsOK : ConsOK r rs),
                                «p, qs» <~ «rs»
                              → «r, p» <~> «p', q»
                              → «p', q :> qs» <~ «r :> rs»
where "« p , qs » <~ « rs »"
    := (@commuteOutLeft _ _ _ _ _ _ _ _ rs p qs).

(*
Inductive commuteOutRight {patchUniverse : PatchUniverse}
                          {from mid to : NameSet} :
                          Sequence patchUniverse from to
                       -> Sequence patchUniverse from mid
                       -> (pu_type patchUniverse) mid to
                       -> Prop
    := commuteOutRightDone : forall (ps : Sequence patchUniverse from mid)
                                    (q : (pu_type patchUniverse) mid to),
                             commuteOutRight (ps :+> q :> ) ps q
     | commuteOutRightSwap : forall {beforeq mid' : NameSet}
                                    {ps : Sequence patchUniverse from beforeq}
                                    {q : (pu_type patchUniverse) beforeq to}
                                    {rs : Sequence patchUniverse from mid'}
                                    {s : (pu_type patchUniverse) mid' beforeq}
                                    {q' : (pu_type patchUniverse) mid' mid}
                                    {s' : (pu_type patchUniverse) mid to},
                             @commuteOutRight patchUniverse from mid' beforeq ps rs s
                          -> «s, q» <~> «q', s'»
                          -> commuteOutRight (ps :+> q :> )
                                             (rs :+> q' :> )
                                             s'.
Notation "« rs » ~> « ps , q »"
    := (commuteOutRight rs ps q)
    (at level 60, no associativity).
*)

Lemma TransitiveTransitiveCommute :
    ∀ {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {patchUniverse : PatchUniverse pui}
            {from to : NameSet}
            {ps qs rs : Sequence patchUniverse from to}
            (ps_qs : «ps» <~~>* «qs»)
            (qs_rs : «qs» <~~>* «rs»),
    («ps» <~~>* «rs»).
```

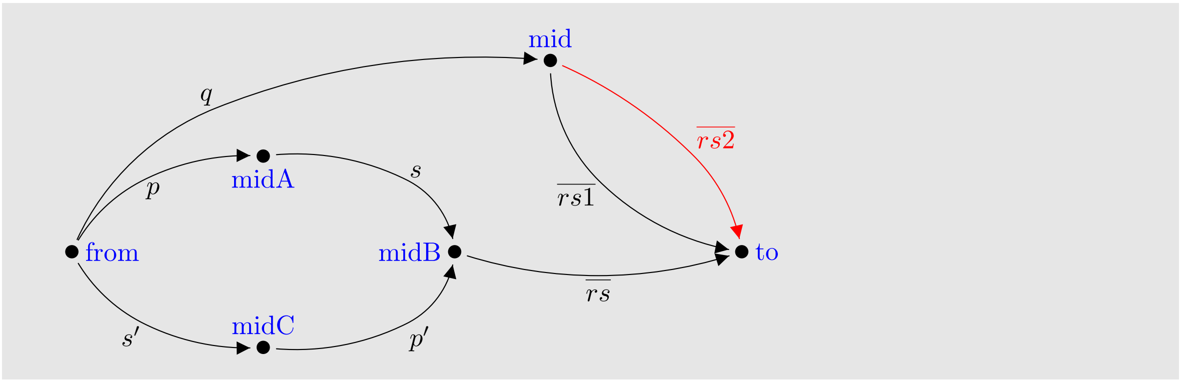
```
Proof with auto.
intros.
induction ps_qs as [? | ? ? ? s_commute_t ? IH]...
apply IH in qs_rs.
apply (Swap s_commute_t qs_rs).
Qed.

Lemma SymmetricCommute :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {from to : NameSet}
           {ps qs : Sequence patchUniverse from to}
           (ps_qs : «ps» <~~> «qs»),
    («qs» <~~> «ps»).
Proof with auto.
intros.
induction ps_qs as [? ? ? ? ? ? ? ? ? ? ? ? ? ? ? commute | ?].
    apply commuteSelfInverse in commute.
    apply (SwapNow _ _ _ _ commute).
apply SwapLater...
Qed.

Lemma SymmetricTransitiveCommute :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {from to : NameSet}
           {ps qs : Sequence patchUniverse from to}
           (ps_qs : «ps» <~~>* «qs»),
    («qs» <~~>* «ps»).
Proof with auto.
intros.
induction ps_qs as [? | ? ? ? s_commute_t ? IH].
    apply Same.
apply SymmetricCommute in s_commute_t.
apply (TransitiveTransitiveCommute IH (Swap s_commute_t (Same _))).
Qed.

Lemma EnlargeTransitiveCommuteRelation :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {from mid to : NameSet}
           {p : pu_type from mid}
           {qs rs : Sequence patchUniverse mid to}
           (pQsOK : ConsOK p qs)
           (pRsOK : ConsOK p rs)
           (qs_rs : «qs» <~~>* «rs»),
    (« (p :> qs) » <~~>* « (p :> rs) »).
Proof with auto.
```

```
intros.
induction qs_rs as [? | ? ? ? s_commute_t ? IH].
    proofIrrel pQsOK pRsOK.
    apply Same.
assert (ptOK : ConsOK p t).
    constructor.
    destruct pQsOK.
    rewrite ← (CommuteRelatesSameContents s_commute_t)...
apply (Swap (SwapLater p _ _ s_commute_t) (IH ptOK pRsOK)).
Qed.

Lemma ConsEquality1
          {pu_type : NameSet → NameSet → Type}
          {ppu : PartPatchUniverse pu_type pu_type}
          {pui : PatchUniverseInv ppu ppu}
          {pu : PatchUniverse pui}
          {from mid1 mid2 to : NameSet}
          {p : pu_type from mid1}
          {q : pu_type from mid2}
          {ps : Sequence pu mid1 to}
          {qs : Sequence pu mid2 to}
          {pConsOK : ConsOK p ps}
          {qConsOK : ConsOK q qs}
          (eq : p :> ps = q :> qs)
        : mid1 = mid2.
Proof with auto.
destruct ps.
destruct qs.
unfold consSeq in eq.
dependent destruction eq...
Qed.

Lemma ConsEquality2
          {pu_type : NameSet → NameSet → Type}
          {ppu : PartPatchUniverse pu_type pu_type}
          {pui : PatchUniverseInv ppu ppu}
          {pu : PatchUniverse pui}
          {from mid to : NameSet}
          {p : pu_type from mid}
          {q : pu_type from mid}
          {ps : Sequence pu mid to}
          {qs : Sequence pu mid to}
          {pConsOK : ConsOK p ps}
          {qConsOK : ConsOK q qs}
          (eq : p :> ps = q :> qs)
        : p = q ∧ ps = qs.
Proof with auto.
destruct ps as [? psSequenceOK].
destruct qs as [? qsSequenceOK].
unfold consSeq in eq.
dependent destruction eq.
proofIrrel psSequenceOK qsSequenceOK.
split...
```

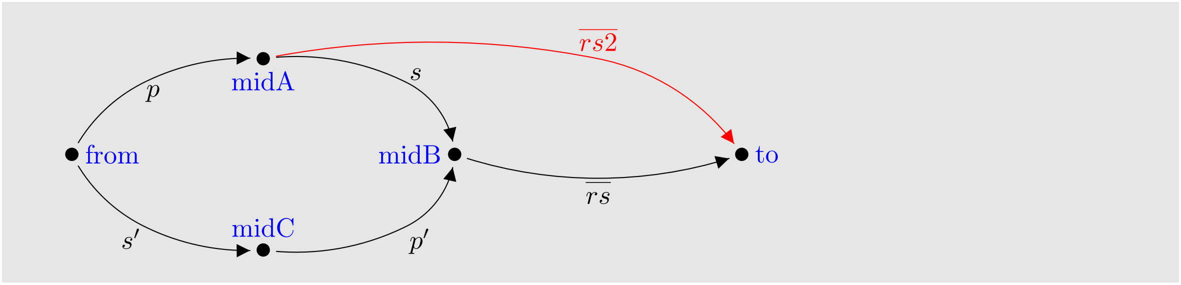
```
Qed.

Lemma ConsEquality3
          {pu_type : NameSet → NameSet → Type}
          {ppu : PartPatchUniverse pu_type pu_type}
          {pui : PatchUniverseInv ppu ppu}
          {pu : PatchUniverse pui}
          {from mid to : NameSet}
          {p : pu_type from mid}
          {ps : Sequence pu mid to}
          {pConsOK : ConsOK p ps}
          {qConsOK : ConsOK p ps}
          (eq : consSeq p ps pConsOK = consSeq p ps qConsOK)
        : pConsOK = qConsOK.
Proof with auto.
apply proof_irrelevance...
Qed.

Ltac consEquality hyp :=
    let hyp' := fresh "H" in
    let H1 := fresh "H" in
    let H2 := fresh "H" in
    let H3 := fresh "H" in
    rename hyp into hyp';
    set (H1 := ConsEquality1 hyp');
    clearbody H1;
    subst;
    destruct (ConsEquality2 hyp') as [H2 hyp];
    try (rewrite hyp in *);
    subst;
    try (set (H3 := ConsEquality3 hyp');
         clearbody H3;
         subst);
    clear hyp'.

Lemma commuteOutLeftConsistentStep :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {from mid to : NameSet}
           {ps1 ps2 : Sequence patchUniverse from to}
           {q : pu_type from mid}
           {rs1 : Sequence patchUniverse mid to}
           (commute_relates_ps1_ps2 : «ps1» <~~> «ps2»)
           (q_rs1_commutes_left_out_of_ps1 : «q, rs1» <~ «ps1»),
    (∃ rs2 : Sequence patchUniverse mid to,
     («rs1» <~~>* «rs2») ∧
     («q, rs2» <~ «ps2»)).
Proof with auto.
intros.
```



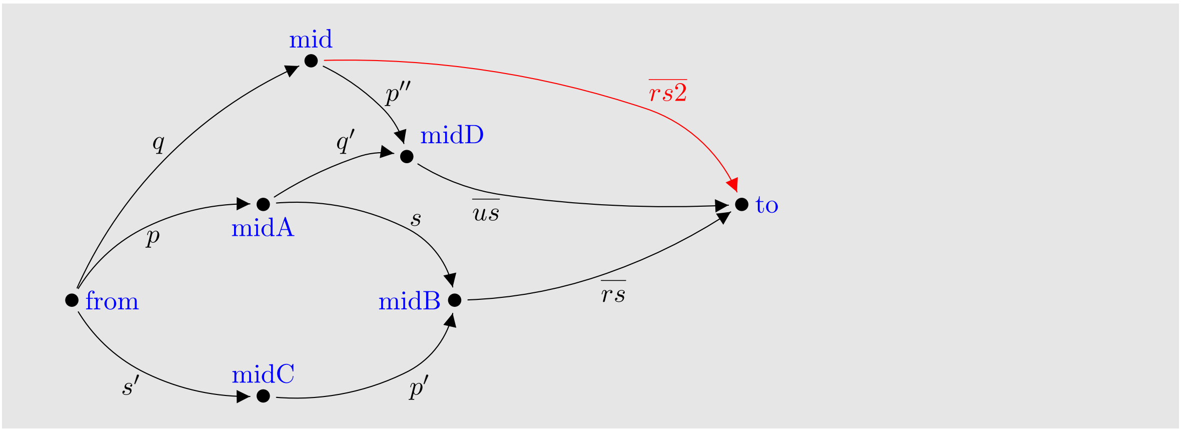
```
(*  XXX disambig down to here *)
dependent induction commute_relates_ps1_ps2
  generalizing mid q rs1 q_rs1_commutes_left_out_of_ps1.
    (*  This is the case where we have a swap at the head *)
    rename op into midA, opq into midB, oq into midC.
    rename q0 into s, q' into s'.
    rename H into p_s_commute_s'_p'.
```



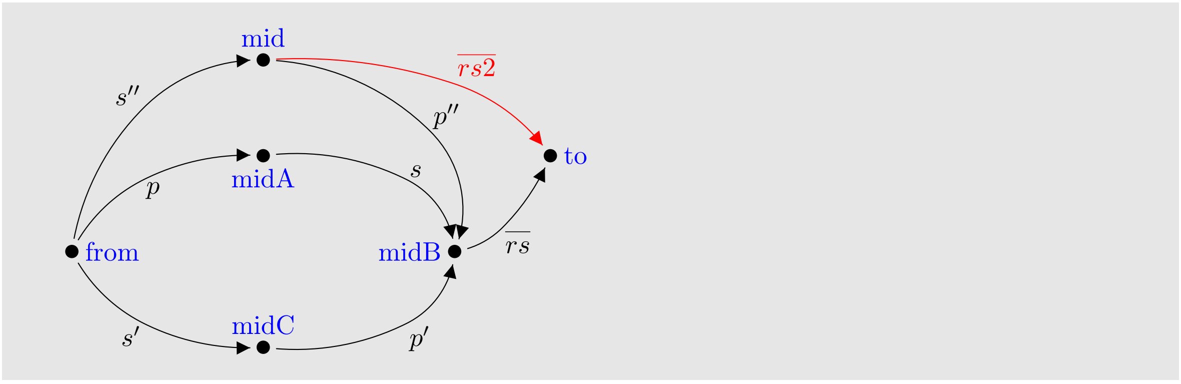
```
    dependent destruction q_rs1_commutes_left_out_of_ps1.
        (*  This is the case where the commute out is done.
            But the swap has messed it up, so we will have to
            undo it to build the commute out. *)
```



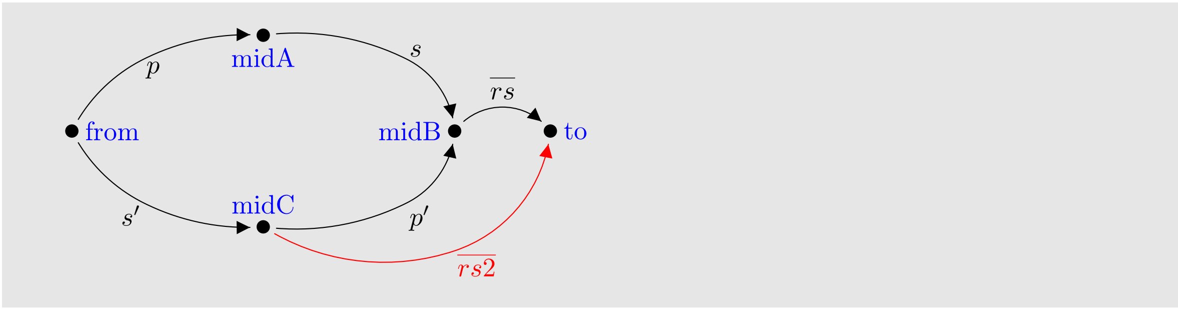
```
        consEquality x.
        ∃ (q :> rs).
        split.
        apply Same.
    apply commuteSelfInverse in p_s_commute_s'_p'.
    refine (commuteOutLeftSwap _ _ _ p_s_commute_s'_p')...
    apply commuteOutLeftDone.
(*  This is the case where the commuteout actually does some work *)
rename mid_q into midD.
rename q into p'', p0 into q', p'0 into q, qs into us.
rename H into p_q'_commute_q_p''.
```



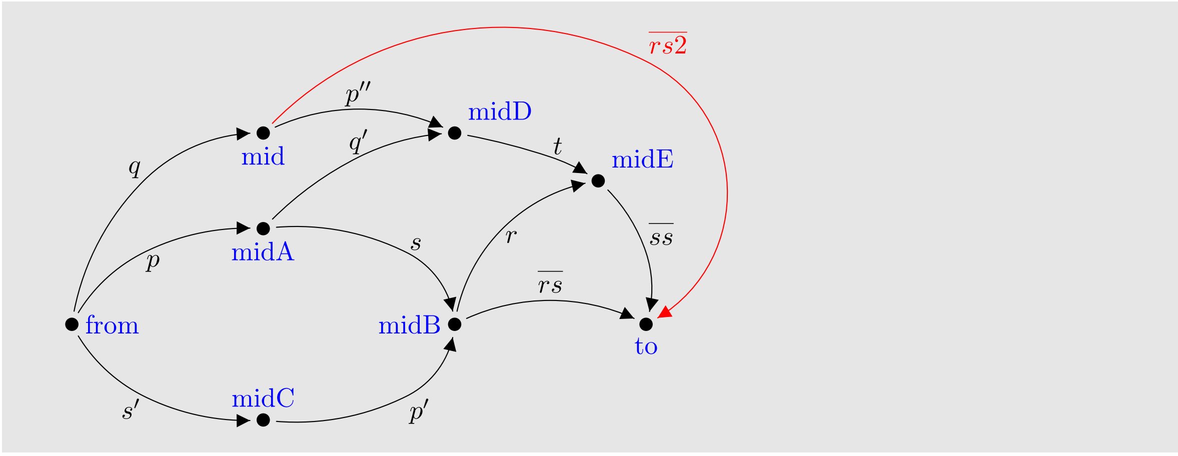
```
(*  There are two possibilities:
   * The commuteout does exactly one commute, and the swap has
     just done it for us
   * The commuteout does more than one swap, and so we need to
     use the lemma to rewrite the last 2 swaps *)
dependent destruction q_rs1_commutes_left_out_of_ps1.
    (*  This is the case where the commuteout does
       exactly one commute, so we now have nothing to do *)
    consEquality x.
    consEquality x.
    rename q into s''.
    rename p_q'_commute_q_p'' into p_s_commute_s''_p''.
```



```
doCommuteUnique p_s_commute_s'_p' p_s_commute_s''_p''.
rename p'' into p', s'' into s'.
rename mid into midC.
```



```
assert (Heq : q_qsConsOK = pRsConsOK).
    apply proof_irrelevance.
subst.
∃ (q0 :> rs).
split.
    apply Same.
apply commuteOutLeftDone.
(*  This is the case where the commuteout does
   at least commute2, so we have to jump over the swap *)
consEquality x.
consEquality x.
rename mid_q into midE.
rename p'0 into q'.
rename p0 into r.
rename q0 into t.
rename qs into ss.
rename H into s_r_commute_q'_t.
```



```
    assert (H := commuteConsistent2
                     p_s_commute_s'_p'
                     s_r_commute_q'_t
                     p_q'_commute_q_p'').
    destruct H as [? [t' [p''' [r'
                    [p''_t_commute_t'_p'''
                     [p'_r_commute_r'_p'''
                      s'_r'_commute_q_t']]]]]].
    destruct (commuteNames p''_t_commute_t'_p''')
          as [name_p''_name_p'''
             [name_t_name_t'
              name_p_not_name_t]].
    assert (p'''_ssConsOK : ConsOK p''' ss).
        constructor.
        destruct q_qsConsOK as [H].
        rewrite SequenceContentsCons in H.
        rewrite ← name_p''_name_p'''.
        (*  coq bug 2699 *)
        remember (sequenceContents ss) as ssContents.
        signedNameSetDec.
    assert (t'_p'''_ssConsOK : ConsOK t' (p''' :> ss)).
        constructor.
        destruct q_qsConsOK0 as [H].
        rewrite SequenceContentsCons.
        rewrite ← name_p''_name_p'''.
        rewrite ← name_t_name_t'.
        (*  coq bug 2699 *)
        (*  clear - name_p_not_name_t H. *)
        (*  signedNameSetDec. *)
        apply (coqBug 2690).
    ∃ (t' :> p''' :> ss).
    split.
        apply (Swap (SwapNow _ _ _ _ p''_t_commute_t'_p''') (Same _)).
    refine (commuteOutLeftSwap _ _ _ s'_r'_commute_q_t').
    refine (commuteOutLeftSwap _ _ _ p'_r_commute_r'_p''')...
(*  This is the case where we have a swap later *)
dependent destruction q_rs1_commutes_left_out_of_ps1.
    (*  This is the case where the commute out left is done.
        If the swap doesn't happen until later, then the commute
        out left is still done in the other sequence. *)
    consEquality x.
    ∃ rs.
    split.
        apply (Swap commute_relates_ps1_ps2 (Same _)).
    apply commuteOutLeftDone.
(*  And this is the real inductive case, which follows trivially
   by induction. *)
consEquality x.
specialize (IHcommute_relates_ps1_ps2 mid_q).
specialize (IHcommute_relates_ps1_ps2 p0).
specialize (IHcommute_relates_ps1_ps2 qs0).
specialize (IHcommute_relates_ps1_ps2 q_rs1_commutes_left_out_of_ps1).
destruct IHcommute_relates_ps1_ps2 as [? [? ?]].
assert (consOK : ConsOK q x).
    constructor.
    destruct q_qsConsOK.
    rewrite ← (TransitiveCommuteRelatesSameContents H0)...
∃ (q :> x).
split.
    apply (EnlargeTransitiveCommuteRelation _ _ H0).
apply (commuteOutLeftSwap _ _ H1 H).
Qed.

Lemma commuteOutLeftConsistent :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {from mid to : NameSet}
           {ps1 ps2 : Sequence patchUniverse from to}
           {q : pu_type from mid}
           {rs1 : Sequence patchUniverse mid to}
           (ps1_ps2 : «ps1» <~~>* «ps2»)
           (p_out_of_ps1 : «q, rs1» <~ «ps1»),
    (∃ rs2 : Sequence patchUniverse mid to,
     («rs1» <~~>* «rs2») ∧
     («q, rs2» <~ «ps2»)).
Proof with auto.
intros.
dependent induction ps1_ps2 generalizing rs1 p_out_of_ps1.
    (*  ps1 = ps2 case *)
    ∃ rs1.
    split...
    apply Same.
(*  case where there is at least one commute to be done *)
set (Hstep := commuteOutLeftConsistentStep H p_out_of_ps1).
destruct Hstep as [rs2 [rs1_rs2 q_out_of_t]].
specialize (IHps1_ps2 rs2 q_out_of_t).
destruct IHps1_ps2 as [rs3 [rs2_rs3 q_out_of_ps2]].
∃ rs3.
split...
apply (TransitiveTransitiveCommute rs1_rs2 rs2_rs3).
Qed.
```

(*

$\overline{p}qr\overline{s} \leftrightarrow^{*} \overline{t}q'r'\overline{u}$

where $n(q) = n(q')$ and $n(r) = n(r')$.

Then $\left(q \underset{?}{\leftrightarrow} r\right) \Leftrightarrow \left(q' \underset{?}{\leftrightarrow} r'\right)$.

*)

```
Lemma CommuteOutLeftPreservesCommute :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
```

```
              {patchUniverse : PatchUniverse pui}
              {o op opq opqr opqrs ot : NameSet}
              {ps : Sequence patchUniverse o op}
              {q : pu_type op opq}
              {r : pu_type opq opqr}
              {ss : Sequence patchUniverse opqr opqrs}
              {t : pu_type o ot}
              {us : Sequence patchUniverse ot opqrs}

              (r_ssConsOK : ConsOK r ss)
              (q_r_ssConsOK : ConsOK q (r :> ss))
              (ps_q_r_ssAppendOK : AppendOK ps (q :> r :> ss))

              (t_q_different_names : pu_nameOf t ≠ pu_nameOf q)
              (t_r_different_names : pu_nameOf t ≠ pu_nameOf r)
              (commuteOut : «t, us» <~ « ps :+> (q :> r :> ss) »),
      (∃ otp : NameSet,
       ∃ otpq : NameSet,
       ∃ otpqr : NameSet,
       ∃ ps' : Sequence patchUniverse ot otp,
       ∃ q' : pu_type otp otpq,
       ∃ r' : pu_type otpq otpqr,
       ∃ ss' : Sequence patchUniverse otpqr opqrs,
       ∃ r'_ss'ConsOK : ConsOK r' ss',
       ∃ q'_r'_ss'ConsOK : ConsOK q' (r' :> ss'),
       ∃ ps'_q'_r'_ss'AppendOK : AppendOK ps' (q' :> (r' :> ss')),
            (pu_nameOf q = pu_nameOf q') ∧
            (pu_nameOf r = pu_nameOf r') ∧
            (us = ps' :+> (q' :> (r' :> ss'))) ∧
            (~(q <~?~> r) → ~(q' <~?~> r'))).
Proof with auto.
intros.
destruct ps as [ps psOK].
revert opq opqr opqrs ot q r ss t us r_ssConsOK q_r_ssConsOK ps_q_r_ssAppendOK
t_q_different_names t_r_different_names commuteOut.
dependent induction ps.
    intros.
    rewrite AppendNilSeq in commuteOut.
    dependent destruction commuteOut.
        consEquality x.
        congruence.
    dependent destruction commuteOut.
        consEquality x.
        consEquality x.
        destruct (commuteNames H) as [? [? ?]].
        congruence.
    ∃ ot.
    ∃ mid_q.
    ∃ mid_q0.
    ∃ [].
    ∃ q0.
    ∃ q1.
```

```
        ∃ qs.
        ∃ q_qsConsOK0.
        ∃ q_qsConsOK.
        solveExists.
            constructor.
            unfold sequenceContents.
            signedNameSetDec.
        consEquality x.
        consEquality x.
        split.
            destruct (commuteNames H0) as [? [? ?]]...
        split.
            destruct (commuteNames H) as [? [? ?]]...
        split.
            rewrite AppendNil.
            f_equal.
        intro.
        intro.
        elim H1.
        destruct H2 as [? [? [? ?]]].
        apply commuteSelfInverse in H.
        apply commuteSelfInverse in H0.
        set (commuteConsistent := commuteConsistent1 H2 H0 H).
        destruct commuteConsistent as [? [? [? [? [? [? ?]]]]]].
        unfold commutable.
        ∃ x2.
        ∃ x3.
        ∃ x4...
    intros.
    destruct (AppendConsSeq p ps (q :> r :> ss)) as [? [? [? ?]]].
    rewrite H in commuteOut.
    move commuteOut at top.
    dependent destruction commuteOut.
        consEquality x.
        ∃ to.
        ∃ opq.
        ∃ opqr.
        ∃ (MkSequence ps x2).
        ∃ q.
        ∃ r.
        ∃ ss.
        ∃ r_ssConsOK.
        ∃ q_r_ssConsOK.
        ∃ x0.
        split...
    rename IHps into IH.
    consEquality x.
    specialize (IH x2).
    specialize (IH opq).
    specialize (IH opqr).
    specialize (IH opqrs).
    specialize (IH mid_q).
```

```
specialize (IH q).
specialize (IH r).
specialize (IH ss).
specialize (IH p0).
specialize (IH qs).
specialize (IH r_ssConsOK).
specialize (IH q_r_ssConsOK).
specialize (IH x0).
destruct (commuteNames H) as [? [name_v_eq_name_v' ?]].
rewrite name_v_eq_name_v' in *.
specialize (IH t_q_different_names).
specialize (IH t_r_different_names).
specialize (IH commuteOut).
destruct IH as [otp [otpq [otpqr
                [ps' [q' [r' [ss'
                [r'_ss'ConsOK [q'_r'_ss'ConsOK [ps'_q'_r'_ss'AppendOK
                [?
                [? [ws_equals_ps'_q'_r'_ss'
                 ?
                ]]]]]]]]]]]]].
∃ otp.
∃ otpq.
∃ otpqr.
subst.
assert (q0_ps'ConsOK : ConsOK q0 ps').
    constructor.
    destruct q_qsConsOK.
    rewrite SequenceContentsAppend in consNotAlreadyThere0.
    (*  coq bug 2699 *)
    remember (pu_nameOf q0) as q0Name.
    remember (sequenceContents ps') as ps'Contents.
    remember (sequenceContents (q' :> r' :> ss')) as otherContents.
    signedNameSetDec.
∃ (q0 :> ps').
∃ q'.
∃ r'.
∃ ss'.
∃ r'_ss'ConsOK.
∃ q'_r'_ss'ConsOK.
solveExists.
    clear - ps'_q'_r'_ss'AppendOK q_qsConsOK.
    constructor.
    destruct ps'_q'_r'_ss'AppendOK.
    destruct q_qsConsOK.
    rewrite SequenceContentsCons.
    rewrite SequenceContentsCons.
    rewrite SequenceContentsCons.
    rewrite SequenceContentsAppend in consNotAlreadyThere0.
    rewrite SequenceContentsCons in consNotAlreadyThere0.
    rewrite SequenceContentsCons in consNotAlreadyThere0.
    rewrite SequenceContentsCons in appendNoIntersection0.
    rewrite SequenceContentsCons in appendNoIntersection0.
    (*  coq bug 2699 *)
    remember (pu_nameOf q0) as q0Name.
    remember (pu_nameOf q') as q'Name.
    remember (pu_nameOf r') as r'Name.
    remember (sequenceContents ps') as ps'Contents.
    remember (sequenceContents ss') as ss'Contents.
    remember (SignedNameSetAdd q'Name (SignedNameSetAdd r'Name ss'Contents)) as
q'r'ss'.
    (*  signedNameSetDec. *)
    apply (coqBug 2690).
split...
split...
split...
destruct (AppendCons q0 ps' (q' :> r' :> ss')) as [? [? ?]].
rewrite H6.
proofIrrel x ps'_q'_r'_ss'AppendOK.
proofIrrel x3 q_qsConsOK...
Qed.

Lemma ShorterCounterExampleExists :
    ∀ {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {patchUniverse : PatchUniverse pui}
            {o op opq opqr opqrs opqrst ou our ours : NameSet}
            {p : pu_type o op}
            {qs : Sequence patchUniverse op opq}
            {r : pu_type opq opqr}
            {s : pu_type opqr opqrs}
            {ts : Sequence patchUniverse opqrs opqrst}
            {us : Sequence patchUniverse o ou}
            {r' : pu_type ou our}
            {s' : pu_type our ours}
            {vs : Sequence patchUniverse ours opqrst}

            (s_tsConsOK : ConsOK s ts)
            (r_s_tsConsOK : ConsOK r (s :> ts))
            (qs_r_s_tsAppendOK : AppendOK qs (r :> s :> ts))
            (p_qs_r_s_tsConsOK : ConsOK p (qs :+> r :> s :> ts))

            (s'_vsConsOK : ConsOK s' vs)
            (r'_s'_vsConsOK : ConsOK r' (s' :> vs))
            (us_r'_s'_vsAppendOK : AppendOK us (r' :> s' :> vs))

            (H : « p :> qs :+> r :> s :> ts » <~~>* « us :+> r' :> s' :> vs »)
            (rs_same : pu_nameOf r = pu_nameOf r')
            (ss_same : pu_nameOf s = pu_nameOf s')
            (r_s_commutable : r <~?~> s)
            (r'_s'_not_commutable : ~(r' <~?~> s')),
    (∃ ou' : NameSet,
     ∃ our' : NameSet,
     ∃ ours' : NameSet,
     ∃ us' : Sequence patchUniverse op ou',
     ∃ r'' : pu_type ou' our',
     ∃ s'' : pu_type our' ours',
     ∃ vs' : Sequence patchUniverse ours' opqrst,
     ∃ s''_vs'ConsOK : ConsOK s'' vs',
     ∃ r''_s''_vs'ConsOK : ConsOK r'' (s'' :> vs'),
     ∃ us'_r''_s''_vs'AppendOK : AppendOK us' (r'' :> s'' :> vs'),
         « qs :+> r :> s :> ts » <~~>* « us' :+> r'' :> s'' :> vs' » ∧
         (pu_nameOf r = pu_nameOf r'') ∧
         (pu_nameOf s = pu_nameOf s'') ∧
         ~(r'' <~?~> s'')).
Proof with auto.
intros.
assert (p_out_of_lhs: «p, qs :+> r :> s :> ts» <~ «p :> qs :+> r :> s :> ts»).
    apply commuteOutLeftDone.
set (p_out_of_rhs := commuteOutLeftConsistent H p_out_of_lhs).
destruct p_out_of_rhs as [? [transitive_commute p_out_of_rhs]].
assert (p_not_r : pu_nameOf p ≠ pu_nameOf r).
    clear - p_qs_r_s_tsConsOK.
    destruct p_qs_r_s_tsConsOK.
    rewrite SequenceContentsAppend in consNotAlreadyThere0.
    rewrite SequenceContentsCons in consNotAlreadyThere0.
    rewrite SequenceContentsCons in consNotAlreadyThere0.
    (*  coq bug 2699 *)
    remember (sequenceContents qs) as qsContents.
    remember (sequenceContents ts) as tsContents.
    remember (pu_nameOf p) as pName.
    remember (pu_nameOf r) as rName.
    remember (pu_nameOf s) as sName.
    (*  signedNameSetDec. *)
    apply (coqBug 2690).
assert (p_not_r' : pu_nameOf p ≠ pu_nameOf r').
    rewrite ← rs_same...
assert (p_not_s : pu_nameOf p ≠ pu_nameOf s).
    clear - p_qs_r_s_tsConsOK.
    destruct p_qs_r_s_tsConsOK.
    rewrite SequenceContentsAppend in consNotAlreadyThere0.
    rewrite SequenceContentsCons in consNotAlreadyThere0.
    rewrite SequenceContentsCons in consNotAlreadyThere0.
    (*  coq bug 2699 *)
    remember (sequenceContents qs) as qsContents.
    remember (sequenceContents ts) as tsContents.
    remember (pu_nameOf p) as pName.
    remember (pu_nameOf r) as rName.
    remember (pu_nameOf s) as sName.
    (*  signedNameSetDec. *)
    apply (coqBug 2690).
assert (p_not_s' : pu_nameOf p ≠ pu_nameOf s').
    rewrite ← ss_same...
set (HX := CommuteOutLeftPreservesCommute
                _ _ _ p_not_r' p_not_s' p_out_of_rhs).
destruct HX as [opp [oppq [oppqr
               [ps'' [q'' [r'' [ss''
```

```
                    [r''_ss''ConsOK [q''_r''_ss''ConsOK
                    [ps''_q''_r''_ss''AppendOK
                    [? [? [? ?]]]
]]] ]]
]]]]].
∃ opp.
∃ oppq.
∃ oppqr.
∃ ps''.
∃ q''.
∃ r''.
∃ ss''.
∃ r''_ss''ConsOK.
∃ q''_r''_ss''ConsOK.
∃ ps''_q''_r''_ss''AppendOK.
subst.
split...
split.
rewrite rs_same...
rewrite ss_same...
Qed.

Lemma EmptyCounterExampleExists :
    ∀ {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {patchUniverse : PatchUniverse pui}
            {o oq oqr oqrs ou our ours : NameSet}
            {qs : Sequence patchUniverse o oq}
            {r : pu_type oq oqr}
            {s : pu_type oqr oqrs}
            {us : Sequence patchUniverse o ou}
            {r' : pu_type ou our}
            {s' : pu_type our ours}
            {vs : Sequence patchUniverse ours oqrs}

            (s_NilConsOK : ConsOK s [])
            (r_s_tsConsOK : ConsOK r (s :> []))
            (qs_r_s_NilAppendOK : AppendOK qs (r :> s :> []))

            (s'_vsConsOK : ConsOK s' vs)
            (r'_s'_vsConsOK : ConsOK r' (s' :> vs))
            (us_r'_s'_vsAppendOK : AppendOK us (r' :> s' :> vs))

            (*  t omitted for now *)
            (transitive_commute : «qs :+> r :> s :> []» <~~>* «us :+> r' :> s' :> vs»)
            (rs_same : pu_nameOf r = pu_nameOf r')
            (ss_same : pu_nameOf s = pu_nameOf s')
            (r_s_commutable : r <~?~> s)
            (r'_s'_not_commutable : ~(r' <~?~> s')),
    (
     ∃ lNil_r_s_NilAppendOK : AppendOK [] (r :> s :> []),
     ∃ ou' : NameSet,
     ∃ our' : NameSet,
     ∃ ours' : NameSet,
     ∃ us' : Sequence patchUniverse oq ou',
     ∃ r'' : pu_type ou' our',
     ∃ s'' : pu_type our' ours',
     ∃ vs' : Sequence patchUniverse ours' oqrs,
     ∃ s''_vs'ConsOK : ConsOK s'' vs',
     ∃ r''_s''_vs'ConsOK : ConsOK r'' (s'' :> vs'),
     ∃ us'_r''_s''_vs'AppendOK : AppendOK us' (r'' :> s'' :> vs'),
         TransitiveCommuteRelates ([] :+> r :> s :> [])
                                  (us' :+> r'' :> s'' :> vs') ∧
         (pu_nameOf r = pu_nameOf r') ∧
         (pu_nameOf s = pu_nameOf s') ∧
         ~(r'' <~?~> s'')).
Proof with trivial.
intros.
destruct qs.
rename s0 into qs.
move qs at top.
revert oqr oqrs ou our ours r s us r' s' vs s_NilConsOK r_s_tsConsOK qs_r_s_NilAppendOK
s'_vsConsOK r'_s'_vsConsOK us_r'_s'_vsAppendOK transitive_commute rs_same ss_same
r_s_commutable r'_s'_not_commutable.
dependent induction qs.
    (*   case *)
    intros.
    solveExists.
     constructor.
     rewrite SequenceContentsNil.
     signedNameSetDec.
    ∃ ou.
    ∃ our.
    ∃ ours.
    ∃ us.
    ∃ r'.
    ∃ s'.
    ∃ vs.
    ∃ s'_vsConsOK.
    ∃ r'_s'_vsConsOK.
    ∃ us_r'_s'_vsAppendOK.
    split.
     rewrite AppendNil.
     rewrite AppendNilSeq in transitive_commute...
    split...
    split...
(*  Cons case *)
intros.
unfold block in IHqs.
destruct (AppendConsSeq p qs (r :> s :> [])) as [? [? [? ?]]].
rewrite H in transitive_commute.
set (HX := ShorterCounterExampleExists _ _ _ _ _ _ _ transitive_commute rs_same ss_same
r_s_commutable r'_s'_not_commutable).
destruct HX as [ou' [our' [ours'
```

```
                        [us' [r'' [s'' [vs'
                        [s''_vs'ConsOK [r''_s''_vs'ConsOK [us'_r''_s''_vs'AppendOK
                        [transitive_commute_3 [r_r''_same [s_s''_same r''_s''_not_commutable
                        ]]]]]]]]]]]].
specialize (IHqs x).
specialize (IHqs oqr oqrs).
specialize (IHqs ou' our' ours').
specialize (IHqs r s).
specialize (IHqs us' r'' s'' vs').
specialize (IHqs s_NilConsOK).
specialize (IHqs r_s_tsConsOK).
specialize (IHqs x0).
specialize (IHqs s''_vs'ConsOK).
specialize (IHqs r''_s''_vs'ConsOK).
specialize (IHqs us'_r''_s''_vs'AppendOK).
specialize (IHqs transitive_commute_3).
specialize (IHqs r_r''_same s_s''_same).
specialize (IHqs r_s_commutable r''_s''_not_commutable).
destruct IHqs as [lNil_r_s_NilAppendOK
                  [ou'' [our'' [ours''
                  [us'' [r''' [s''' [vs''
                  [s'''_vs''ConsOK
                  [r'''_s'''_vs''ConsOK
                  [us''_r'''_s'''_vs''AppendOK
                    [? [? [? ?]]]
]]]]]]]]]]].
∃ lNil_r_s_NilAppendOK.
∃ ou''.
∃ our''.
∃ ours''.
∃ us''.
∃ r'''.
∃ s'''.
∃ vs''.
∃ s'''_vs''ConsOK.
∃ r'''_s'''_vs''ConsOK.
∃ us''_r'''_s'''_vs''AppendOK.
repeat split...
Qed.

Lemma TwoSequenceTransitiveCommuteRelatesTwoSequence1 :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {o or ors ou our ours : NameSet}
           {r : pu_type o or}
           {s : pu_type or ors}
           {us : Sequence patchUniverse o ou}
           {r' : pu_type ou our}
           {s' : pu_type our ours}
           {vs : Sequence patchUniverse ours ors}
```

```
              (s_NilConsOK : ConsOK s [])
              (r_s_NilConsOK : ConsOK r (s :> []))
              (Nil_r_s_NilAppendOK : AppendOK [] (r :> (s :> [])))
              (s'_vsConsOK : ConsOK s' vs)
              (r'_s'_vsConsOK : ConsOK r' (s' :> vs))
              (us_r'_s'_vsAppendOK : AppendOK us (r' :> (s' :> vs)))
              ,
      («[] :+> r :> s :> []» <~~>*
       «us :+> r' :> s' :> vs»)
  → (ou = o) ∧ (ours = ors).
Proof with auto.
intros.
rewrite AppendNil in H. (*  XXX Remove this, and don't take  *)
dependent induction H.
    (*  This is the case where the transitive commute is finished *)
    destruct us.
    dependent destruction s0.
        (*  This is the case where the original us is  *)
        split...
        rewrite AppendNilSeq in x.
        consEquality x.
        consEquality x...
    destruct (AppendConsSeq p s0 (r' :> s' :> vs)) as [? [? [? Heq1]]].
    rewrite Heq1 in x.
    consEquality x.
    destruct s0.
        rewrite AppendNilSeq in x.
        consEquality x.
        symmetry in x.
        contradiction (ConsEqNil x).
    destruct (AppendConsSeq p0 s0 (r' :> s' :> vs)) as [? [? [? Heq2]]].
    rewrite Heq2 in x.
    consEquality x.
    destruct s0.
        rewrite AppendNilSeq in x.
        symmetry in x.
        contradiction (ConsEqNil x).
    destruct (AppendConsSeq p1 s0 (r' :> s' :> vs)) as [? [? [? Heq3]]].
    rewrite Heq3 in x.
    symmetry in x.
    contradiction (ConsEqNil x).
dependent destruction H.
    consEquality x.
    consEquality x.
    rename IHTransitiveCommuteRelates into IH.
    specialize (IH ours).
    specialize (IH vs).
    specialize (IH our).
    specialize (IH s').
    specialize (IH s'_vsConsOK).
    specialize (IH ou).
    specialize (IH us).
    specialize (IH r').
    specialize (IH r'_s'_vsConsOK).
    specialize (IH us_r'_s'_vsAppendOK).
    specialize (IH oq).
    specialize (IH q').
    specialize (IH p').
    specialize (IH pRsConsOK).
    specialize (IH qPRsConsOK).
    solveFirstIn IH.
        constructor.
        rewrite SequenceContentsNil.
        signedNameSetDec.
    specialize (IH eq_refl).
    specialize (IH eq_refl).
    unfold block in IH...
consEquality x.
dependent destruction H.
    consEquality x.
    contradiction (ConsEqNil x).
consEquality x.
dependent destruction H.
    contradiction (ConsEqNil x).
contradiction (ConsEqNil x).
Qed.

Lemma TwoSequenceTransitiveCommuteRelatesTwoSequence2 :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {o or ors our: NameSet}
           {r : pu_type o or}
           {s : pu_type or ors}
           {us : Sequence patchUniverse o o}
           {r' : pu_type o our}
           {s' : pu_type our ors}
           {vs : Sequence patchUniverse ors ors}

           (s_NilConsOK : ConsOK s [])
           (r_s_NilConsOK : ConsOK r (s :> []))
           (Nil_r_s_NilAppendOK : AppendOK [] (r :> (s :> [])))
           (s'_vsConsOK : ConsOK s' vs)
           (r'_s'_vsConsOK : ConsOK r' (s' :> vs))
           (us_r'_s'_vsAppendOK : AppendOK us (r' :> (s' :> vs)))
           ,

    («[] :+> r :> s :> []» <~~>* «us :+> r' :> s' :> vs»)
 → ((us = []) ∧ (vs = [])).
Proof with auto.
intros.
rewrite AppendNil in H. (*  XXX Remove this, and don't take  *)
dependent induction H.
    destruct us.
    dependent destruction s0.
        split.
            unfold nilSeq.
            f_equal.
            apply proof_irrelevance.
        rewrite AppendNilSeq in x.
        consEquality x.
        consEquality x...
    destruct (AppendConsSeq p s0 (r' :> s' :> vs)) as [? [? [? Heq1]]].
    rewrite Heq1 in x.
    consEquality x.
    destruct s0.
        rewrite AppendNilSeq in x.
        consEquality x.
        symmetry in x.
        contradiction (ConsEqNil x).
    destruct (AppendConsSeq p0 s0 (r' :> s' :> vs)) as [? [? [? Heq2]]].
    rewrite Heq2 in x.
    consEquality x.
    destruct s0.
        rewrite AppendNilSeq in x.
        symmetry in x.
        contradiction (ConsEqNil x).
    destruct (AppendConsSeq p1 s0 (r' :> s' :> vs)) as [? [? [? Heq3]]].
    rewrite Heq3 in x.
    symmetry in x.
    contradiction (ConsEqNil x).
dependent destruction H.
    consEquality x.
    consEquality x.

    rename IHTransitiveCommuteRelates into IH.
    specialize (IH vs).
    specialize (IH us).
    specialize (IH our).
    specialize (IH r').
    specialize (IH s').
    specialize (IH s'_vsConsOK).
    specialize (IH r'_s'_vsConsOK).
    specialize (IH us_r'_s'_vsAppendOK).
    specialize (IH oq).
    specialize (IH q').
    specialize (IH p').
    specialize (IH pRsConsOK).
    specialize (IH qPRsConsOK).
    solveFirstIn IH.
        constructor.
        rewrite SequenceContentsNil.
        signedNameSetDec.
    specialize (IH eq_refl).
    specialize (IH eq_refl).
    unfold block in IH...
consEquality x.
```

```
dependent destruction H.
    consEquality x.
    contradiction (ConsEqNil x).
consEquality x.
dependent destruction H.
    contradiction (ConsEqNil x).
contradiction (ConsEqNil x).
Qed.

Lemma commuteInSequenceConsistent :
    ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {patchUniverse : PatchUniverse pui}
           {o oq oqr oqrs (*  oqrst *) ou our ours : NameSet}
           {qs : Sequence patchUniverse o oq}
           {r : pu_type oq oqr}
           {s : pu_type oqr oqrs}
        (*  {ts : Sequence patchUniverse _ oqrs oqrst} *)
           {us : Sequence patchUniverse o ou}
           {r' : pu_type ou our}
           {s' : pu_type our ours}
           {vs : Sequence patchUniverse ours oqrs}

           (s_NilConsOK : ConsOK s [])
           (r_s_NilConsOK : ConsOK r (s :> []))
           (qs_r_s_NilAppendOK : AppendOK qs (r :> s :> []))

           (s'_vsConsOK : ConsOK s' vs)
           (r'_s'_vsConsOK : ConsOK r' (s' :> vs))
           (us_r'_s'_vsAppendOK : AppendOK us (r' :> (s' :> vs)))

           (transitive_commute : «qs :+> r :> s :> []» <~~>* «us :+> r' :> s' :> vs»)
           (rs_same : pu_nameOf r = pu_nameOf r')
           (ss_same : pu_nameOf s = pu_nameOf s')
           (r_s_commutable : r <~?~> s),
    (r' <~?~> s').
Proof with trivial.
intros.
case (commutable_dec r' s')...
    intros.
    destruct s0 as [? [? [? ?]]].
    ∃ x.
    ∃ x0.
    ∃ x1...
intro r'_s'_not_commutable.
destruct (EmptyCounterExampleExists _ _ _ _ _ _ transitive_commute rs_same ss_same
r_s_commutable r'_s'_not_commutable)
    as [? [? [? [? [? [r'' [s'' [? [? [? [? [HX [? [? not_commutable]]]]]]]]]]]]]].
(*  XXX rewrite AppendNil in HX. *)
assert (HY : «[] :+> r :> s :> []» <~~>* «[] :+> r :> s :> []»).
    apply Same.
(*  XXX rewrite AppendNil in HY. *)
```

```
destruct (TwoSequenceTransitiveCommuteRelatesTwoSequence1 _ _ _ _ _ _ HX).
destruct (TwoSequenceTransitiveCommuteRelatesTwoSequence1 _ _ _ _ _ _ HY).
subst.
destruct (TwoSequenceTransitiveCommuteRelatesTwoSequence2 _ _ _ _ _ _ HX) as [? ?].
destruct (TwoSequenceTransitiveCommuteRelatesTwoSequence2 _ _ _ _ _ _ HY) as [? ?].
subst.
assert (r'' <~?~> s'').
    apply SymmetricTransitiveCommute in HX.
    apply (TransitiveTransitiveCommute HX) in HY.
    dependent destruction HX...
    rewrite AppendNil in HY.
    rewrite AppendNil in H1.
    dependent destruction H1.
        consEquality x.
        consEquality x.
        ∃ oq0.
        ∃ q'.
        ∃ p'...
    consEquality x.
    dependent destruction H1.
        consEquality x.
        contradiction (ConsEqNil x).
    consEquality x.
    dependent destruction H1.
        contradiction (ConsEqNil x).
    contradiction (ConsEqNil x).
contradiction.
Qed.
```

XXX Is this right? Looks wrong to me now:

**Lemma 9.2 (patch-in-sequence-has-minimal-context)**
If $\overline{p}q\overline{r}s\overline{t} \leftrightarrow^* \overline{u}q'\overline{v}$ (where $n(q) = n(q')$) and $n(q) \notin N(\overline{u})$, then $q \underset{?}{\leftrightarrow} \overline{r}s$.

**Explanation**
*Within a sequence, a patch has a minimal context.*

**Proof**
Suppose, for the purpose of contraction, that we have a counterexample. Given $s$, let us consider the right-most $q$ for which the property doesn't hold. Then $N(\overline{r}) \subseteq N(\overline{u})$, so at some point during the $\leftrightarrow^*$ relation $q$ commuted with each of them, and with $s$. Therefore, by Lemma 9.1, it can be commuted past them all. But we assumed that it could not. Contradiction. ■

**Lemma 9.3 (patch-in-universe-has-minimal-context)**
If $\overline{ps}q\overline{r}s\overline{t}$ and $\overline{u}q'\overline{v}$, where $n(q) = n(q')$ and $n(q) \notin N(\overline{u})$, are in a patch universe then $\langle q, \overline{r}s\rangle \leftrightarrow \langle \overline{r'}s', q''\rangle$.

**Explanation**
*Within a patch universe, a patch has a minimal context.*

**Proof**
When a patch is first recorded, Lemma 9.2 tells us that it has a minimal context.

Another patch being recorded or unrecorded trivially doesn't alter its minimal context, as both operations act on the end of the patch sequence.

Commute trivially doesn't affect the minimal context, as the sequence doesn't change.

When merging, the patch is either in the common prefix or one of the suffixes. If it is in the common prefix or the suffix that remains unchanged then the minimal context is unaffected for the same reason that it is when recording new patches. If it is in the other suffix, then the definition of $\oplus$ tells us that we can commute it to give us the case where it is in the first suffix, so this case is satisfied too. ■

XXX This is one of the important theorems: That given a patch universe, we can get to the point that the merge algorithm starts at

**Theorem 9.1 (patch-universe-provides-merge-input)**
Suppose we have two patch sequences $\overline{p}$ and $\overline{q}$.

Then $\langle\overline{p}\rangle \leftrightarrow^* \langle\overline{rs}\rangle$ and $\langle\overline{q}\rangle \leftrightarrow^* \langle\overline{rt}\rangle$ where $N\left(\overline{s}\right) \cap N\left(\overline{t}\right) = \emptyset$.

**Explanation**
*XXX*

**Proof**
XXX ■

XXX

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

```
End patch_universes.
```

coqdoc

printing <~>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ merging

## 10 Merging

```
Module Export merging.

Require Import names.
Require Import patch_universes.
Require Import patch_universes_sequences.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.

(*  XXX *)
Axiom cheat : ∀ {a}, a.

Definition commuteOutCommonPrefix
    {pu_type : NameSet → NameSet → Type}
    {ppu : PartPatchUniverse pu_type pu_type}
    {pui : PatchUniverseInv ppu ppu}
    {pu : PatchUniverse pui}
    {from tox toy : NameSet}
```

```
      (xs : Sequence pu from tox)
      (ys : Sequence pu from toy)
   : { from' : NameSet &
         prod (Sequence pu from' tox)
                (Sequence pu from' toy) }
      := cheat.

Lemma commuteOutCommonPrefixCons
      {pu_type : NameSet → NameSet → Type}
      {ppu : PartPatchUniverse pu_type pu_type}
      {pui : PatchUniverseInv ppu ppu}
      {pu : PatchUniverse pui}
      {ipl : InvertiblePatchlike pu_type}
      {from mid to1 to2 : NameSet}
      (p : pu_type from mid)
      (xs : Sequence pu mid to1)
      (ys : Sequence pu mid to2)
      {pXsConsOK : ConsOK p xs}
      {pYsConsOK : ConsOK p ys}
   : commuteOutCommonPrefix xs ys
   = commuteOutCommonPrefix (p :> xs) (p :> ys).
Proof.
admit.
Qed.

Definition mergable {pu_type : NameSet → NameSet → Type}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {ipl : InvertiblePatchlike pu_type}
                        {ipu : InvertiblePatchUniverse pu ipl}
                        {from tox toy : NameSet}
                        (xs : Sequence pu from tox)
                        (ys : Sequence pu from toy)
                      : Prop
      := match commuteOutCommonPrefix xs ys with
          existT _ (pair xs' ys') ⇒ xs'^ <~?~> ys'
          end.

Lemma mergable_dec {pu_type : NameSet → NameSet → Type}
                       {ppu : PartPatchUniverse pu_type pu_type}
                       {pui : PatchUniverseInv ppu ppu}
                       {pu : PatchUniverse pui}
                       {ipl : InvertiblePatchlike pu_type}
                       {ipu : InvertiblePatchUniverse pu ipl}
                       {fromx mid1x mid2x tox : NameSet}
                       {p : Sequence pu fromx mid1x}
                       {q : Sequence pu fromx mid2x}
                     : { mergable p q }
                     + {~(mergable p q)}.
Proof with auto.
unfold mergable.
destruct (commuteOutCommonPrefix p q) as [newFrom [xs ys]].
destruct (commutable_dec (xs^) ys).
```

```
    left.
    destruct s as [to [ys' [xs'inv commute]]].
    ∃ to.
    ∃ ys'.
    ∃ xs'inv...
right...
Qed.

Lemma mergableCons1
    {pu_type : NameSet → NameSet → Type}
    {ppu : PartPatchUniverse pu_type pu_type}
    {pui : PatchUniverseInv ppu ppu}
    {pu : PatchUniverse pui}
    {ipl : InvertiblePatchlike pu_type}
    {ipu : InvertiblePatchUniverse pu ipl}
    {from mid to1 to2 : NameSet}
    (p : pu_type from mid)
    {xs : Sequence pu mid to1}
    {ys : Sequence pu mid to2}
    {pXsConsOK : ConsOK p xs}
    {pYsConsOK : ConsOK p ys}
    (mergableXsYs : mergable xs ys)
  : mergable (p :> xs) (p :> ys).
Proof with auto.
unfold mergable.
rewrite ← (commuteOutCommonPrefixCons p xs ys).
fold (mergable xs ys)...
Qed.

Lemma mergableCons2
    {pu_type : NameSet → NameSet → Type}
    {ppu : PartPatchUniverse pu_type pu_type}
    {pui : PatchUniverseInv ppu ppu}
    {pu : PatchUniverse pui}
    {ipl : InvertiblePatchlike pu_type}
    {ipu : InvertiblePatchUniverse pu ipl}
    {from mid to1 to2 : NameSet}
    {p : pu_type from mid}
    {xs : Sequence pu mid to1}
    {ys : Sequence pu mid to2}
    {pXsConsOK : ConsOK p xs}
    {pYsConsOK : ConsOK p ys}
    (mergablePXsPYs : mergable (p :> xs) (p :> ys))
  : mergable xs ys.
Proof with auto.
unfold mergable.
rewrite (commuteOutCommonPrefixCons p xs ys).
fold (mergable (p :> xs) (p :> ys))...
Qed.

Lemma mergableAppend1
    {pu_type : NameSet → NameSet → Type}
    {ppu : PartPatchUniverse pu_type pu_type}
    {pui : PatchUniverseInv ppu ppu}
    {pu : PatchUniverse pui}
    {ipl : InvertiblePatchlike pu_type}
    {ipu : InvertiblePatchUniverse pu ipl}
    {from mid to1 to2 : NameSet}
    (ps : Sequence pu from mid)
    {xs : Sequence pu mid to1}
    {ys : Sequence pu mid to2}
    {psXsConsOK : AppendOK ps xs}
    {psYsConsOK : AppendOK ps ys}
    (mergableXsYs : mergable xs ys)
  : mergable (ps :+> xs) (ps :+> ys).
Proof with auto.
destruct ps as [ps psSequenceOK].
induction ps.
    rewrite AppendNilSeq.
    rewrite AppendNilSeq...
destruct (ConsSeqToCons p ps) as [XpsSequenceOK [XpPsConsOK XpPsEquality]].
set (X := MkSequence (Cons p ps) psSequenceOK).
generalize (eq_refl X).
(*
unfold X at 2.
rewrite XpPsEquality.
intro.
rewrite H.
clearbody X.
clearbody X.

consSeqToCons p ps.
clearbody X.
destruct (ConsSeqToCons p ps) as XpsSequenceOK [XpPsConsOK XpPsEquality].
rewrite XpPsEquality.
rewrite AppendConsSeq.
unfold mergable.
rewrite <- (commuteOutCommonPrefixCons p xs ys).
fold (mergable xs ys)...
*)
admit.
Qed.
Lemma mergableAppend2
    {pu_type : NameSet → NameSet → Type}
    {ppu : PartPatchUniverse pu_type pu_type}
    {pui : PatchUniverseInv ppu ppu}
    {pu : PatchUniverse pui}
    {ipl : InvertiblePatchlike pu_type}
    {ipu : InvertiblePatchUniverse pu ipl}
    {from mid to1 to2 : NameSet}
    (ps : Sequence pu from mid)
    {xs : Sequence pu mid to1}
    {ys : Sequence pu mid to2}
    {psXsConsOK : AppendOK ps xs}
    {psYsConsOK : AppendOK ps ys}
    (mergablePXsPYs : mergable (ps :+> xs) (ps :+> ys))
  : mergable xs ys.
Proof with auto.
destruct ps as [ps psSequenceOK].
induction ps.
    rewrite AppendNilSeq in mergablePXsPYs.
    rewrite AppendNilSeq in mergablePXsPYs...
destruct (ConsSeqToCons p ps) as [XpsSequenceOK [XpPsConsOK XpPsEquality]].
(*
rewrite XpPsEquality.
set (X := MkSequence (Cons p ps) psSequenceOK).
generalize (eq_refl X).
unfold mergable.
rewrite (commuteOutCommonPrefixCons p xs ys).
fold (mergable (p :> xs) (p :> ys))...
*)
admit.
Qed.

Lemma mergableSymmetric
    {pu_type : NameSet → NameSet → Type}
    {ppu : PartPatchUniverse pu_type pu_type}
    {pui : PatchUniverseInv ppu ppu}
    {pu : PatchUniverse pui}
    {ipl : InvertiblePatchlike pu_type}
    {ipu : InvertiblePatchUniverse pu ipl}
    {from to1 to2 : NameSet}
    {xs : Sequence pu from to1}
    {ys : Sequence pu from to2}
    (mergableXsYs : mergable xs ys)
  : mergable ys xs.
Proof with auto.
admit.
Qed.

End merging.
```

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ contexted_patches

## 11 Contexted patches

```
Module Export contexted_patches.

Require Import Equality.
Require Import names.
Require Import patch_universes.
```

**Definition 11.1 (contexted-patch)**
For all patches $r \in \mathbf{P}$, $: r$ is a *contexted patch* of $r$. For all contexted patches $\overline{q} : r$ and patches $p \in \mathbf{P}$, $p\overline{q} : r$ is a *contexted patch* of $r$ if and only if $\langle p, \overline{q}r\rangle \leftrightarrow$ fail and $\overline{q} \neq p^{-1}\overline{q}'$ for some

sequence $\overline{q}'$.

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

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

```
Class ContextedPatchOK {pu_type : NameSet → NameSet → Type}
                       {ppu : PartPatchUniverse pu_type pu_type}
                       {pui : PatchUniverseInv ppu ppu}
                       {pu : PatchUniverse pui}
                       {from mid to : NameSet}
                       (c : Sequence pu from mid)
                       (p : pu_type mid to)
    := {
           contextedPatchNotInContext : ¬ SignedNameSetIn (pu_nameOf p) (sequence-
Contents c)
       }.
Implicit Arguments ContextedPatchOK [pu_type ppu pui pu from mid to].
Instance EmptyContextedPatchOK
    : ∀ {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {pu : PatchUniverse pui}
            {from to : NameSet}
            (p : pu_type from to),
      ContextedPatchOK [] p.
Proof.
intros.
constructor.
rewrite SequenceContentsNil.
signedNameSetDec.
Qed.
Inductive ContextedPatch {pu_type : (NameSet → NameSet → Type)}
                         {ppu : PartPatchUniverse pu_type pu_type}
                         {pui : PatchUniverseInv ppu ppu}
                         (pu : PatchUniverse pui)
                         (from : NameSet)
                       : Type
    := MkContextedPatch : ∀ {mid to : NameSet}
                                (c : Sequence pu from mid)
                                (p : pu_type mid to)
                                {contextedPatchOK : ContextedPatchOK c p},
                  ContextedPatch pu from.
Implicit Arguments MkContextedPatch [pu_type ppu pui pu from mid to contextedPatchOK].
(*  XXX Describe this in the latex: *)
Definition contextedPatch_name {pu_type : (NameSet → NameSet → Type)}
                               {ppu : PartPatchUniverse pu_type pu_type}
                               {pui : PatchUniverseInv ppu ppu}
```

```
                                               {pu : PatchUniverse pui}
                                               {from : NameSet}
                                               (c : ContextedPatch pu from) : SignedName
  := match c with
     | MkContextedPatch _ _ _ p _ ⇒ pu_nameOf p
     end.
```

**Definition 11.2 (patch-commute-past)**
Given a patch $p$ and a contexted patch $\overline{q} : r$, we define $\rightarrow$, pronounced *commute past*, thus:

$\langle p, \overline{q} : r\rangle \rightarrow \left\langle \overline{q}' : r' \right\rangle$ if $(\langle p, \overline{q}\rangle \leftrightarrow \left\langle \overline{q}', p' \right\rangle) \wedge (\langle p', r\rangle \leftrightarrow \langle r', p''\rangle)$
$\langle p, \overline{q} : r\rangle \rightarrow \text{fail}$ otherwise

```
Axiom cheat : ∀ {a}, a.

Inductive CommutePast {pu_type : (NameSet → NameSet → Type)}
                      {ppu : PartPatchUniverse pu_type pu_type}
                      {pui : PatchUniverseInv ppu ppu}
                      {pu : PatchUniverse pui}
                    : ∀ {from mid : NameSet},
                      pu_type from mid
                    → ContextedPatch pu mid
                    → ContextedPatch pu from
                    → Prop
    := CommutePastNil :
                 ∀ (from mid mid' to : NameSet)
                        (p : pu_type from mid)
                        (ident : pu_type mid to)
                        (ident' : pu_type from mid')
                        (p' : pu_type mid' to)
                        (H : «p, ident» <~> «ident', p'»),
                 CommutePast p
                             (MkContextedPatch [] ident)
                             (MkContextedPatch [] ident')
     | CommutePastCons :
                 ∀ (rsContains qRsContains rs'Contains q'Rs'Contains : SignedNameSet)
                        (o op opq opqr opqrs oq oqr oqrs : NameSet)
                        (p : pu_type o op)
                        (contextQ : pu_type op opq)
                        (contextRs : Sequence pu opq opqr)
                        (ident : pu_type opqr opqrs)

                        (contextQ' : pu_type o oq)
                        (contextRs' : Sequence pu oq oqr)
                        (ident' : pu_type oqr oqrs)
                        (p' : pu_type oq opq)

                        (contextRsIdentOK : ContextedPatchOK contextRs ident)
                        (contextRs'Ident'OK : ContextedPatchOK contextRs' ident')

                        (H : CommutePast p'
                             (MkContextedPatch contextRs ident)
                             (MkContextedPatch contextRs' ident'))
                        (qRsConsOK : ConsOK contextQ contextRs)
                        (q'Rs'ConsOK : ConsOK contextQ' contextRs')

                        (contextQContextRsIdentOK : ContextedPatchOK (contextQ :>
contextRs) ident)
                        (contextQ'ContextRs'Ident'OK : ContextedPatchOK (contextQ' :>
contextRs') ident'),
                  CommutePast p
                              (MkContextedPatch (contextQ :> contextRs) ident)
                              (MkContextedPatch (contextQ' :> contextRs') ident').

Definition CommutePastable {pu_type : (NameSet → NameSet → Type)}
                           {ppu : PartPatchUniverse pu_type pu_type}
                           {pui : PatchUniverseInv ppu ppu}
                           {pu : PatchUniverse pui}
                           {from mid : NameSet}
                           (p : pu_type from mid)
                           (cp : ContextedPatch pu mid)
                         : Prop
 := (∃ cp' : ContextedPatch pu from,
     CommutePast p cp cp').

Lemma CommutePastable_dec {pu_type : (NameSet → NameSet → Type)}
                          {ppu : PartPatchUniverse pu_type pu_type}
                          {pui : PatchUniverseInv ppu ppu}
                          {pu : PatchUniverse pui}
                          {from mid : NameSet}
                          (p : pu_type from mid)
                          (cp : ContextedPatch pu mid)
                        : {CommutePastable p cp} + {~CommutePastable p cp}.
Proof.
intros.
destruct cp.
dependent induction c.
(*
XXX
    destruct nilOK.
    remember (NameSetEquality nilFromEqTo0) as H.
    clear HeqH.
    subst.
    destruct (commutable_dec p p0).
        left.
        destruct c as ?  [p0' [p' ?]].
        assert (nilFromOK : NilOK from from SignedNameSetMod.empty).
            constructor.
                nameSetDec.
            signedNameSetDec.
        exists (MkContextedPatch pu from  p0').
        (*  XXX Tidy this up: *)
        apply (CommutePastNil
                   SignedNameSetMod.empty
                   contains
                   from
```

```
                        to0
                        x
                        to
                        p
                        p0
                        p0'
                        p'
                        H
                        _
                        _
                       ).
    right.
    admit. (*  XXX *)
*)
admit. (*  XXX *)
Qed.

Inductive CommuteManyPast {pu_type : (NameSet → NameSet → Type)}
                          {ppu : PartPatchUniverse pu_type pu_type}
                          {pui : PatchUniverseInv ppu ppu}
                          {pu : PatchUniverse pui}
                        : ∀ {from mid : NameSet},
                          Sequence pu from mid
                        → ContextedPatch pu mid
                        → ContextedPatch pu from
                        → Prop
    := CommuteNilPast : ∀ (fromNilContains : SignedNameSet)
                               (from : NameSet)
                               (cp : ContextedPatch pu from),
                         CommuteManyPast [] cp cp
     | CommuteConsPast : ∀ (qsContains pQsContains : SignedNameSet)
                                (o op opq : NameSet)
                                (p : pu_type o op)
                                (qs : Sequence pu op opq)
                                (cp : ContextedPatch pu opq)
                                (cp' : ContextedPatch pu op)
                                (cp'' : ContextedPatch pu o)
                                (CommuteQsPast : CommuteManyPast qs cp cp')
                                (CommutePPast : CommutePast p cp' cp'')
                                (consOK : ConsOK p qs),
                          CommuteManyPast (p :> qs) cp cp''.

(*
Definition commutePast {pu : PatchUniverse}
                       {from mid : NameSet}
                       (p : (pu_type pu) from mid)
                       (c : ContextedPatch pu mid)
                     : option (ContextedPatch pu from)
 := match c with
    (*  XXX Should have Nil = cContext *)
    | MkContextedPatch _ _ _ (Nil _) cPatch =>
        match (commutable_dec pu) p (cheat cPatch) with
        | left _ => None
```

```
                | right _ => None
                end
          | _ => None
          end.
    *)
```

**Definition 11.3 (patch-commute-past-set)**
We extend $\rightarrow$ to work on sets of contexted patches in the natural way, i.e. for any patch $p$ and set of contexted patches $S$:
$\langle p, S\rangle \rightarrow \left\langle\left\{(\overline{q}' : r') \mid (\overline{q} : r) \in S \wedge \langle p, \overline{q} : r\rangle \rightarrow \left\langle \overline{q}' : r'\right\rangle\right\}\right\rangle$
if all the commute pasts succeed, and $\langle p, S\rangle \rightarrow$ fail otherwise.

From here on we assume that contexted patches magically maintain their invariant, i.e. if we have a patch $p$ and a contexted patch $\overline{q} : r$ then when we write $p\overline{q} : r$ what we mean is, if $\langle p, \overline{q} : r\rangle \rightarrow \left\langle \overline{q}' : r'\right\rangle$ then $\overline{q}' : r'$, otherwise $p\overline{q} : r$.

```
Definition addToContext {pu_type : (NameSet → NameSet → Type)}
                         {ppu : PartPatchUniverse pu_type pu_type}
                         {pui : PatchUniverseInv ppu ppu}
                         {pu : PatchUniverse pui}
                         {from mid : NameSet}
                         (p : pu_type from mid)
                         (c : ContextedPatch pu mid)
                       : ContextedPatch pu from
 := cheat.
(*
match c with
     | MkContextedPatch _ _ _ _ p => pu_nameOf _ p
     end.
*)
Program Fixpoint addSequenceBaseToContext
                        {pu_type : (NameSet → NameSet → Type)}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {from mid : NameSet}
                        (ps : SequenceBase pu from mid)
                        (c : ContextedPatch pu mid)
                      : ContextedPatch pu from
 := match ps with
     | Nil _ ⇒ c
     | Cons _ _ _ p ps' ⇒
           (addToContext p (addSequenceBaseToContext ps' c))
     end.
Definition addSequenceToContext
                        {pu_type : (NameSet → NameSet → Type)}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {from mid : NameSet}
                        (ps : Sequence pu from mid)
```

```
                              (c : ContextedPatch pu mid)
                            : ContextedPatch pu from
 := match ps with
    | MkSequence s _ ⇒ addSequenceBaseToContext s c
    end.
```

**Definition 11.4 (contexted-patch-conflict)**
We define $\leftrightarrows$, pronounced "does not conflict with", such that $(\overline{p} : q) \leftrightarrows (\overline{r} : s)$ holds if $\langle q^{-1}, \overline{p}^{-1}\overline{r} : s\rangle \rightarrow \langle\_\rangle$, and does not hold otherwise.

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

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

```
Definition invertContextedPatch {pu_type : (NameSet → NameSet → Type)}
                                    {ppu : PartPatchUniverse pu_type pu_type}
                                    {pui : PatchUniverseInv ppu ppu}
                                    {pu : PatchUniverse pui}
                                    {from : NameSet}
                                    (c : ContextedPatch pu from)
                                  : ContextedPatch pu from
 := cheat.
(*
 := match c with
    | MkContextedPatch _ _ _ ps ident =>
          addSequenceToContext ps
              (addToContext ident
                   (MkContextedPatch _ _  (invert _ ident)))
    end.
*)
End contexted_patches.
```

coqdoc

printing <~>u $\leftrightsquigarrow_u$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_definition

## 12 Catches and Repos

```
Module Export catches_definition.
Require Import Equality.
Require Import List.
Require Import util.
Require Import names.
Require Import patch_universes.
Require Import patch_universes_sequences.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import commute_square.
Require Import merging.
Require Import contexted_patches.
Require Import canonical_order.
```

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

**Explanation**
*The basic idea is that, if two catches do not conflict, then we can merge them, similar to the way that we merge patches. However, if they do conflict then when we merge, we get a conflictor which records the conflict. To compute the contents of a repository we take the effects of all the patches that are in the repository and do not conflict with any other patch in the repository.*

**Definition 12.1 (catches)**
A catch is either $[p]$ (the non-conflicted patch $p \in \mathbf{P}$), $\lceil \overline{r}, X, \overline{p} : q \rceil$ (a conflicted patch $q \in \mathbf{P}$), or $[\![\overline{r}, X, \overline{p} : q]\!]$ (the inverse of a conflicted patch $q \in \mathbf{P}$). In both cases, $\overline{r}$ is a sequence of patches, $X$ is a set of contexted patches, and $\overline{p} : q$ is a contexted patch.

```
Program Definition conflictsWith {pu_type : NameSet → NameSet → Type}
                                 {ppu : PartPatchUniverse pu_type pu_type}
                                 {pui : PatchUniverseInv ppu ppu}
                                 {pu : PatchUniverse pui}
                                 {ipl : InvertiblePatchlike pu_type}
                                 {ipu : InvertiblePatchUniverse pu ipl}
                                 {from : NameSet}
                                 (p : ContextedPatch pu from)
                                 (q : ContextedPatch pu from)
                               : Prop
    := match p, q with
        | MkContextedPatch _ _ psContext psPatch _,
          MkContextedPatch _ _ qsContext qsPatch _ ⇒
              ~(mergable (psContext :+> psPatch :> [])
                         (qsContext :+> qsPatch :> []))
        end.
Next Obligation.
constructor.
rewrite SequenceContentsNil.
signedNameSetDec.
Qed.
Next Obligation.
constructor.
destruct wildcard'1.
destruct psContext.
```

```
unfold sequenceContents in *.
rewrite SequenceContentsBaseCons.
rewrite SequenceContentsBaseNil.
(*  XXX coq bug: signedNameSetDec. *)
admit.
Qed.
Next Obligation.
constructor.
rewrite SequenceContentsNil.
signedNameSetDec.
Qed.
Next Obligation.
constructor.
destruct wildcard'4.
destruct qsContext.
unfold sequenceContents in *.
rewrite SequenceContentsBaseCons.
rewrite SequenceContentsBaseNil.
(*  XXX coq bug: signedNameSetDec. *)
admit.
Qed.

Inductive Catch {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                (ipl : InvertiblePatchlike pu_type)
                {ipu : InvertiblePatchUniverse pu ipl}
                (from to : NameSet)
              : Type
    := MkCatch : ∀ (p : pu_type from to),
                  Catch ipl from to
     | Conflictor : ∀ (effect : Sequence pu from to)
                           (conflicts : list (ContextedPatch pu to))
                           (ident : ContextedPatch pu to)
                           (*  XXX Proof conflicts /= ?
                                  Proof effect^ \subseteq conflicts?
                                  Proof conflicts no dupes? *)
                           (allConflict : Forall (conflictsWith ident) conflicts)
                           (effectCanonicallyOrdered : CanonicallyOrdered effect),
                     Catch ipl from to.
Implicit Arguments MkCatch [pu_type ppu pui pu ipl ipu from to].
Implicit Arguments Conflictor [pu_type ppu pui pu ipl ipu from to].
```

**Definition 12.2 (repos)**
A *repo* is a sequence of catches (up to commutation, to be defined later) satisfying some requirements (that, again, we will give later).

Let **R** be the (possibly infinite) set of repos.

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

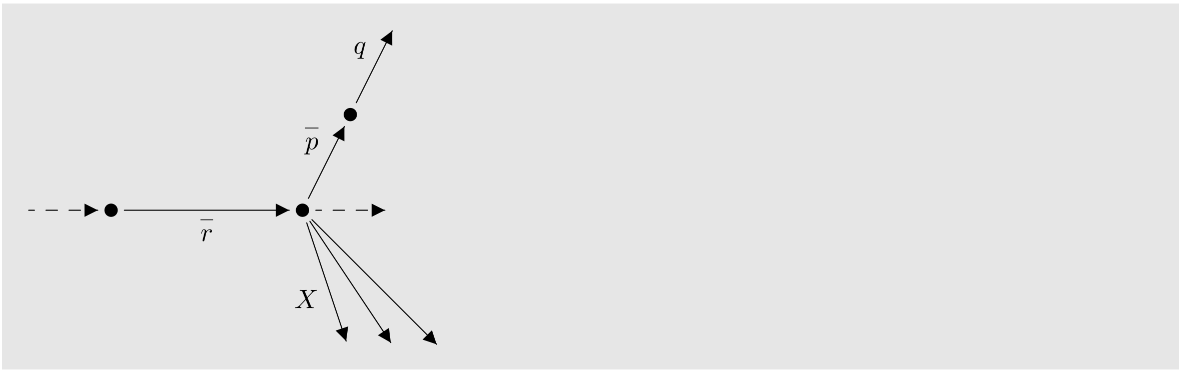
The meaning of $[p]$ is hopefully clear, but what is the meaning of $[\overline{r}, X, \overline{p} : q]$? To answer this question, we need to consider it in the context of a repo.

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

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

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

We can picture a conflictor $\left\lceil \overline{r}, X, \overline{p} : q \right\rfloor$ in a repo as looking like this:



**Definition 12.3 (catch-effect)**
We define $\mathscr{E}$ to tell us the effect that a catch has on the repo:
$\mathscr{E}\left([p]\right) = p$
$\mathscr{E}\left(\left\lceil \overline{r}, X, y \right\rfloor\right) = \overline{r}$
$\mathscr{E}\left(\llbracket \overline{r}, X, y \rrbracket\right) = \overline{r}^{-1}$

**Definition 12.4 (catch-conflicts)**
We define $\mathscr{C}$ to tell us the patches that a catch conflicts with, i.e.:
$\mathscr{C}\left([p]\right) = \emptyset$
$\mathscr{C}\left(\left\lceil \overline{r}, X, y \right\rfloor\right) = N\left(X\right)$
$\mathscr{C}\left(\llbracket \overline{r}, X, y \rrbracket\right) = N\left(X\right)^{-1}$

**Definition 12.5 (catch-names)**
We extend $n$ and $N$ to work on catches in the natural way, i.e.:
$n\left([p]\right) = p$
$n\left(\left\lceil \overline{r}, X, \overline{p} : q \right\rfloor\right) = n\left(q\right)$
$n\left(\llbracket \overline{r}, X, \overline{p} : q \rrbracket\right) = n\left(q\right)^{-1}$
$N\left(\epsilon\right) = \emptyset$
$N\left(c\overline{d}\right) = \{n\left(c\right)\} \cup N\left(\overline{d}\right)$

```
Definition catch_name {pu_type : NameSet → NameSet → Type}
                      {ppu : PartPatchUniverse pu_type pu_type}
                      {pui : PatchUniverseInv ppu ppu}
                      {pu : PatchUniverse pui}
                      {ipl : InvertiblePatchlike pu_type}
                      {ipu : InvertiblePatchUniverse pu ipl}
                      {from to : NameSet}
```

```
                        (c : Catch ipl from to) : SignedName
 := match c with
      | MkCatch p ⇒ pu_nameOf p
      | Conflictor _ _ p _ _ ⇒ contextedPatch_name p
      end.
```

**Definition 12.6 (repo-properties)**
Repos are inductively defined as follows:

$\epsilon$ is a repo.

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

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

$\overline{cde}\,\left[\overline{r}, X, \overline{p} : q\right]$ is a repo if:

- $\overline{cde}$ is a repo
- $\overline{e}$ is a sequence of patches (i.e. there are no conflictors in $\overline{e}$)
- $\mathscr{E}\left(\overline{e}\right) = \overline{r}^{-1}$
- $\mathscr{E}\left(\overline{d}\right) = \overline{p}^{-1}$
- $N\left(\overline{de}\right) = N\,(X)$
- Every catch in $d$ is either a conflictor, or in $\mathscr{C}\,(d)$
- $n\,(y)$ is positive
- $n\,(y) \notin N\left(\overline{cde}\right)$
- There is no catch $f$ and catch sequence $\overline{g}$ such that $\overline{de} = f\overline{g}$ and $f$ does not conflict with $[q]$.
- $\mathscr{E}\,(c)\,q$ is sensible

**Definition 12.7 (catch-inverse)**
We define
$[p]^{-1} = \left[p^{-1}\right]$
$\left[\overline{r}, X, y\right]^{-1} = \left[\!\left[\overline{r}, X, y\right]\!\right]$
$\left[\!\left[\overline{r}, X, y\right]\!\right]^{-1} = \left[\overline{r}, X, y\right]$

```
(*  XXX Put this elsewhere? *)
Definition invertSignedNameSet (s : SignedNameSet)
                                   : SignedNameSet
 := SignedNameSetMod.fold (fun sn s' ⇒ SignedNameSetAdd (signedNameInverse sn) s') s
SignedNameSetMod.empty.

(*
Definition CatchInverse {ipl : InvertiblePatchlike}
                        {from to : NameSet}
                        (c : Catch ipl from to)
                      : Catch ipl to from
```

```
  := match c with
     | MkCatch p => MkCatch (invert _ p)
     | Conflictor _ effect conflicts ident =>
          Conflictor (invertSequence effect)
                     (map (fun cp => addSequenceToContext effect (invertContextedPatch cp)) co
                     (addSequenceToContext effect (invertContextedPatch ident))
     end.
 *)
```

## 12.1 Merge

We now define merge for catches.

**Definition 12.8 (merge)**
We write the merge operator for catches as $+$. It cannot fail: It introduces conflictors instead.
$\forall (\overline{c}d) \in \mathbf{R}, (\overline{c}e) \in \mathbf{R} \cdot$
$(n\left(d\right) = n\left(e\right)) \vee$
$\exists d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$d + e = \langle e', d' \rangle$

**Explanation**
*What we're saying here is that if we have two repos that differ in only their last patch then we can always merge them. The two (equal up to commutation) results of the repo merge are $\overline{c}de'$ and $\overline{c}ed'$.*

We define $+$ thus:

$$\begin{array}{lll}
c & + d & = \langle d', c' \rangle \\
 & & \quad \textbf{if } \left\langle c^{-1}, d \right\rangle \leftrightarrow \left\langle d', c'^{-1} \right\rangle \\
[p] & + [q] & = \left\langle \left[p^{-1}, \{: p\}, : q\right], \left[q^{-1}, \{: q\}, : p\right] \right\rangle \\
[p] & + \left[\overline{r}, X, y\right] & = \left\langle \left[p^{-1}\overline{r}, \left\{\overline{r}^{-1} : p\right\} \cup X, y\right], \left[\epsilon, \{y\}, \overline{r}^{-1} : p\right] \right\rangle \\
\left[\overline{r}, X, y\right] & + [q] & = \left\langle \left[\epsilon, \{y\}, \overline{r}^{-1} : q\right], \left[q^{-1}\overline{r}, \left\{\overline{r}^{-1} : q\right\} \cup X, y\right] \right\rangle \\
\left[\overline{rs}, W, x\right] & + \left[\overline{rt}, Y, z\right] & = \left\langle \left[\overline{t}', \left\{\overline{t}'^{-1}x\right\} \cup (\overline{s}'^{-1}Y), \overline{s}'^{-1}z\right], \left[\overline{s}', \left\{\overline{s}'^{-1}z\right\} \cup (\overline{t}'^{-1}W), \overline{t}'^{-1}x\right] \right\rangle \\
 & & \quad \textbf{if } N\left(\overline{s}\right) \cap N\left(\overline{t}\right) = \emptyset \\
 & & \quad \left\langle \overline{s}^{-1}, \overline{t} \right\rangle \leftrightarrow \left\langle \overline{t}', \overline{s}'^{-1} \right\rangle
\end{array}$$

Note that we haven't yet defined $\leftrightarrow$ on catches, but based on how it works on patches you should have some intuition for what it means.

**Explanation**
*The first rule handles the case where there is no conflict. This works just like patch merging.*

*The second rule handles the case where we have two patches, $p$ and $q$, that conflict. If we have $p$ in our repo and we pull $q$, then we make a conflictor that inverts $p$, conflicts with $p$, and represents $q$.*

*Pulling $p$ into a repo containing $q$ is analogous.*

*The third rule handles the case where we have a patch $[p]$ in one repository and a conflictor, $\left[\overline{r}, X, y\right]$ in the other repository, and they conflict.*

*If we pull the conflictor into the repository containing the patch then the conflictor is the first catch to conflict with $p$, so it must revert it (along with everything it already reverts). It also needs to record that it conflicts with $p$, along with everything that it conflicted with before. And finally, it records that its identity is still $y$.*

*On the other hand, if we pull the patch into a repo containing the conflictor, then the patch turns into a conflictor too. It doesn't need to revert $y$ as it is not the first catch to conflict with it: everything in $X$ has already conflicted with it (XXX lemma that $X$ is not empty). Therefore it has no effect. It does need to record that it conflicts with $y$ (and only $y$). And, of course, it records that its identity is $p$.*

*The fourth rule is the same as the third rule, but with the catches in the opposite order.*

*The fifth rule handles the case where we have a conflictor in each repo, and the conflictors also conflict with each other. The mechanics of the rule are similar to the previous two rules, but we need to do some setup work before we can apply it. The $x$ conflictor is the first in its repo to conflict with everything in $\overline{rs}$, and the $z$ conflictor is the first in its repo to conflict with everything in $\overline{rt}$. But if we pull the $z$ into the repo already containing $y$ then the $z$ conflictor won't be the first catch to conflict with $\overline{r}$, as $x$ already conflicts with it. So we need to commute all common patches in the two conflictor effects to the left. Furthermore, the details of the rule requires that the remaining effects do not conflict with each other. As merging cannot fail, we will have to show that this is the case!*

## 12.2 Commute

We now define commutation of catches.

**Definition 12.9 (catch-commute)**
We extend $\leftrightarrow$ to operate on catches, as defined below.

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

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

**Conflicted merge**

First, if we have a patch, and a conflictor that has only conflicted with that patch, then they swap places:
$\left\langle [p], \left[p^{-1}, \{: p\}, : q\right]\right\rangle \leftrightarrow \left\langle [q], \left[q^{-1}, \{: q\}, : p\right]\right\rangle$

**Explanation**
*This should make sense if you consider the result of merging two patches:*
$[p] + [q] = \left\langle \left[p^{-1}, \{: p\}, : q\right], \left[q^{-1}, \{: q\}, : p\right]\right\rangle$

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

**Explanation**
*Again, this follows from the merge:*
$\left[\overline{r}, X, y\right] + [q] = \left\langle \left[\epsilon, \{y\}, \overline{r}^{-1} : q\right], \left[q^{-1}\overline{r}, \left\{\overline{r}^{-1} : q\right\} \cup X, y\right]\right\rangle$

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

**Explanation**
*Again, this follows from the merge:*
$[p] + \left[\overline{r}, X, y\right] = \left\langle \left[p^{-1}\overline{r}, \left\{\overline{r}^{-1} : p\right\} \cup X, y\right], \left[\epsilon, \{y\}, \overline{r}^{-1} : p\right]\right\rangle$

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

**Explanation**
*XXX Copy the merge rule and rewrite*

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

### Unrelated

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

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

Now commute a conflictor past a patch, where everything goes smoothly:
$\left\langle \left[\overline{r}, X, y\right], [q] \right\rangle \leftrightarrow \left\langle [q'], \left[\overline{r'}, X', y'\right] \right\rangle$ if $\left\langle \overline{r}, q \right\rangle \leftrightarrow \left\langle q', \overline{r'} \right\rangle$
$\left\langle q^{-1}, y \right\rangle \rightarrow \langle y' \rangle$
$\left\langle q^{-1}, X \right\rangle \rightarrow \langle X' \rangle$

**Explanation**
*XXX explanation and pretty pictures will be later.*

Next we have the case where the patch and conflictor start off the other way round:
$\left\langle [p], \left[\overline{r}, X, y\right] \right\rangle \leftrightarrow \left\langle \left[\overline{r'}, X', y'\right], [p'] \right\rangle$ if $\left\langle p, \overline{r} \right\rangle \leftrightarrow \left\langle \overline{r'}, p' \right\rangle$
$\langle p', X \rangle \rightarrow \langle X' \rangle$
$\langle p', y \rangle \rightarrow \langle y' \rangle$

**Explanation**
*XXX ditto*

And the last and most complex of the unrelated cases, commuting a conflictor past another conflictor:
$\left\langle \left[\overline{rs}, W, x\right], \left[\overline{t}, Y, z\right] \right\rangle \leftrightarrow \left\langle \left[\overline{rt'}, \overline{s'}Y, z'\right], \left[\overline{s'}, \overline{t}^{-1}W, x'\right] \right\rangle$
if $N\left(\overline{r}^{-1}\right) \subseteq N(Y)$
$N\left(\overline{s}^{-1}\right) \cap N(Y) = \emptyset$
$\left\langle \overline{s}, \overline{t} \right\rangle \leftrightarrow \left\langle \overline{t'}, \overline{s'} \right\rangle$
$x \leftrightarrows \overline{t}z$
$\forall w \in W \cdot \left(w \leftrightarrows \overline{t}z\right)$
$\forall y \in Y \cdot \left(x \leftrightarrows \overline{t}y\right)$
$\left\langle \overline{t}^{-1}, x \right\rangle \rightarrow \langle x' \rangle$
$\left\langle \overline{s'}, z \right\rangle \rightarrow \langle z' \rangle$

**Explanation**
*XXX ditto*

### Fail

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

**Definition 12.10 (catch-conflicts)**
If $(\bar{c}d) \in \mathbf{R}$ and $(\bar{c}e) \in \mathbf{R}$, We say $d$ *conflicts* with $e$ if and only if $\langle d^{-1}, e\rangle \leftrightarrow$ fail.

**Explanation**
*XXX*

```
(*  XXX Put these things below in the right places: *)
Lemma commute_OneMany_ManyOne
    : ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {from mid1 mid2 to : NameSet}
              (p : pu_type from mid1)
              (qs : Sequence pu mid1 to)
              (qs' : Sequence pu from mid2)
              (p' : pu_type mid2 to),
      «p, qs» <~>om «qs', p'»
   → «qs', p'» <~>mo «p, qs».
Proof.
intros.
dependent induction H.
    constructor.
apply commuteSelfInverse in commutePQ.
apply (ConsManyOneCommute IHOneManyCommute commutePQ).
Qed.

Lemma commute_ManyOne_OneMany
    : ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {from mid1 mid2 to : NameSet}
              (ps : Sequence pu from mid1)
              (q : pu_type mid1 to)
              (q' : pu_type from mid2)
              (ps' : Sequence pu mid2 to),
      «ps, q» <~>mo «q', ps'»
   → «q', ps'» <~>om «ps, q».
Proof.
intros.
dependent induction H.
    constructor.
apply commuteSelfInverse in commutePQ.
apply (ConsOneManyCommute commutePQ IHManyOneCommute).
Qed.

(*
XXX
Inductive AllCommutePast {pu_type : NameSet -> NameSet -> Type}
                          {ppu : PartPatchUniverse pu_type pu_type}
                          {pui : PatchUniverseInv ppu ppu}
                          {pu : PatchUniverse pui}
                          {ipl : InvertiblePatchlike pu}
```

```
                         {from mid : NameSet}
                         (p : pu_type from mid)
                       : list (ContextedPatch pu mid)
                      -> list (ContextedPatch pu from)
                      -> Prop
    := AllCommutePastNil :
            AllCommutePast p nil nil
     | AllCommutePastCons :
            forall {cp : ContextedPatch pu mid}
                   {cps : list (ContextedPatch pu mid)}
                   {cp' : ContextedPatch pu from}
                   {cps' : list (ContextedPatch pu from)},
            CommutePast p cp cp' ->
            AllCommutePast p cps cps' ->
            AllCommutePast p (cons cp cps) (cons cp' cps').
*)

(*
(*  XXX Need to define this properly *)
Parameter sequenceCommuteInverse
    : forall {pu_type : NameSet -> NameSet -> Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike ppu}
             {from mid mid' to : NameSet}
             {psContains qsContains ps'Contains qs'Contains : SignedNameSet}
             (ps  : Sequence pu from mid )
             (qs  : Sequence pu mid  to  )
             (qs' : Sequence pu from mid')
             (ps' : Sequence pu mid' to  ),
      «ps, qs» <~> «qs', ps'»
   -> «qs', ps'» <~> «ps, qs».
*)

(*  XXX Put this somewhere else? *)
```
Lemma *CommuteOutNameInSequence* $\{pu\_type : NameSet \rightarrow NameSet \rightarrow$ Type$\}$
$\{ppu : PartPatchUniverse\ pu\_type\ pu\_type\}$
$\{pui : PatchUniverseInv\ ppu\ ppu\}$
$\{pu : PatchUniverse\ pui\}$
$\{from\ mid\ to : NameSet\}$
$\{p : pu\_type\ from\ mid\}$
$\{qs : Sequence\ pu\ mid\ to\}$
$\{rs : Sequence\ pu\ from\ to\}$
$(commuteOutLeft : «p,\ qs» <\tilde{}\ «rs»)$
$: SignedNameSetIn\ (pu\_nameOf\ p)\ (sequenceContents\ rs)$.

Proof with auto.
induction *commuteOutLeft*.
    rewrite *SequenceContentsCons*.
    *signedNameSetDec*.
destruct (*commuteNames H*) as [? [? ?]].
rewrite ← *H1*.
rewrite *SequenceContentsCons*.

```
(*  coq bug 2699 *)
remember (sequenceContents rs) as X.
signedNameSetDec.
Qed.

(*  XXX Put this somewhere else? *)
Lemma TransitiveCommuteRelatesCons {pu_type : NameSet → NameSet → Type}
                                   {ppu : PartPatchUniverse pu_type pu_type}
                                   {pui : PatchUniverseInv ppu ppu}
                                   {pu : PatchUniverse pui}
                                   {from mid to : NameSet}
                                   {p : pu_type from mid}
                                   {qs : Sequence pu mid to}
                                   {rs : Sequence pu mid to}
                                   {pQs : ConsOK p qs}
                                   {pRs : ConsOK p rs}
                                   (trans : «p :> qs» <~~>* «p :> rs»)
                                 : «qs» <~~>* «rs».
Proof with auto.
destruct (commuteOutLeftConsistent trans (commuteOutLeftDone p qs _)) as [rs' [X2 X3]].
dependent induction X3.
    consEquality x...
consEquality x...
set (X := CommuteOutNameInSequence X3).
clearbody X.
destruct pRs.
elim consNotAlreadyThere0.
destruct (commuteNames H) as [? [? ?]].
rewrite ← H1...
Qed.

(*  XXX Put this elsewhere? *)
Fixpoint conflictsNames {pu_type : NameSet → NameSet → Type}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {ipl : InvertiblePatchlike pu_type}
                        {from : NameSet}
                        (conflicts : list (ContextedPatch pu from))
                  : SignedNameSet
    := match conflicts with
       | nil ⇒ SignedNameSetMod.empty
       | cp :: conflicts' ⇒ SignedNameSetMod.add (contextedPatch_name cp) (conflictsNames
conflicts')
       end.

(*  XXX Move this elsewhere *)
Program Fixpoint noneCommutePastBase
            {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {pu : PatchUniverse pui}
            {from mid to : NameSet}
            (x : SequenceBase pu from mid)
            (y : SequenceBase pu mid to)
            (appendOK : SignedNameSetMod.Empty
                            (SignedNameSetMod.inter
                                (sequenceBaseContents x) (sequenceBaseContents y)))
            (xNoDupes : SequenceNoDupes x)
            (yNoDupes : SequenceNoDupes y)
          : Prop
    := match x with
        | Nil _ ⇒ True
        | Cons _ _ _ p ps ⇒
              ~(commutable p (MkSequence (appendBase ps y) _))
              ∧ noneCommutePastBase ps y _ _ _
        end.
Next Obligation.
simpl.
subst.
constructor.
rewrite SequenceContentsBaseCons in appendOK.
rewrite SequenceNoDupesCons in xNoDupes.
destruct xNoDupes as [? xNoDupes].
remember (pu_nameOf p) as pName.
clear HeqpName.
dependent induction ps generalizing from p.
    rewrite AppendBaseNil.
    auto.
rewrite SequenceNoDupesCons in xNoDupes.
destruct xNoDupes as [? xNoDupes].
rewrite AppendBaseCons.
unfold SequenceNoDupes.
fold (SequenceNoDupes (from := mid) (to := to)).
split.
    rewrite SequenceContentsBaseAppend.
    rewrite SequenceContentsBaseCons in appendOK.
    try signedNameSetDec. (*  XXX coq bug *)
    admit.
specialize (IHps y).
specialize (IHps yNoDupes).
specialize (IHps pName).
solveFirstIn IHps.
    rewrite SequenceContentsBaseCons in appendOK.
    clear - appendOK.
    try signedNameSetDec. (*  XXX coq bug *)
    admit.
solveFirstIn IHps.
    rewrite SequenceContentsBaseCons in H.
    try signedNameSetDec. (*  XXX coq bug *)
    admit.
specialize (IHps xNoDupes).
specialize (IHps from).
specialize (IHps p).
auto.
Qed.
```

```
Next Obligation.
simpl.
subst.
rewrite SequenceContentsBaseCons in appendOK.
try signedNameSetDec. (*  coq bug 2699 *)
remember (sequenceBaseContents ps) as psC.
remember (sequenceBaseContents y) as yC.
signedNameSetDec.
Qed.
Next Obligation.
subst.
rewrite SequenceNoDupesCons in xNoDupes.
destruct xNoDupes.
auto.
Qed.

(*  XXX Move this elsewhere *)
Program Definition noneCommutePast
              {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {from mid to : NameSet}
              (x : Sequence pu from mid)
              (y : Sequence pu mid to)
              (appendOK : AppendOK x y)
           : Prop
    := match x with
        MkSequence xs _ ⇒
             match y with
             MkSequence ys _ ⇒
                  noneCommutePastBase xs ys _ _ _
             end
        end.
Next Obligation.
destruct appendOK.
simpl in appendNoIntersection0.
auto.
Qed.
Next Obligation.
destruct wildcard'.
auto.
Qed.
Next Obligation.
destruct wildcard'0.
auto.
Qed.

(*  XXX Move this elsewhere *)
Lemma uncommutableSame
                 {pu_type : NameSet → NameSet → Type}
                 {ppu : PartPatchUniverse pu_type pu_type}
                 {pui : PatchUniverseInv ppu ppu}
```

```
                {pu : PatchUniverse pui}
                {from mid to : NameSet}
                (xs : Sequence pu from mid)
                (ys : Sequence pu mid to)

                {midsplit1 midsplit2 mid1' mid2' : NameSet}
                {ns1 : Sequence pu from midsplit1}
                {ns2 : Sequence pu from midsplit2}
                {cs1 : Sequence pu midsplit1 mid}
                {cs2 : Sequence pu midsplit2 mid}
                {ys1 : Sequence pu midsplit1 mid1'}
                {ys2 : Sequence pu midsplit2 mid2'}
                {cs1' : Sequence pu mid1' to}
                {cs2' : Sequence pu mid2' to}
                {appendOKNs1Cs1 : AppendOK ns1 cs1}
                {appendOKNs2Cs2 : AppendOK ns2 cs2}
                (split1 : «xs» <~~>* «ns1 :+> cs1»)
                (split2 : «xs» <~~>* «ns2 :+> cs2»)
                (commute1 : «cs1, ys» <~> «ys1, cs1'»)
                (commute2 : «cs2, ys» <~> «ys2, cs2'»)
                {appendOKNs1Ys1 : AppendOK ns1 ys1}
                {appendOKNs2Ys2 : AppendOK ns2 ys2}
                (ns1NonCommute : noneCommutePast ns1 ys1 _)
                (ns2NonCommute : noneCommutePast ns2 ys2 _)
        : midsplit1 = midsplit2 ∧ ns1 ~= ns2 ∧ cs1 ~= cs2.
Proof.
admit.
Qed.
(*  p p^, {:p}, :q <-> q q^, {:q}, :p *)
Inductive CatchCommute1 {pu_type : NameSet → NameSet → Type}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {ipl : InvertiblePatchlike pu_type}
                        {ipu : InvertiblePatchUniverse pu ipl}
         : ∀ {from mid1 mid2 to : NameSet},
           Catch ipl from mid1
         → Catch ipl mid1 to
         → Catch ipl from mid2
         → Catch ipl mid2 to
         → Prop
    := MkCatchCommute1 :
           ∀ {from to1 to2 : NameSet}
                  (p : pu_type from to1)
                  (q : pu_type from to2)
                  (namesDifferent : pu_nameOf p ≠ pu_nameOf q)
                  (do_not_commute : ~(commutable (invert q) p))
                  (invPConsOK : ConsOK (invert p) [])
                  (invQConsOK : ConsOK (invert q) [])
                  (contextedP := MkContextedPatch [] p)
                  (contextedQ := MkContextedPatch [] q)
                  (conflictsP := cons contextedP nil)
```

```
                    (conflictsQ := cons contextedQ nil)
                    (q'Effect := invert p :> [])
                    (p'Effect := invert q :> [])
                    (q'EffectCanonicallyOrdered : CanonicallyOrdered q'Effect)
                    (p'EffectCanonicallyOrdered : CanonicallyOrdered p'Effect)
                    (*  XXX Want to deduce these from do_not_commute really *)
                    (qConflictsP : Forall (conflictsWith contextedQ) conflictsP)
                    (pConflictsQ : Forall (conflictsWith contextedP) conflictsQ)
                    ,
             CatchCommute1 (MkCatch p)
                           (Conflictor q'Effect
                                       conflictsP
                                       contextedQ
                                       qConflictsP
                                       q'EffectCanonicallyOrdered)
                           (MkCatch q)
                           (Conflictor p'Effect
                                       conflictsQ
                                       contextedP
                                       pConflictsQ
                                       p'EffectCanonicallyOrdered).
Notation "« p , q » <~>1 « q' , p' »"
    := (CatchCommute1 p q q' p')
    (at level 60, no associativity).
Inductive CatchCommute2 {pu_type : NameSet → NameSet → Type}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {ipl : InvertiblePatchlike pu_type}
                        {ipu : InvertiblePatchUniverse pu ipl}
        : ∀ {from mid1 mid2 to : NameSet},
           Catch ipl from mid1
         → Catch ipl mid1 to
         → Catch ipl from mid2
         → Catch ipl mid2 to
         → Prop
      := MkCatchCommute2 :
             (*  p p^ r, {r^:p} U X, y <-> r, X, y  , {y}, r^:p *)
             ∀ {from mid to : NameSet}
                    (p : pu_type from mid)
                    (qEffect : Sequence pu mid to)
                    (qEffectCanonicallyOrdered : CanonicallyOrdered qEffect)
                    (qConflicts : list (ContextedPatch pu to))
                    (qIdentity : ContextedPatch pu to)
                    (qEffect' : Sequence pu from to)
                    (qEffect'CanonicallyOrdered : CanonicallyOrdered qEffect')
                    (qConflicts' : list (ContextedPatch pu to))
                    (qIdentity' : ContextedPatch pu to)
                    (pEffect' : Sequence pu to to)
                    (pEffect'CanonicallyOrdered : CanonicallyOrdered pEffect')
                    (pConflicts' : list (ContextedPatch pu to))
                    (pIdentity' : ContextedPatch pu to)
                    (namesDifferent : pu_nameOf p ≠ contextedPatch_name qIdentity)
                    (invPRsConsOK : ConsOK (invert p) qEffect')
                    (invQEffect'pOK : ContextedPatchOK (invert qEffect') p)
                    (qConflictsConflict : Forall (conflictsWith qIdentity) qConflicts)
                    (q'ConflictsConflict : Forall (conflictsWith qIdentity') qConflicts')
                    (p'ConflictsConflict : Forall (conflictsWith pIdentity') pConflicts'),
              «qEffect» <~~>* «invert p :> qEffect'»
           → qConflicts = (MkContextedPatch (invert qEffect') p) :: qConflicts'
           → qIdentity' = qIdentity
           → pEffect' = []
           → pConflicts' = qIdentity :: nil
           → pIdentity' = MkContextedPatch (invert qEffect') p
           → ~(qConflicts' = nil)
           → CatchCommute2 (MkCatch p)
                           (Conflictor qEffect
                                       qConflicts
                                       qIdentity
                                       qConflictsConflict
                                       qEffectCanonicallyOrdered)
                           (Conflictor qEffect'
                                       qConflicts'
                                       qIdentity'
                                       q'ConflictsConflict
                                       qEffect'CanonicallyOrdered)
                           (Conflictor pEffect'
                                       pConflicts'
                                       pIdentity'
                                       p'ConflictsConflict
                                       pEffect'CanonicallyOrdered).
Notation "« p , q » <~>2 « q' , p' »"
    := (CatchCommute2 p q q' p')
    (at level 60, no associativity).
Inductive CatchCommute3 {pu_type : NameSet → NameSet → Type}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {ipl : InvertiblePatchlike pu_type}
                        {ipu : InvertiblePatchUniverse pu ipl}
          : ∀ {from mid1 mid2 to : NameSet},
            Catch ipl from mid1
          → Catch ipl mid1 to
          → Catch ipl from mid2
          → Catch ipl mid2 to
          → Prop
      := MkCatchCommute3 :
             (*  r, X, y  , {y}, r^:q <-> q q^ r, {r^:q} U X, y *)
             ∀ {from mid to : NameSet}
                    (pEffect : Sequence pu from to)
                    (pEffectCanonicallyOrdered : CanonicallyOrdered pEffect)
                    (pConflicts : list (ContextedPatch pu to))
                    (pIdentity : ContextedPatch pu to)
                    (qEffect : Sequence pu to to)
```

```
                                (qEffectCanonicallyOrdered : CanonicallyOrdered qEffect)
                                (qConflicts : list (ContextedPatch pu to))
                                (qIdentity : ContextedPatch pu to)
                                (q' : pu_type from mid)
                                (pEffect' : Sequence pu mid to)
                                (pEffect'CanonicallyOrdered : CanonicallyOrdered pEffect')
                                (pConflicts' : list (ContextedPatch pu to))
                                (pIdentity' : ContextedPatch pu to)
                                (namesDifferent : contextedPatch_name pIdentity ≠ contextedPatch_name
qIdentity)
                                (invPRsConsOK : ConsOK (invert q') pEffect)
                                (invPEffectq'OK : ContextedPatchOK (invert pEffect) q')
                                (pConflictsConflict : Forall (conflictsWith pIdentity) pConflicts)
                                (qConflictsConflict : Forall (conflictsWith qIdentity) qConflicts)
                                (p'ConflictsConflict : Forall (conflictsWith pIdentity') pConflicts'),
                  qEffect = []
             → qConflicts = pIdentity :: nil
             → qIdentity = MkContextedPatch (invert pEffect) q'
             → «pEffect'» <~~>* «invert q' :> pEffect»
             → pConflicts' = (MkContextedPatch (invert pEffect) q') :: pConflicts
             → pIdentity' = pIdentity
             → ~(pConflicts = nil)
             → CatchCommute3 (Conflictor pEffect
                                          pConflicts
                                          pIdentity
                                          pConflictsConflict
                                          pEffectCanonicallyOrdered)
                             (Conflictor qEffect
                                          qConflicts
                                          qIdentity
                                          qConflictsConflict
                                          qEffectCanonicallyOrdered)
                             (MkCatch q')
                             (Conflictor pEffect'
                                          pConflicts'
                                          pIdentity'
                                          p'ConflictsConflict
                                          pEffect'CanonicallyOrdered).
Notation "« p , q » <~>3 « q' , p' »"
    := (CatchCommute3 p q q' p')
    (at level 60, no associativity).
Inductive CatchCommute4 {pu_type : NameSet → NameSet → Type}
                         {ppu : PartPatchUniverse pu_type pu_type}
                         {pui : PatchUniverseInv ppu ppu}
                         {pu : PatchUniverse pui}
                         {ipl : InvertiblePatchlike pu_type}
                         {ipu : InvertiblePatchUniverse pu ipl}
          : ∀ {from mid1 mid2 to : NameSet},
             Catch ipl from mid1
          → Catch ipl mid1 to
          → Catch ipl from mid2
          → Catch ipl mid2 to
```

```
        → Prop
      := MkCatchCommute4 :
(*
            (*  r s, W, x t, {t^x} U Y, z <-> r t', s'Y, s'z s', z U t^W,
t^x *)
*)
              ∀ {o or ors orst ort : NameSet}

                      (pEffect : Sequence pu o ors)
                      (pEffectCanonicallyOrdered : CanonicallyOrdered pEffect)
                      (pConflicts : list (ContextedPatch pu ors))
                      (pIdentity : ContextedPatch pu ors)
                      (qEffect : Sequence pu ors orst)
                      (qEffectCanonicallyOrdered : CanonicallyOrdered qEffect)
                      (qConflicts : list (ContextedPatch pu orst))
                      (qOtherConflicts : list (ContextedPatch pu orst))
                      (qIdentity : ContextedPatch pu orst)
                      (qEffect' : Sequence pu o ort)
                      (qEffect'CanonicallyOrdered : CanonicallyOrdered qEffect')
                      (qConflicts' : list (ContextedPatch pu ort))
                      (qIdentity' : ContextedPatch pu ort)
                      (pEffect' : Sequence pu ort orst)
                      (pEffect'CanonicallyOrdered : CanonicallyOrdered pEffect')
                      (pConflicts' : list (ContextedPatch pu orst))
                      (pIdentity' : ContextedPatch pu orst)
                      (commonEffect : Sequence pu o or)
                      (pOnlyEffect : Sequence pu or ors)
                      (pEffectCanonicallyOrdered : CanonicallyOrdered pEffect)
                      (qOnlyEffect : Sequence pu or ort)
                      (pEffectCanonicallyOrdered : CanonicallyOrdered pEffect)

                      (namesDifferent : contextedPatch_name pIdentity ≠ contextedPatch_name
qIdentity)

                      (appendOK_commonEffect_pOnlyEffect : AppendOK commonEffect pOnly-
Effect)
                      (appendOK_commonEffect_qOnlyEffect : AppendOK commonEffect qOnly-
Effect)

                      (pConflictsConflict : Forall (conflictsWith pIdentity) pConflicts)
                      (qConflictsConflict : Forall (conflictsWith qIdentity) qConflicts)
                      (q'ConflictsConflict : Forall (conflictsWith qIdentity') qConflicts')
                      (p'ConflictsConflict : Forall (conflictsWith pIdentity') pConflicts')

                      (noneCommutePastCommonEffectPOnlyEffect : noneCommutePast com-
monEffect pOnlyEffect appendOK_commonEffect_pOnlyEffect)
                      (noneCommutePastCommonEffectQOnlyEffect : noneCommutePast com-
monEffect qOnlyEffect appendOK_commonEffect_qOnlyEffect),

              «pEffect» <~~>* «commonEffect :+> pOnlyEffect»
          → qConflicts = addSequenceToContext (invert qEffect) pIdentity :: qOtherConflicts
          → «qEffect'» <~~>* «commonEffect :+> qOnlyEffect»
          → qConflicts' = map (addSequenceToContext pEffect') qOtherConflicts
          → qIdentity' = addSequenceToContext pEffect' qIdentity
          → pConflicts' = qIdentity :: map (addSequenceToContext (invert qEffect)) pConflicts
          → pIdentity' = addSequenceToContext (invert qEffect) pIdentity
          → «pOnlyEffect, qEffect» <~> «qOnlyEffect, pEffect'»
          → ~(pConflicts = nil)
          → ~(qConflicts' = nil)
          → CatchCommute4 (Conflictor pEffect
                                      pConflicts
                                      pIdentity
                                      pConflictsConflict
                                      pEffectCanonicallyOrdered)
                          (Conflictor qEffect
                                      qConflicts
                                      qIdentity
                                      qConflictsConflict
                                      qEffectCanonicallyOrdered)
                          (Conflictor qEffect'
                                      qConflicts'
                                      qIdentity'
                                      q'ConflictsConflict
                                      qEffect'CanonicallyOrdered)
                          (Conflictor pEffect'
                                      pConflicts'
                                      pIdentity'
                                      p'ConflictsConflict
                                      pEffect'CanonicallyOrdered).
Notation "« p , q » <~>4 « q' , p' »"
    := (CatchCommute4 p q q' p')
    (at level 60, no associativity).
Inductive CatchCommute5 {pu_type : NameSet → NameSet → Type}
                          {ppu : PartPatchUniverse pu_type pu_type}
                          {pui : PatchUniverseInv ppu ppu}
                          {pu : PatchUniverse pui}
                          {ipl : InvertiblePatchlike pu_type}
                          {ipu : InvertiblePatchUniverse pu ipl}
          : ∀ {from mid1 mid2 to : NameSet},
            Catch ipl from mid1
        → Catch ipl mid1 to
        → Catch ipl from mid2
        → Catch ipl mid2 to
        → Prop
      := MkCatchCommute5 :
              (*  Successful commute: patch/patch *)
              ∀ {from mid1 mid2 to : NameSet}
                      (p : pu_type from mid1)
                      (q : pu_type mid1 to)
                      (q' : pu_type from mid2)
                      (p' : pu_type mid2 to),
              «p, q» <~> «q', p'» →
              CatchCommute5 (MkCatch p) (MkCatch q) (MkCatch q') (MkCatch p').
Notation "« p , q » <~>5 « q' , p' »"
```

```
    := (CatchCommute5 p q q' p')
    (at level 60, no associativity).
(*  XXX The ones below here should check there isn't already a conflict *)
Inductive CatchCommute6 {pu_type : NameSet → NameSet → Type}
                        {ppu : PartPatchUniverse pu_type pu_type}
                        {pui : PatchUniverseInv ppu ppu}
                        {pu : PatchUniverse pui}
                        {ipl : InvertiblePatchlike pu_type}
                        {ipu : InvertiblePatchUniverse pu ipl}
        : ∀ {from mid1 mid2 to : NameSet},
          Catch ipl from mid1
        → Catch ipl mid1 to
        → Catch ipl from mid2
        → Catch ipl mid2 to
        → Prop
    := MkCatchCommute6 :
           (*  Successful commute: patch/conflictor *)
           ∀ {from mid1 mid2 to : NameSet}
                   (p : pu_type from mid1)
                   (qEffect : Sequence pu mid1 to)
                   (qEffectCanonicallyOrdered : CanonicallyOrdered qEffect)
                   (qConflicts : list (ContextedPatch pu to))
                   (qIdentity : ContextedPatch pu to)
                   (qEffect' : Sequence pu from mid2)
                   (qEffect'CanonicallyOrdered : CanonicallyOrdered qEffect')
                   (qConflicts' : list (ContextedPatch pu mid2))
                   (qIdentity' : ContextedPatch pu mid2)
                   (p' : pu_type mid2 to)
                   (namesDifferent : pu_nameOf p ≠ contextedPatch_name qIdentity)
                   (noConflict : ¬SignedNameSetIn (pu_nameOf p) (conflictsNames qCon-
flicts))
                   (qConflictsConflict : Forall (conflictsWith qIdentity) qConflicts)
                   (q'ConflictsConflict : Forall (conflictsWith qIdentity') qConflicts'),
           «p, qEffect» <~>om «qEffect', p'» →
           CommutePast p' qIdentity qIdentity' →
           qConflicts' = map (addToContext p') qConflicts →
           ~(qConflicts = nil) →
           CatchCommute6 (MkCatch p)
                         (Conflictor qEffect
                                     qConflicts
                                     qIdentity
                                     qConflictsConflict
                                     qEffectCanonicallyOrdered)
                         (Conflictor qEffect'
                                     qConflicts'
                                     qIdentity'
                                     q'ConflictsConflict
                                     qEffect'CanonicallyOrdered)
                         (MkCatch p').
Notation "« p , q » <~>6 « q' , p' »"
    := (CatchCommute6 p q q' p')
    (at level 60, no associativity).
```

```
Inductive CatchCommute7 {pu_type : NameSet → NameSet → Type}
                            {ppu : PartPatchUniverse pu_type pu_type}
                            {pui : PatchUniverseInv ppu ppu}
                            {pu : PatchUniverse pui}
                            {ipl : InvertiblePatchlike pu_type}
                            {ipu : InvertiblePatchUniverse pu ipl}
         : ∀ {from mid1 mid2 to : NameSet},
           Catch ipl from mid1
        → Catch ipl mid1 to
        → Catch ipl from mid2
        → Catch ipl mid2 to
        → Prop
    := MkCatchCommute7 :
           (*  Successful commute: conflictor/patch *)
           ∀ {from mid1 mid2 to : NameSet}
                   (pEffect : Sequence pu from mid1)
                   (pEffectCanonicallyOrdered : CanonicallyOrdered pEffect)
                   (pConflicts : list (ContextedPatch pu mid1))
                   (pIdentity : ContextedPatch pu mid1)
                   (q : pu_type mid1 to)
                   (q' : pu_type from mid2)
                   (pEffect' : Sequence pu mid2 to)
                   (pEffect'CanonicallyOrdered : CanonicallyOrdered pEffect')
                   (pConflicts' : list (ContextedPatch pu to))
                   (pIdentity' : ContextedPatch pu to)
                   (namesDifferent : contextedPatch_name pIdentity ≠ pu_nameOf q)
                   (pConflictsConflict : Forall (conflictsWith pIdentity) pConflicts)
                   (p'ConflictsConflict : Forall (conflictsWith pIdentity') pConflicts'),
           «pEffect, q» <~>mo «q', pEffect'» →
           CommutePast (invert q) pIdentity pIdentity' →
           pConflicts' = map (addToContext (invert q)) pConflicts →
           ~(pConflicts = nil) →
           CatchCommute7 (Conflictor pEffect
                                        pConflicts
                                        pIdentity
                                        pConflictsConflict
                                        pEffectCanonicallyOrdered)
                            (MkCatch q)
                            (MkCatch q')
                            (Conflictor pEffect'
                                        pConflicts'
                                        pIdentity'
                                        p'ConflictsConflict
                                        pEffect'CanonicallyOrdered).
Notation "« p , q » <~>7 « q' , p' »"
    := (CatchCommute7 p q q' p')
    (at level 60, no associativity).
Inductive CatchCommute8 {pu_type : NameSet → NameSet → Type}
                            {ppu : PartPatchUniverse pu_type pu_type}
                            {pui : PatchUniverseInv ppu ppu}
                            {pu : PatchUniverse pui}
                            {ipl : InvertiblePatchlike pu_type}
```

```
                              {ipu : InvertiblePatchUniverse pu ipl}
          : ∀ {from mid1 mid2 to : NameSet},
             Catch ipl from mid1
          → Catch ipl mid1 to
          → Catch ipl from mid2
          → Catch ipl mid2 to
          → Prop
       := MkCatchCommute8 :
              (*  Successful commute: conflictor/conflictor *)
              ∀ {from common mid1 mid2 to : NameSet}
                      (pEffect : Sequence pu from mid1)
                      (pEffectCanonicallyOrdered : CanonicallyOrdered pEffect)
                      (pConflicts : list (ContextedPatch pu mid1))
                      (pIdentity : ContextedPatch pu mid1)
                      (qEffect : Sequence pu mid1 to)
                      (qEffectCanonicallyOrdered : CanonicallyOrdered qEffect)
                      (qConflicts : list (ContextedPatch pu to))
                      (qIdentity : ContextedPatch pu to)
                      (qEffect' : Sequence pu from mid2)
                      (qEffect'CanonicallyOrdered : CanonicallyOrdered qEffect')
                      (qConflicts' : list (ContextedPatch pu mid2))
                      (qIdentity' : ContextedPatch pu mid2)
                      (pEffect' : Sequence pu mid2 to)
                      (pEffect'CanonicallyOrdered : CanonicallyOrdered pEffect')
                      (pConflicts' : list (ContextedPatch pu to))
                      (pIdentity' : ContextedPatch pu to)
                      (commonEffect : Sequence pu from common)
                      (commonEffectCanonicallyOrdered : CanonicallyOrdered commonEffect)
                      (pOnlyEffect : Sequence pu common mid1)
                      (pOnlyEffectCanonicallyOrdered : CanonicallyOrdered pOnlyEffect)
                      (qOnlyEffect : Sequence pu common mid2)
                      (qOnlyEffectCanonicallyOrdered : CanonicallyOrdered qOnlyEffect)
                      (appendOK_commonEffect_pOnlyEffect : AppendOK commonEffect pOnly-
Effect)
                      (appendOK_commonEffect_qOnlyEffect : AppendOK commonEffect qOnly-
Effect)
                      (namesDifferent : contextedPatch_name pIdentity ≠ contextedPatch_name
qIdentity)
                      (pConflictsConflict : Forall (conflictsWith pIdentity) pConflicts)
                      (qConflictsConflict : Forall (conflictsWith qIdentity) qConflicts)
                      (q'ConflictsConflict : Forall (conflictsWith qIdentity') qConflicts')
                      (p'ConflictsConflict : Forall (conflictsWith pIdentity') pConflicts'),

              (*  Effects *)
(*
                            pEffect                                      qEffect
    <from> commonEffect <common> pOnlyEffect <mid1>       <mid1>qEffect<to>

                            qEffect'                                     pEffect'
    <from> commonEffect <common> qOnlyEffect <mid2>       <mid2>pEffect'<to>
*)
              «pOnlyEffect, qEffect» <~> «qOnlyEffect, pEffect'» →
```

```
      «pEffect» <~~>* «commonEffect :+> pOnlyEffect» →
      «qEffect'» <~~>* «commonEffect :+> qOnlyEffect» →
      SignedNameSetMod.Empty
          (SignedNameSetMod.inter
              (sequenceContents pOnlyEffect)
              (sequenceContents qOnlyEffect)) →

      (* Identities *)
      CommuteManyPast (invert qEffect) pIdentity pIdentity' →
      CommuteManyPast pEffect' qIdentity qIdentity' →

      (* Conflicts *)
      pConflicts' = map (addSequenceToContext (invert qEffect)) pConflicts →
      qConflicts' = map (addSequenceToContext pEffect') qConflicts →

      (* The patches themselves can't conflict *)
      ~(conflictsWith pIdentity (addSequenceToContext qEffect qIdentity)) →

      ~(pConflicts = nil) →
      ~(qConflicts = nil) →

      CatchCommute8 (Conflictor pEffect
                                pConflicts
                                pIdentity
                                pConflictsConflict
                                pEffectCanonicallyOrdered)
                    (Conflictor qEffect
                                qConflicts
                                qIdentity
                                qConflictsConflict
                                qEffectCanonicallyOrdered)
                    (Conflictor qEffect'
                                qConflicts'
                                qIdentity'
                                q'ConflictsConflict
                                qEffect'CanonicallyOrdered)
                    (Conflictor pEffect'
                                pConflicts'
                                pIdentity'
                                p'ConflictsConflict
                                pEffect'CanonicallyOrdered).
Notation "« p , q » <~>8 « q' , p' »"
    := (CatchCommute8 p q q' p')
    (at level 60, no associativity).

(* XXX Temporary? *)
Inductive commuteNum : nat → Prop
    := mkCommuteNum : ∀ (myCommuteNum : nat), commuteNum myCommuteNum.

Inductive CatchCommute {pu_type : NameSet → NameSet → Type}
                       {ppu : PartPatchUniverse pu_type pu_type}
                       {pui : PatchUniverseInv ppu ppu}
                       {pu : PatchUniverse pui}
```

```
                              {ipl : InvertiblePatchlike pu_type}
                              {ipu : InvertiblePatchUniverse pu ipl}
                              {from mid1 mid2 to : NameSet}
                              (p : Catch ipl from mid1)
                              (q : Catch ipl mid1 to)
                              (q' : Catch ipl from mid2)
                              (p' : Catch ipl mid2 to)
                            : Prop
         (*  p p^, {:p}, :q <-> q q^, {:q}, :p *)
      := isCatchCommute1 : ∀ (myCommuteNum : commuteNum 1)
                                    (catchCommuteDetails : CatchCommute1 p q q' p'),
                               CatchCommute p q q' p'
            (*  p p^ r, {r^:p} U X, y <-> r, X, y  , {y}, r^:p *)
       | isCatchCommute2 : ∀ (myCommuteNum : commuteNum 2)
                                    (catchCommuteDetails : CatchCommute2 p q q' p'),
                               CatchCommute p q q' p'
            (*  r, X, y  , {y}, r^:q <-> q q^ r, {r^:q} U X, y *)
       | isCatchCommute3 : ∀ (myCommuteNum : commuteNum 3)
                                    (catchCommuteDetails : CatchCommute3 p q q' p'),
                               CatchCommute p q q' p'
            (*  r s, W, x t, {t^x} U Y, z <-> r t', s'Y, s'z s', z U t^W,
t^x *)
       | isCatchCommute4 : ∀ (myCommuteNum : commuteNum 4)
                                    (catchCommuteDetails : CatchCommute4 p q q' p'),
                               CatchCommute p q q' p'
            (*  Successful commute: patch/patch *)
       | isCatchCommute5 : ∀ (myCommuteNum : commuteNum 5)
                                    (catchCommuteDetails : CatchCommute5 p q q' p'),
                               CatchCommute p q q' p'
            (*  Successful commute: patch/conflictor *)
       | isCatchCommute6 : ∀ (myCommuteNum : commuteNum 6)
                                    (catchCommuteDetails : CatchCommute6 p q q' p'),
                               CatchCommute p q q' p'
            (*  Successful commute: conflictor/patch *)
       | isCatchCommute7 : ∀ (myCommuteNum : commuteNum 7)
                                    (catchCommuteDetails : CatchCommute7 p q q' p'),
                               CatchCommute p q q' p'
            (*  Successful commute: conflictor/conflictor *)
       | isCatchCommute8 : ∀ (myCommuteNum : commuteNum 8)
                                    (catchCommuteDetails : CatchCommute8 p q q' p'),
                               CatchCommute p q q' p'.
Notation "« p , q » <~>c « q' , p' »"
    := (CatchCommute p q q' p')
    (at level 60, no associativity).

(*  XXX Move this lemma to contexted_patches: *)
Lemma nameOfAddSequenceToContext :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {from to : NameSet}
```

```
                {ps : Sequence pu from to}
                {cp : ContextedPatch pu to},
        contextedPatch_name (addSequenceToContext ps cp) = contextedPatch_name cp.
Proof with auto.
intros.
(*
XXX
induction ps.
    admit.
*)
admit.
Qed.

(*  XXX *)
Lemma addSequenceToContextInverse1 :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {ipu : InvertiblePatchUniverse pu ipl}
                {from to : NameSet}
                {ps : Sequence pu from to}
                {cp : ContextedPatch pu from},
        addSequenceToContext ps
            (addSequenceToContext (invert ps) cp) = cp.
Proof with auto.
(*
XXX
intros.
induction ps.
    admit.
*)
admit.
Qed.

Lemma addSequenceToContextInverse2 :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {ipu : InvertiblePatchUniverse pu ipl}
                {from to : NameSet}
                {ps : Sequence pu from to}
                {cp : ContextedPatch pu to},
        addSequenceToContext (invert ps)
            (addSequenceToContext ps cp) = cp.
Proof with auto.
intros.
(*
XXX
induction ps.
```

```
    admit.
*)
admit.
Qed.

Lemma addSequenceToContextIdentity1 :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {ipu : InvertiblePatchUniverse pu ipl}
              {from to : NameSet}
              {ps : Sequence pu from to},
      ∀ (cp : ContextedPatch pu from),
      (fun x ⇒
           addSequenceToContext ps
               (addSequenceToContext (invert ps) x)) cp
    = (fun x ⇒ x) cp.
Proof with auto.
intros.
simpl.
apply addSequenceToContextInverse1...
Qed.

Lemma addSequenceToContextIdentity2 :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {ipu : InvertiblePatchUniverse pu ipl}
              {from to : NameSet}
              {ps : Sequence pu from to},
      ∀ (cp : ContextedPatch pu to),
      (fun x ⇒
           addSequenceToContext (invert ps)
               (addSequenceToContext ps x)) cp
    = (fun x ⇒ x) cp.
Proof with auto.
intros.
simpl.
apply addSequenceToContextInverse2...
Qed.

(*  XXX Move this lemma to contexted_patches: *)
Lemma nameOfCommutePast :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {from to : NameSet}
              {p : pu_type from to}
```

```
                {cp : ContextedPatch pu to}
                {cp' : ContextedPatch pu from},
        CommutePast p cp cp'
    → contextedPatch_name cp' = contextedPatch_name cp.
Proof with auto.
intros.
induction H.
    simpl.
    destruct (commuteNames H) as [? [? ?]]...
simpl in *...
Qed.

(*  XXX Move this lemma to contexted_patches: *)
Lemma nameOfCommuteManyPast :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {from to : NameSet}
                {ps : Sequence pu from to}
                {cp : ContextedPatch pu to}
                {cp' : ContextedPatch pu from},
        CommuteManyPast ps cp cp'
    → contextedPatch_name cp' = contextedPatch_name cp.
Proof with auto.
intros.
induction H...
rewrite ← IHCommuteManyPast.
apply (nameOfCommutePast CommutePPast).
Qed.

(*  XXX Move this lemma?: *)
Lemma nameOfCommuteOneMany :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {from mid1 mid2 to : NameSet}
                {p : pu_type from mid1}
                {qs : Sequence pu mid1 to}
                {qs' : Sequence pu from mid2}
                {p' : pu_type mid2 to},
        «p, qs» <~> «qs', p'»
    → pu_nameOf p = pu_nameOf p'.
Proof with auto.
intros.
induction H...
rewrite ← IHOneManyCommute.
destruct (commuteNames commutePQ)...
Qed.

(*  XXX Move this lemma?: *)
Lemma nameOfCommuteManyOne :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {from mid1 mid2 to : NameSet}
                {ps : Sequence pu from mid2}
                {q : pu_type mid2 to}
                {q' : pu_type from mid1}
                {ps' : Sequence pu mid1 to},
        «ps, q» <~> «q', ps'»
    → pu_nameOf q = pu_nameOf q'.
Proof with auto.
intros.
induction H...
rewrite IHManyOneCommute.
destruct (commuteNames commutePQ) as [? [? ?]]...
Qed.
Lemma CatchCommuteNames :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {ipu : InvertiblePatchUniverse pu ipl}
                {from mid1 mid2 to : NameSet}
                {p : Catch ipl from mid1} {q : Catch ipl mid1 to}
                {q' : Catch ipl from mid2} {p' : Catch ipl mid2 to}
                (c : « p , q » <~>c « q' , p' »),
        (catch_name p = catch_name p') ∧
        (catch_name q = catch_name q') ∧
        (catch_name p ≠ catch_name q).
Proof with auto.
intros.
dependent destruction c; dependent destruction catchCommuteDetails; simpl; subst...
                (*  case 4 *)
                rewrite nameOfAddSequenceToContext.
                rewrite nameOfAddSequenceToContext...
                (*  case 5 *)
                apply commuteNames...
            (*  case 6 *)
            destruct qIdentity.
            destruct qIdentity'.
            apply nameOfCommutePast in H0.
            simpl in *.
            destruct (OneManyCommuteNames H).
            split...
        (*  case 7 *)
        destruct pIdentity.
        destruct pIdentity'.
        simpl in *.
        apply nameOfCommutePast in H0.
    simpl in *.
    destruct (ManyOneCommuteNames H).
    split...
(*  case 8 *)
destruct pIdentity.
destruct pIdentity'.
destruct qIdentity.
destruct qIdentity'.
simpl in *.
apply nameOfCommuteManyPast in H3.
apply nameOfCommuteManyPast in H4...
Qed.

Definition CatchCommutable {pu_type : NameSet → NameSet → Type}
                           {ppu : PartPatchUniverse pu_type pu_type}
                           {pui : PatchUniverseInv ppu ppu}
                           {pu : PatchUniverse pui}
                           {ipl : InvertiblePatchlike pu_type}
                           {ipu : InvertiblePatchUniverse pu ipl}
                           {from mid1 to : NameSet}
                           (p : Catch ipl from mid1)
                           (q : Catch ipl mid1 to) : Prop
 := ∃ mid2 : NameSet,
    ∃ q' : Catch ipl from mid2,
    ∃ p' : Catch ipl mid2 to,
    «p, q» <~>c «q', p'».
Notation "p <~?~>c q" := (CatchCommutable p q)
    (at level 60, no associativity).

Lemma CatchCommutable_dec :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike pu_type}
             {ipu : InvertiblePatchUniverse pu ipl}
             {from mid to : NameSet}
             (p : Catch ipl from mid)
             (q : Catch ipl mid to),
      {p <~?~>c q} + {~(p <~?~>c q)}.
Proof with auto.
intros.
destruct p; destruct q.
            (*  Two patches case *)
            destruct (commutable_dec p p0) as [commutable | not_commutable].
                left.
                destruct commutable as [mid' [q' [p' c']]].
                ∃ mid'.
                ∃ (MkCatch q').
                ∃ (MkCatch p').
                apply isCatchCommute5.
                    apply mkCommuteNum.
                apply MkCatchCommute5...
```

```
              right.
              intro c.
              destruct c as [mid' [q' [p' c']]].
              destruct c'; dependent destruction catchCommuteDetails.
              elim not_commutable.
              ∃ mid2.
              ∃ q'.
              ∃ p'...
          (*  Patch/Conflictor case *)
          admit.
      (*  Conflictor/Patch case *)
      admit.
  (*  Conflictor/Conflictor case *)
  admit.
  Qed.

  Lemma CatchCommuteSelfInverse :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {ipu : InvertiblePatchUniverse pu ipl}
                {cs : CommuteSquare pu_type pu_type}
                {from mid1 mid2 to : NameSet}
                {p : Catch ipl from mid1}
                {q : Catch ipl mid1 to}
                {q' : Catch ipl from mid2}
                {p' : Catch ipl mid2 to}
                (c : «p, q» <~>c «q', p'»),
        («q', p'» <~>c «p, q»).
  Proof with auto.
  intros.
  destruct c; destruct catchCommuteDetails.
                                  (*  MkCatchCommute1 case *)
                                  apply isCatchCommute1.
                                      apply mkCommuteNum.
                                  apply MkCatchCommute1.
                                      apply sym_not_eq...
                                  intro commutable.
                                  elim do_not_commute.
                                  destruct commutable as [mid [q' [p' Hcommute]]].
                                  ∃ mid.
                                  ∃ (invert p').
                                  ∃ (invert q')...
                                  apply commuteInverses in Hcommute.
                                  rewrite invertInverse in Hcommute.
                                  apply commuteSelfInverse...
                              (*  MkCatchCommute2 case *)
                              subst.
                              apply isCatchCommute3.
                                  apply mkCommuteNum.
                              eapply MkCatchCommute3...
```

```
            (*  MkCatchCommute3 case *)
            subst.
            apply isCatchCommute2.
                apply mkCommuteNum.
            eapply MkCatchCommute2...
        (*  MkCatchCommute4 case *)
        apply isCatchCommute4.
            apply mkCommuteNum.
        (*
         XXX
         refine (MkCatchCommute4
                     qEffect' qConflicts' qIdentity'
                     pEffect' pConflicts' (map (addSequenceToContext (invert qEffect)) pC
                     pEffect pConflicts pIdentity
                     qEffect qConflicts qIdentity
                     commonEffect _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)...
                                           subst.
                                           rewrite nameOfAddSequenceToContext.
                                           rewrite nameOfAddSequenceToContext...
                                       apply H1.
                                   subst.
                                   rewrite addSequenceToContextInverse2...
                               apply H.
                           rewrite map_map.
                           rewrite (map_ext _ _ addSequenceToContextIdentity1).
                           rewrite map_id...
                       subst.
                       rewrite addSequenceToContextInverse1...
                   subst.
                   rewrite map_map.
                   rewrite (map_ext _ _ addSequenceToContextIdentity2).
                   rewrite map_id...
             subst.
             rewrite addSequenceToContextInverse2...
         apply sequenceCommuteInverse...
         *)
        admit.
    (*  MkCatchCommute5 case *)
    apply isCatchCommute5.
        apply mkCommuteNum.
    apply MkCatchCommute5.
    apply commuteSelfInverse in H...
(*  MkCatchCommute6 case *)
apply isCatchCommute7.
    apply mkCommuteNum.
apply MkCatchCommute7.
                rewrite ← (nameOfCommuteOneMany H).
                rewrite (nameOfCommutePast H0).
                apply sym_not_eq...
            apply commute_OneMany_ManyOne...
        admit.
    admit.
```

```
        subst.
        apply map_neq_nil...
    (*  MkCatchCommute7 case *)
    apply isCatchCommute6.
        apply mkCommuteNum.
    apply MkCatchCommute6.
                    rewrite ← (nameOfCommuteManyOne H).
                    rewrite (nameOfCommutePast H0).
                    apply sym_not_eq...
                admit.
            apply commute_ManyOne_OneMany...
        admit.
    admit.
    subst.
    apply map_neq_nil...
(*  MkCatchCommute8 case *)
apply isCatchCommute8.
    apply mkCommuteNum.
eapply MkCatchCommute8...
                                    rewrite (nameOfCommuteManyPast H3).
                                    rewrite (nameOfCommuteManyPast H4)...
                                (*  XXX apply sequenceCommuteInverse... *)
                                admit.
                            clear - H2.
                            (*  XXX signedNameSetDec. *)
                            admit.
                        admit.
                    admit.
                admit.
            admit.
        admit.
    subst.
    apply map_neq_nil...
subst.
apply map_neq_nil...
Qed.

Lemma CatchCommuteUnique :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike pu_type}
             {ipu : InvertiblePatchUniverse pu ipl}
             {from mid mid' mid'' to : NameSet}
             {p : Catch ipl from mid} {q : Catch ipl mid to}
             {q' : Catch ipl from mid'} {p' : Catch ipl mid' to}
             {q'' : Catch ipl from mid''} {p'' : Catch ipl mid'' to}
             (commute1 : «p, q» <~>c «q', p'»)
             (commute2 : «p, q» <~>c «q'', p''»),
      (mid' = mid'') ∧ (p' ~= p'') ∧ (q' ~= q'').
Proof with auto.
intros.
```

```
dependent destruction commute1;
dependent destruction catchCommuteDetails;
dependent destruction commute2;
dependent destruction catchCommuteDetails.
(*  20: Rule 1 / Rule 1 *)
(*  XXX Can we replace all these substs with a single tactic? *)
subst contextedP.
subst contextedP0.
subst contextedQ.
subst contextedQ0.
subst conflictsP.
subst conflictsP0.
subst conflictsQ.
subst conflictsQ0.
subst p'Effect0.
subst q'Effect.
subst p'Effect.
subst q'Effect0.
proofIrrel invQConsOK invQConsOK0.
proofIrrel invPConsOK invPConsOK0.
proofIrrel pConflictsQ pConflictsQ0.
proofIrrel p'EffectCanonicallyOrdered p'EffectCanonicallyOrdered0.
split...
(*  19: Rule 1 / Rule 2 *)
congruence.
(*  18: Rule 1 / Rule 6 *)
simpl in noConflict.
elim noConflict.
signedNameSetDec.
(*  17: Rule 2 / Rule 1 *)
congruence.
(*  16: Rule 2 / Rule 2 *)
proofIrrel namesDifferent namesDifferent0.
apply (f_equal invert) in x.
rewrite invertInverse in x.
rewrite invertInverse in x.
subst.
proofIrrel invPRsConsOK invPRsConsOK0.
proofIrrel pEffect'CanonicallyOrdered pEffect'CanonicallyOrdered0.
proofIrrel qEffect'CanonicallyOrdered qEffect'CanonicallyOrdered0.
proofIrrel invQEffect'pOK invQEffect'pOK0.
proofIrrel p'ConflictsConflict p'ConflictsConflict0.
proofIrrel q'ConflictsConflict q'ConflictsConflict0.
split...
(*  15: Rule 2 / Rule 6 *)
subst.
simpl in noConflict.
elim noConflict.
signedNameSetDec.
(*  14: Rule 3 / Rule 3 *)
proofIrrel invPRsConsOK invPRsConsOK0.
assert (pEffect'sEqual : «pEffect'» <~~>* «pEffect'0»).
```

```
    apply SymmetricTransitiveCommute in H9.
    apply (TransitiveTransitiveCommute H2 H9)...
assert (pEffect' = pEffect'0).
    apply (CanonicallyOrderedUnique pEffect'CanonicallyOrdered pEffect'CanonicallyOrdered0
pEffect'sEqual).
subst.
proofIrrel invPEffectq'OK invPEffectq'OK0.
proofIrrel p'ConflictsConflict p'ConflictsConflict0.
proofIrrel pEffect'CanonicallyOrdered pEffect'CanonicallyOrdered0.
split...
(*  13: Rule 3 / Rule 4 *)
apply map_neq_nil2 in H15.
congruence.
(*  12: Rule 3 / Rule 8 *)
subst.
elim H14.
assert (X : addSequenceToContext [] (MkContextedPatch (invert pEffect) q')
          = MkContextedPatch (invert pEffect) q').
    admit.
rewrite X.
dependent destruction p'ConflictsConflict...
(*  11: Rule 4 / Rule 3 *)
apply map_neq_nil2 in H8.
congruence.
(*  10: Rule 4 / Rule 4 *)
subst.
destruct (uncommutableSame _ _ H H9 H6 H16
                 noneCommutePastCommonEffectQOnlyEffect
                 noneCommutePastCommonEffectQOnlyEffect0)
    as [X1 [X2 X3]].
subst.
subst.
doCommuteUnique H6 H16.
proofIrrel appendOK_commonEffect_qOnlyEffect appendOK_commonEffect_qOnlyEffect0.
assert («qEffect'» <~~>* «qEffect'0»).
    apply SymmetricTransitiveCommute in H11.
    apply (TransitiveTransitiveCommute H1 H11).
assert (qEffect' = qEffect'0).
    apply CanonicallyOrderedUnique...
subst.
split...
proofIrrel p'ConflictsConflict p'ConflictsConflict0.
proofIrrel pEffect'CanonicallyOrdered pEffect'CanonicallyOrdered0.
proofIrrel q'ConflictsConflict q'ConflictsConflict0.
proofIrrel qEffect'CanonicallyOrdered qEffect'CanonicallyOrdered0.
split...
(*  9: Rule 4 / Rule 8 *)
subst.
```

XXX dependent destruction qConflictsConflict.

H17 : ~ conflictsWith pIdentity (addSequenceToContext qEffect qIdentity)

H0 : conflictsWith qIdentity (addSequenceToContext (invert qEffect) pIdentity)

assert (pConflictsWithQ: conflictsWith pIdentity (addSequenceToContext qEffect qIdentity)). admit.

congruence. XXX.

Want: conflictsWith pIdentity (addSequenceToContext qEffect qIdentity)).

qConflictsConflict : Forall (conflictsWith qIdentity) (addSequenceToContext (invert qEffect) pIdentity :: qOtherConflicts) => conflictsWith qIdentity (addSequenceToContext (invert qEffect) pIdentity) => conflictsWith (addSequenceToContext (invert qEffect) pIdentity) qIdentity

qConflictsConflict : Forall (conflictsWith qIdentity) (addSequenceToContext (invert qEffect) pIdentity :: qOtherConflicts)

H17 : ~ conflictsWith pIdentity (addSequenceToContext qEffect qIdentity)

qConflictsConflict0 : Forall (conflictsWith qIdentity) (addSequenceToContext (invert qEffect) pIdentity :: qOtherConflicts)

p'ConflictsConflict : Forall (conflictsWith (addSequenceToContext (invert qEffect) pIdentity)) (qIdentity :: map (addSequenceToContext (invert qEffect)) pConflicts)

```
admit.
(*  8: Rule 5 / Rule 5 *)
doCommuteUnique H H0...
(*  7: Rule 6 / Rule 1 *)
simpl in noConflict.
elim noConflict.
signedNameSetDec.
(*  6: Rule 6 / Rule 2 *)
simpl in noConflict.
elim noConflict.
signedNameSetDec.
(*  5: Rule 6 / Rule 6 *)
admit.
(*  4: Rule 7 / Rule 7 *)
admit.
(*  3: Rule 8 / Rule 3 *)
admit.
(*  2: Rule 8 / Rule 4 *)
admit.
(*  1: Rule 8 / Rule 8 *)
admit.
Qed.

Lemma CatchCommuteUnique2 :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {ipu : InvertiblePatchUniverse pu ipl}
              {from mid mid' to : NameSet}
              {p : Catch ipl from mid} {q : Catch ipl mid to}
              {q' : Catch ipl from mid'} {p' : Catch ipl mid' to}
```

```
                {q'' : Catch ipl from mid'} {p'' : Catch ipl mid' to}
                (commute1 : «p, q» <~>c «q', p'»)
                (commute2 : «p, q» <~>c «q'', p''»),
        (p' = p'') ∧ (q' = q'').
Proof with auto.
(*
XXX
intros.
dependent destruction commute1;
dependent destruction catchCommuteDetails;
dependent destruction commute2;
dependent destruction catchCommuteDetails.
(*  Rule 1 / Rule 1 *)
split...
assert (HX1 : nilOK = nilOK0).
    apply proof_irrelevance.
rewrite HX1.
assert (HX2 : invQContains = invQContains0).
    clear - invQConsOK invQConsOK0.
    destruct invQConsOK.
    destruct invQConsOK0.
    apply SignedNameSetEquality.
    remember (pu_nameOf (q^)) as HX. (*  Workaround for coq 2464 *)
    signedNameSetDec.
subst.
assert (HX2 : invQConsOK = invQConsOK0).
    apply proof_irrelevance.
rewrite HX2...
(*  Rule 1 / Rule 2 *)
congruence.
(*  Rule 1 / Rule 6 *)
admit.
(*  Others... *)
congruence.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
admit.
(*
destruct (commuteUnique _ H H0).
subst.
split...
admit.
*)
*)
admit.
Qed.

End catches_definition.
```

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches

## 13 Catches and Repos

```
Module Export catches.

Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import commute_square.
Require Import contexted_patches.
Require Import catches_definition.
Require Import catches_commute_consistent.

Lemma CatchCommuteConsistent2 :
      ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                (ipl : InvertiblePatchlike pu_type)
                {ipu : InvertiblePatchUniverse pu ipl}
                {o op opq opqr or oq oqr : NameSet}
                {q3 : Catch ipl o oq}
                {r3 : Catch ipl oq oqr}
                {p3 : Catch ipl oqr opqr}
                {q4 : Catch ipl or oqr}
                {r4 : Catch ipl o or}
                {p1 : Catch ipl o op}
                {q1 : Catch ipl op opq}
                {r1 : Catch ipl opq opqr}
                {p5 : Catch ipl oq opq},
      «q3, r3» <˜>c «r4, q4»
   → «r3, p3» <˜>c «p5, r1»
   → «q3, p5» <˜>c «p1, q1»
   → (∃ opr : NameSet,
        (∃ r2 : Catch ipl op opr,
          (∃ q2 : Catch ipl opr opqr,
          (∃ p6 : Catch ipl or opr,
           «q1, r1» <~>c «r2, q2» ∧
           «q4, p3» <~>c «p6, q2» ∧
           «r4, p6» <~>c «p1, r2»)))).
Proof with auto.
intros.
(*
dependent destruction H; dependent destruction H0; dependent destruction H1...

XXX.



    destruct (commuteNames _ H) as HX1 [HX2 HX3].
    admit.
(*     congruence. *)
destruct (commuteConsistent1 _ H H0 H1) as or [r4 [q4 [p6 [H2 [H3 H4]]]]].
exists or.
exists (MkCatch r4).
exists (MkCatch q4).
exists (MkCatch p6).
split.
    apply MkCatchCommute...
split.
    apply MkCatchCommute...
apply MkCatchCommute...

(*  XXX *)
*)
admit.
Qed.

Instance CatchPartPatchUniverse
            {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {pu : PatchUniverse pui}
            (ipl : InvertiblePatchlike pu_type)
            {ipu : InvertiblePatchUniverse pu ipl}
          : PartPatchUniverse (Catch ipl) (Catch ipl)
    := mkPartPatchUniverse
          (Catch ipl)
          (Catch ipl)
          (@CatchCommute _ _ _ _ _ _)
          cheat (*  (@CatchCommutable_dec _ _ _ _ _) *)
          cheat (*  (@CatchCommuteUnique1 _ _ _ _ _) *)
          (*  (@CatchCommuteUnique2 _ _ _ _ _) *) .

Instance CatchPatchUniverseInv
            {pu_type : NameSet → NameSet → Type}
            {ppu : PartPatchUniverse pu_type pu_type}
            {pui : PatchUniverseInv ppu ppu}
            {pu : PatchUniverse pui}
```

```
                 (ipl : InvertiblePatchlike pu_type)
                 {ipu : InvertiblePatchUniverse pu ipl}
                 {cs : CommuteSquare pu_type pu_type}
               : PatchUniverseInv (CatchPartPatchUniverse ipl) (CatchPartPatchUniverse ipl)
     := mkPatchUniverseInv
               (Catch ipl)
               (Catch ipl)
               (CatchPartPatchUniverse ipl)
               (CatchPartPatchUniverse ipl)
               (@CatchCommuteSelfInverse _ _ _ _ _ _ _).

Instance CatchPatchUniverse
                 {pu_type : NameSet → NameSet → Type}
                 {ppu : PartPatchUniverse pu_type pu_type}
                 {pui : PatchUniverseInv ppu ppu}
                 {pu : PatchUniverse pui}
                 (ipl : InvertiblePatchlike pu_type)
                 {ipu : InvertiblePatchUniverse pu ipl}
                 {cs : CommuteSquare pu_type pu_type}
               : PatchUniverse (CatchPatchUniverseInv ipl)
     := mkPatchUniverse
               (Catch ipl)
               (CatchPartPatchUniverse ipl)
               (CatchPatchUniverseInv ipl)
               (@catch_name _ _ _ _ _ _)
               (@CatchCommuteConsistent1 _ _ _ _ _ _ _)
               (@CatchCommuteConsistent2 _ _ _ _ _ _)
               (@CatchCommuteNames _ _ _ _ _ _).

End catches.
```

coqdoc

printing <~>u $\leftrightsquigarrow_u$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent

```
Module Export catches_commute_consistent.

Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import commute_square.
Require Import catches_definition.
Require Import catches_commute_consistent_1_1.
Require Import catches_commute_consistent_1_2.
Require Import catches_commute_consistent_1_3.
Require Import catches_commute_consistent_1_4.
Require Import catches_commute_consistent_1_5.
Require Import catches_commute_consistent_1_6.
Require Import catches_commute_consistent_1_7.
Require Import catches_commute_consistent_1_8.
(*  This is split up into multiple files to work around high coq memory
   usage *)
Lemma CatchCommuteConsistent1 :
      ∀ {pu_type : NameSet → NameSet → Type}
               {ppu : PartPatchUniverse pu_type pu_type}
               {pui : PatchUniverseInv ppu ppu}
               {pu : PatchUniverse pui}
               {ipl : InvertiblePatchlike pu_type}
               {ipu : InvertiblePatchUniverse pu ipl}
               {cs : CommuteSquare pu_type pu_type}
               {o op opq opqr opr oq oqr : NameSet}
               {p1 : Catch ipl o op}
               {q1 : Catch ipl op opq}
               {r1 : Catch ipl opq opqr}
               {q2 : Catch ipl opr opqr}
               {r2 : Catch ipl op opr}
               {q3 : Catch ipl o oq}
               {r3 : Catch ipl oq oqr}
               {p3 : Catch ipl oqr opqr}
               {p5 : Catch ipl oq opq}
               (commute1 : «q1, r1» <~>c «r2, q2»)
               (commute2 : «p1, q1» <~>c «q3, p5»)
               (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <~>c «r4, q4» ∧
          «p1, r2» <~>c «r4, p6» ∧
          «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction commute1.
apply (CatchCommuteConsistent1_1 catchCommuteDetails commute2 commute3).
apply (CatchCommuteConsistent1_2 catchCommuteDetails commute2 commute3).
apply (CatchCommuteConsistent1_3 catchCommuteDetails commute2 commute3).
apply (CatchCommuteConsistent1_4 catchCommuteDetails commute2 commute3).
apply (CatchCommuteConsistent1_5 catchCommuteDetails commute2 commute3).
apply (CatchCommuteConsistent1_6 catchCommuteDetails commute2 commute3).
apply (CatchCommuteConsistent1_7 catchCommuteDetails commute2 commute3).
apply (CatchCommuteConsistent1_8 catchCommuteDetails commute2 commute3).
Qed.
End catches_commute_consistent.
```

coqdoc

printing <~>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent_1_1

```
Module Export catches_commute_consistent_1_1.

Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import commute_square.
Require Import canonical_order.
Require Import contexted_patches.
Require Import catches_definition.

Lemma CatchCommuteConsistent1_1 :
      ∀ {pu_type : NameSet → NameSet → Type}
               {ppu : PartPatchUniverse pu_type pu_type}
               {pui : PatchUniverseInv ppu ppu}
               {pu : PatchUniverse pui}
               {ipl : InvertiblePatchlike pu_type}
               {ipu : InvertiblePatchUniverse pu ipl}
               {cs : CommuteSquare pu_type pu_type}
               {o op opq opqr opr oq oqr : NameSet}
               {p1 : Catch ipl o op}
               {q1 : Catch ipl op opq}
               {r1 : Catch ipl opq opqr}
               {q2 : Catch ipl opr opqr}
               {r2 : Catch ipl op opr}
               {q3 : Catch ipl o oq}
               {r3 : Catch ipl oq oqr}
               {p3 : Catch ipl oqr opqr}
               {p5 : Catch ipl oq opq}
               (catchCommuteDetails : «q1, r1» <~>1 «r2, q2»)
               (commute2 : «p1, q1» <~>c «q3, p5»)
               (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <~>c «r4, q4» ∧
          «p1, r2» <~>c «r4, p6» ∧
          «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails;
destruct commute2;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails;
destruct commute3;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails.
                (*  case 1/5/1 *)
                destruct (commuteNames H) as [? [? ?]].
                congruence.
            (*  case 1/5/2 *)
            destruct (commuteNames H) as [? [? ?]].
            congruence.
        (*  case 1/5/6 *)
        (*   p1  q1  r1: x  y                   y^; {:y}; :z *)
        (*   p1  r2  q2: x  z                   z^; {:z}; :y *)
        (*   q3  p5  r1: y' x'                  y^; {:y}; :z *)
        (*   q3  r3  p3: y' y'^; {:y'}; :z' x               *)
        (*  q4  p3: z' z'^; {:z'}; :y' x              *)
        (*  p6  q2: z' x''                 z^; {:z}; :y *)
        dependent destruction H0.
        consEquality x.
        dependent destruction H0.
            dependent destruction H1.
                rename p0 into x.
                rename p into y.
                rename q into z.
                rename q' into y'.
                rename p1 into x'.
                set (HX := H).
                clearbody HX.
                apply commuteSelfInverse in HX.
                apply commuteSquare in HX.
                apply commuteSquare in HX.
                apply commuteSquare in HX.
                rewrite invertInverse in HX.
                rewrite invertInverse in HX.
                destruct (commuteUnique HX commutePQ) as [? [? ?]].
                subst.
                subst.
                clear HX commutePQ.
                rename p2 into x.
                rename ident' into z'.
                rename p' into x''.
                subst q'Effect.
                subst p'Effect.
                subst contextedP.
                subst contextedQ.
                subst conflictsP.
                subst conflictsQ.
                rename from1 into o.
                rename mid into ox.
                rename mid1 into oxy.
                rename to2 into oxz.
                rename from0 into oy.
                rename mid' into oz.
                assert (HX : List.map (addToContext x) (MkContextedPatch [] y :: nil)
                            = (MkContextedPatch [] y' :: nil)%list).
                    admit.
                revert q'ConflictsConflict.
                rewrite HX.
                intros.
                clear HX.
```

```
                    ∃ oz.
                    ∃ (MkCatch z').
                    assert (z'i_OK : ConsOK (z' ^) []).
                         constructor.
                         rewrite SequenceContentsNil.
                         signedNameSetDec.
                    assert (canonicallyOrdered : CanonicallyOrdered (z' ^ :> [])).
                         admit.
                    assert (conflictsConflict : List.Forall (conflictsWith (MkContextedPatch [] y'))
                                                                                           (MkContextedPatch
[] z' :: nil)).
                         admit.
                    ∃ (Conflictor (z' ^ :> [])
                                               (MkContextedPatch [] z' :: nil)
                                               (MkContextedPatch [] y') conflictsConflict canonically-
Ordered).
                    ∃ (MkCatch x'').
                    split.
                         apply isCatchCommute1.
                              constructor.
                         apply MkCatchCommute1.
                              destruct (commuteNames H) as [? [HX1 ?]].
                              destruct (commuteNames H0) as [? [HX2 ?]].
                              rewrite ← HX1.
                              rewrite ← HX2...
                         intro z'i_y'_commute.
                         destruct z'i_y'_commute as [sns [y'' [z''i z'i_y'_commute]]].
                         contradiction do_not_commute.
                         apply commuteSquare in H0.
                         apply commuteSelfInverse in H.
                         destruct (commuteConsistent2 z'i_y'_commute H H0) as [? [? [? [? [?
[? ?]]]]]].

                         ∃ x0.
                         ∃ x1.
                         ∃ x2...
                    split.
                         apply isCatchCommute5.
                              constructor.
                         apply MkCatchCommute5...
                    apply isCatchCommute6.
                         constructor.
                    apply MkCatchCommute6.
                                             admit.
                                        admit.
                                   admit.
                              admit.
                         admit.
                    congruence.
               contradiction (ConsEqNil x).
          contradiction (ConsEqNil x).
     (*  case 1/7/4 *)
     admit.
```

```
(*  case 1/7/8 *)
admit.
Qed.
End catches_commute_consistent_1_1.
```

coqdoc

printing <~>u $\rightsquigarrow_u$ printing <~?~>u $\overset{?}{\rightsquigarrow}_u$ printing □u $\epsilon_u$

printing <~> $\rightsquigarrow$ printing <~?~> $\overset{?}{\rightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent_1_2

```
Module Export catches_commute_consistent_1_2.
Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import catches_definition.
Lemma CatchCommuteConsistent1_2 :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {ipu : InvertiblePatchUniverse pu ipl}
              {o op opq opqr opr oq oqr : NameSet}
              {p1 : Catch ipl o op}
              {q1 : Catch ipl op opq}
              {r1 : Catch ipl opq opqr}
              {q2 : Catch ipl opr opqr}
              {r2 : Catch ipl op opr}
              {q3 : Catch ipl o oq}
              {r3 : Catch ipl oq oqr}
              {p3 : Catch ipl oqr opqr}
              {p5 : Catch ipl oq opq}
              (catchCommuteDetails : «q1, r1» <~>2 «r2, q2»)
              (commute2 : «p1, q1» <~>c «q3, p5»)
              (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <~>c «r4, q4» ∧
          «p1, r2» <~>c «r4, p6» ∧
          «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails;
destruct commute2;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails;
```

```
destruct commute3;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails.
                    (*  case 2/5/1 *)
                    destruct (commuteNames H6) as [? [? ?]].
                    congruence.
                (*  case 2/5/2 *)
                destruct (commuteNames H6) as [? [? ?]].
                congruence.
            (*  case 2/5/6 *)
            admit.
        (*  case 2/7/3 *)
        admit.
    (*  case 2/7/4 *)
    admit.
(*  case 2/7/8 *)
admit.
Qed.

End catches_commute_consistent_1_2.
```

coqdoc

printing <˜>u $\leftrightsquigarrow_u$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent_1_3

```
Module Export catches_commute_consistent_1_3.

Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import catches_definition.

Lemma CatchCommuteConsistent1_3_1 :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {ipu : InvertiblePatchUniverse pu ipl}
              {o op opq opqr opr oq oqr : NameSet}
              {p1 : Catch ipl o op}
              {q1 : Catch ipl op opq}
              {r1 : Catch ipl opq opqr}
              {q2 : Catch ipl opr opqr}
              {r2 : Catch ipl op opr}
              {q3 : Catch ipl o oq}
              {r3 : Catch ipl oq oqr}
              {p3 : Catch ipl oqr opqr}
              {p5 : Catch ipl oq opq}
              (catchCommuteDetails1 : «q1, r1» <˜>3 «r2, q2»)
```

```
                (catchCommuteDetails2 : «p1, q1» <~>1 «q3, p5»)
                (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <~>c «r4, q4» ∧
          «p1, r2» <~>c «r4, p6» ∧
          «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
admit.
admit.
admit.
Qed.

Lemma CatchCommuteConsistent1_3_2 :
      ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {ipu : InvertiblePatchUniverse pu ipl}
                {o op opq opqr opr oq oqr : NameSet}
                {p1 : Catch ipl o op}
                {q1 : Catch ipl op opq}
                {r1 : Catch ipl opq opqr}
                {q2 : Catch ipl opr opqr}
                {r2 : Catch ipl op opr}
                {q3 : Catch ipl o oq}
                {r3 : Catch ipl oq oqr}
                {p3 : Catch ipl oqr opqr}
                {p5 : Catch ipl oq opq}
                (catchCommuteDetails1 : «q1, r1» <~>3 «r2, q2»)
                (catchCommuteDetails2 : «p1, q1» <~>2 «q3, p5»)
                (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <~>c «r4, q4» ∧
          «p1, r2» <~>c «r4, p6» ∧
          «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
```

```
admit.
admit.
admit.
Qed.

Lemma CatchCommuteConsistent1_3_3 :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike pu_type}
             {ipu : InvertiblePatchUniverse pu ipl}
             {o op opq opqr opr oq oqr : NameSet}
             {p1 : Catch ipl o op}
             {q1 : Catch ipl op opq}
             {r1 : Catch ipl opq opqr}
             {q2 : Catch ipl opr opqr}
             {r2 : Catch ipl op opr}
             {q3 : Catch ipl o oq}
             {r3 : Catch ipl oq oqr}
             {p3 : Catch ipl oqr opqr}
             {p5 : Catch ipl oq opq}
             (catchCommuteDetails1 : «q1, r1» <~>3 «r2, q2»)
             (catchCommuteDetails2 : «p1, q1» <~>3 «q3, p5»)
             (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <~>c «r4, q4» ∧
          «p1, r2» <~>c «r4, p6» ∧
          «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
admit.
admit.
admit.
Qed.

Lemma CatchCommuteConsistent1_3_4 :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike pu_type}
             {ipu : InvertiblePatchUniverse pu ipl}
             {o op opq opqr opr oq oqr : NameSet}
             {p1 : Catch ipl o op}
             {q1 : Catch ipl op opq}
```

```
                {r1 : Catch ipl opq opqr}
                {q2 : Catch ipl opr opqr}
                {r2 : Catch ipl op opr}
                {q3 : Catch ipl o oq}
                {r3 : Catch ipl oq oqr}
                {p3 : Catch ipl oqr opqr}
                {p5 : Catch ipl oq opq}
                (catchCommuteDetails1 : «q1, r1» <~>3 «r2, q2»)
                (catchCommuteDetails2 : «p1, q1» <~>4 «q3, p5»)
                (commute3 : «p5, r1» <~>c «r3, p3»),
        (∃ or : NameSet,
         (∃ r4 : Catch ipl o or,
          (∃ q4 : Catch ipl or oqr,
           (∃ p6 : Catch ipl or opr,
            «q3, r3» <~>c «r4, q4» ∧
            «p1, r2» <~>c «r4, p6» ∧
            «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
admit.
admit.
admit.
Qed.

Lemma CatchCommuteConsistent1_3_5 :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {ipu : InvertiblePatchUniverse pu ipl}
                {o op opq opqr opr oq oqr : NameSet}
                {p1 : Catch ipl o op}
                {q1 : Catch ipl op opq}
                {r1 : Catch ipl opq opqr}
                {q2 : Catch ipl opr opqr}
                {r2 : Catch ipl op opr}
                {q3 : Catch ipl o oq}
                {r3 : Catch ipl oq oqr}
                {p3 : Catch ipl oqr opqr}
                {p5 : Catch ipl oq opq}
                (catchCommuteDetails1 : «q1, r1» <~>3 «r2, q2»)
                (catchCommuteDetails2 : «p1, q1» <~>5 «q3, p5»)
                (commute3 : «p5, r1» <~>c «r3, p3»),
        (∃ or : NameSet,
         (∃ r4 : Catch ipl o or,
          (∃ q4 : Catch ipl or oqr,
           (∃ p6 : Catch ipl or opr,
            «q3, r3» <~>c «r4, q4» ∧
            «p1, r2» <~>c «r4, p6» ∧
            «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
Qed.

Lemma CatchCommuteConsistent1_3_6 :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike pu_type}
             {ipu : InvertiblePatchUniverse pu ipl}
             {o op opq opqr opr oq oqr : NameSet}
             {p1 : Catch ipl o op}
             {q1 : Catch ipl op opq}
             {r1 : Catch ipl opq opqr}
             {q2 : Catch ipl opr opqr}
             {r2 : Catch ipl op opr}
             {q3 : Catch ipl o oq}
             {r3 : Catch ipl oq oqr}
             {p3 : Catch ipl oqr opqr}
             {p5 : Catch ipl oq opq}
             (catchCommuteDetails1 : «q1, r1» <~>3 «r2, q2»)
             (catchCommuteDetails2 : «p1, q1» <~>6 «q3, p5»)
             (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
            «q3, r3» <~>c «r4, q4» ∧
            «p1, r2» <~>c «r4, p6» ∧
            «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
admit.
admit.
Qed.

Lemma CatchCommuteConsistent1_3_7 :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike pu_type}
```

```
                {ipu : InvertiblePatchUniverse pu ipl}
                {o op opq opqr opr oq oqr : NameSet}
                {p1 : Catch ipl o op}
                {q1 : Catch ipl op opq}
                {r1 : Catch ipl opq opqr}
                {q2 : Catch ipl opr opqr}
                {r2 : Catch ipl op opr}
                {q3 : Catch ipl o oq}
                {r3 : Catch ipl oq oqr}
                {p3 : Catch ipl oqr opqr}
                {p5 : Catch ipl oq opq}
                (catchCommuteDetails1 : «q1, r1» <~>3 «r2, q2»)
                (catchCommuteDetails2 : «p1, q1» <~>7 «q3, p5»)
                (commute3 : «p5, r1» <~>c «r3, p3»),
        (∃ or : NameSet,
         (∃ r4 : Catch ipl o or,
          (∃ q4 : Catch ipl or oqr,
           (∃ p6 : Catch ipl or opr,
            «q3, r3» <~>c «r4, q4» ∧
            «p1, r2» <~>c «r4, p6» ∧
            «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
Qed.

Lemma CatchCommuteConsistent1_3_8 :
        ∀ {pu_type : NameSet → NameSet → Type}
                {ppu : PartPatchUniverse pu_type pu_type}
                {pui : PatchUniverseInv ppu ppu}
                {pu : PatchUniverse pui}
                {ipl : InvertiblePatchlike pu_type}
                {ipu : InvertiblePatchUniverse pu ipl}
                {o op opq opqr opr oq oqr : NameSet}
                {p1 : Catch ipl o op}
                {q1 : Catch ipl op opq}
                {r1 : Catch ipl opq opqr}
                {q2 : Catch ipl opr opqr}
                {r2 : Catch ipl op opr}
                {q3 : Catch ipl o oq}
                {r3 : Catch ipl oq oqr}
                {p3 : Catch ipl oqr opqr}
                {p5 : Catch ipl oq opq}
                (catchCommuteDetails1 : «q1, r1» <~>3 «r2, q2»)
                (catchCommuteDetails2 : «p1, q1» <~>8 «q3, p5»)
                (commute3 : «p5, r1» <~>c «r3, p3»),
        (∃ or : NameSet,
         (∃ r4 : Catch ipl o or,
          (∃ q4 : Catch ipl or oqr,
           (∃ p6 : Catch ipl or opr,
            «q3, r3» <~>c «r4, q4» ∧
            «p1, r2» <~>c «r4, p6» ∧
            «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails1;
dependent destruction catchCommuteDetails2;
destruct commute3;
dependent destruction catchCommuteDetails.
admit.
admit.
admit.
Qed.

Lemma CatchCommuteConsistent1_3 :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {ipu : InvertiblePatchUniverse pu ipl}
              {o op opq opqr opr oq oqr : NameSet}
              {p1 : Catch ipl o op}
              {q1 : Catch ipl op opq}
              {r1 : Catch ipl opq opqr}
              {q2 : Catch ipl opr opqr}
              {r2 : Catch ipl op opr}
              {q3 : Catch ipl o oq}
              {r3 : Catch ipl oq oqr}
              {p3 : Catch ipl oqr opqr}
              {p5 : Catch ipl oq opq}
              (catchCommuteDetails : «q1, r1» <~>3 «r2, q2»)
              (commute2 : «p1, q1» <~>c «q3, p5»)
              (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
            «q3, r3» <~>c «r4, q4» ∧
            «p1, r2» <~>c «r4, p6» ∧
            «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
destruct commute2.
apply (CatchCommuteConsistent1_3_1 catchCommuteDetails catchCommuteDetails0)...
apply (CatchCommuteConsistent1_3_2 catchCommuteDetails catchCommuteDetails0)...
apply (CatchCommuteConsistent1_3_3 catchCommuteDetails catchCommuteDetails0)...
apply (CatchCommuteConsistent1_3_4 catchCommuteDetails catchCommuteDetails0)...
apply (CatchCommuteConsistent1_3_5 catchCommuteDetails catchCommuteDetails0)...
apply (CatchCommuteConsistent1_3_6 catchCommuteDetails catchCommuteDetails0)...
apply (CatchCommuteConsistent1_3_7 catchCommuteDetails catchCommuteDetails0)...
apply (CatchCommuteConsistent1_3_8 catchCommuteDetails catchCommuteDetails0)...
Qed.
```

```
End catches_commute_consistent_1_3.
```

coqdoc

printing <~>u $\leftrightsquigarrow_u$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent_1_4

```
Module Export catches_commute_consistent_1_4.
Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import catches_definition.
Lemma CatchCommuteConsistent1_4 :
     ∀ {pu_type : NameSet → NameSet → Type}
           {ppu : PartPatchUniverse pu_type pu_type}
           {pui : PatchUniverseInv ppu ppu}
           {pu : PatchUniverse pui}
           {ipl : InvertiblePatchlike pu_type}
           {ipu : InvertiblePatchUniverse pu ipl}
           {o op opq opqr opr oq oqr : NameSet}
           {p1 : Catch ipl o op}
           {q1 : Catch ipl op opq}
           {r1 : Catch ipl opq opqr}
           {q2 : Catch ipl opr opqr}
           {r2 : Catch ipl op opr}
           {q3 : Catch ipl o oq}
           {r3 : Catch ipl oq oqr}
           {p3 : Catch ipl oqr opqr}
           {p5 : Catch ipl oq opq}
           (catchCommuteDetails : «q1, r1» <~>4 «r2, q2»)
           (commute2 : «p1, q1» <~>c «q3, p5»)
           (commute3 : «p5, r1» <~>c «r3, p3»),
     (∃ or : NameSet,
      (∃ r4 : Catch ipl o or,
       (∃ q4 : Catch ipl or oqr,
        (∃ p6 : Catch ipl or opr,
         «q3, r3» <~>c «r4, q4» ∧
         «p1, r2» <~>c «r4, p6» ∧
         «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails;
destruct commute2;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails;
destruct commute3;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails.
```

```
(*  case 4/1/3 *)
unfold List.map in H8.
congruence.
(*  case 4/1/4 *)
admit.
(*  case 4/1/8 *)
admit.
(*  case 4/2/3 *)
unfold List.map in H8.
congruence.
(*  case 4/2/4 *)
admit.
(*  case 4/2/8 *)
admit.
(*  case 4/3/3 *)
admit.
(*  case 4/3/4 *)
admit.
(*  case 4/3/8 *)
admit.
(*  case 4/4/3 *)
admit.
(*  case 4/4/4 *)
admit.
(*  case 4/4/8 *)
admit.
(*  case 4/6/1 *)
admit.
(*  case 4/6/2 *)
admit.
(*  case 4/6/6 *)
admit.
(*  case 4/8/3 *)
admit.
(*  case 4/8/4 *)
admit.
(*  case 4/8/8 *)
admit.
Qed.

End catches_commute_consistent_1_4.
```

coqdoc

printing <~>u $\leftrightsquigarrow_u$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent_1_5

```
Module Export catches_commute_consistent_1_5.

Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import catches_definition.
Lemma CatchCommuteConsistent1_5 :
      ∀ {pu_type : NameSet → NameSet → Type}
              {ppu : PartPatchUniverse pu_type pu_type}
              {pui : PatchUniverseInv ppu ppu}
              {pu : PatchUniverse pui}
              {ipl : InvertiblePatchlike pu_type}
              {ipu : InvertiblePatchUniverse pu ipl}
              {o op opq opqr opr oq oqr : NameSet}
              {p1 : Catch ipl o op}
              {q1 : Catch ipl op opq}
              {r1 : Catch ipl opq opqr}
              {q2 : Catch ipl opr opqr}
              {r2 : Catch ipl op opr}
              {q3 : Catch ipl o oq}
              {r3 : Catch ipl oq oqr}
              {p3 : Catch ipl oqr opqr}
              {p5 : Catch ipl oq opq}
              (catchCommuteDetails : «q1, r1» <~>5 «r2, q2»)
              (commute2 : «p1, q1» <~>c «q3, p5»)
              (commute3 : «p5, r1» <~>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <~>c «r4, q4» ∧
          «p1, r2» <~>c «r4, p6» ∧
          «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails;
destruct commute2;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails;
destruct commute3;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails.
    (*  Case 5/5/5 *)
    set (X := commuteConsistent1 H H0 H1).
    destruct X as [or [r4 [q4 [p6 [c1 [c2 c3]]]]]].
    ∃ or.
    ∃ (MkCatch r4).
    ∃ (MkCatch q4).
    ∃ (MkCatch p6).
    split.
        apply isCatchCommute5...
        apply MkCatchCommute5...
    split.
        apply isCatchCommute5...
        apply MkCatchCommute5...
    apply isCatchCommute5...
    apply MkCatchCommute5...
(*  Case 5/7/7 *)
admit.
Qed.
End catches_commute_consistent_1_5.
```

coqdoc

printing <˜>u $\leftrightsquigarrow_{\mathrm{u}}$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_{\mathrm{u}}$ printing □u $\epsilon_{\mathrm{u}}$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent_1_6

```
Module Export catches_commute_consistent_1_6.
Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import catches_definition.
Lemma CatchCommuteConsistent1_6 :
      ∀ {pu_type : NameSet → NameSet → Type}
               {ppu : PartPatchUniverse pu_type pu_type}
               {pui : PatchUniverseInv ppu ppu}
               {pu : PatchUniverse pui}
               {ipl : InvertiblePatchlike pu_type}
               {ipu : InvertiblePatchUniverse pu ipl}
               {o op opq opqr opr oq oqr : NameSet}
               {p1 : Catch ipl o op}
               {q1 : Catch ipl op opq}
               {r1 : Catch ipl opq opqr}
               {q2 : Catch ipl opr opqr}
               {r2 : Catch ipl op opr}
               {q3 : Catch ipl o oq}
               {r3 : Catch ipl oq oqr}
               {p3 : Catch ipl oqr opqr}
               {p5 : Catch ipl oq opq}
               (catchCommuteDetails : «q1, r1» <˜>6 «r2, q2»)
               (commute2 : «p1, q1» <˜>c «q3, p5»)
               (commute3 : «p5, r1» <˜>c «r3, p3»),
      (∃ or : NameSet,
       (∃ r4 : Catch ipl o or,
        (∃ q4 : Catch ipl or oqr,
         (∃ p6 : Catch ipl or opr,
          «q3, r3» <˜>c «r4, q4» ∧
          «p1, r2» <˜>c «r4, p6» ∧
          «p6, q2» <˜>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails;
destruct commute2;
```

```
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails;
destruct commute3;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails.
                    (*  case 6/5/1 *)
                    admit.
                (*  case 6/5/2 *)
                admit.
            (*  case 6/5/6 *)
            admit.
        (*  case 6/7/3 *)
        admit.
    (*  case 6/7/4 *)
    admit.
(*  case 6/7/8 *)
admit.
Qed.

End catches_commute_consistent_1_6.
```

coqdoc

printing <˜>u $\leftrightsquigarrow_u$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catches_commute_consistent_1_7

```
Module Export catches_commute_consistent_1_7.

Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import catches_definition.

Lemma CatchCommuteConsistent1_7 :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
             {ipl : InvertiblePatchlike pu_type}
             {ipu : InvertiblePatchUniverse pu ipl}
             {o op opq opqr opr oq oqr : NameSet}
             {p1 : Catch ipl o op}
             {q1 : Catch ipl op opq}
             {r1 : Catch ipl opq opqr}
             {q2 : Catch ipl opr opqr}
             {r2 : Catch ipl op opr}
             {q3 : Catch ipl o oq}
             {r3 : Catch ipl oq oqr}
             {p3 : Catch ipl oqr opqr}
             {p5 : Catch ipl oq opq}
             (catchCommuteDetails : «q1, r1» <˜>7 «r2, q2»)
```

```
                 (commute2 : «p1, q1» <~>c «q3, p5»)
                 (commute3 : «p5, r1» <~>c «r3, p3»),
         (∃ or : NameSet,
          (∃ r4 : Catch ipl o or,
           (∃ q4 : Catch ipl or oqr,
            (∃ p6 : Catch ipl or opr,
             «q3, r3» <~>c «r4, q4» ∧
             «p1, r2» <~>c «r4, p6» ∧
             «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails;
destruct commute2;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails;
destruct commute3;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails.
                         (*  case 7/1/7 *)
                         admit.
                    (*  case 7/2/7 *)
                    admit.
               (*  case 7/3/7 *)
               admit.
          (*  case 7/4/7 *)
          admit.
     (*  case 7/6/5 *)
     admit.
(*  case 7/8/7 *)
admit.
Qed.

End catches_commute_consistent_1_7.
```

coqdoc

printing <~>u $\leftrightsquigarrow_{u}$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_{u}$ printing $\Box$u $\epsilon_{u}$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing $\Box$ $\epsilon$ catches_commute_consistent_1_8

```
Module Export catches_commute_consistent_1_8.

Require Import Equality.
Require Import names.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.
Require Import contexted_patches.
Require Import catches_definition.

Lemma CatchCommuteConsistent1_8 :
      ∀ {pu_type : NameSet → NameSet → Type}
             {ppu : PartPatchUniverse pu_type pu_type}
             {pui : PatchUniverseInv ppu ppu}
             {pu : PatchUniverse pui}
```

```
                  {ipl : InvertiblePatchlike pu_type}
                  {ipu : InvertiblePatchUniverse pu ipl}
                  {o op opq opqr opr oq oqr : NameSet}
                  {p1 : Catch ipl o op}
                  {q1 : Catch ipl op opq}
                  {r1 : Catch ipl opq opqr}
                  {q2 : Catch ipl opr opqr}
                  {r2 : Catch ipl op opr}
                  {q3 : Catch ipl o oq}
                  {r3 : Catch ipl oq oqr}
                  {p3 : Catch ipl oqr opqr}
                  {p5 : Catch ipl oq opq}
                  (catchCommuteDetails : «q1, r1» <~>8 «r2, q2»)
                  (commute2 : «p1, q1» <~>c «q3, p5»)
                  (commute3 : «p5, r1» <~>c «r3, p3»),
         (∃ or : NameSet,
          (∃ r4 : Catch ipl o or,
           (∃ q4 : Catch ipl or oqr,
            (∃ p6 : Catch ipl or opr,
             «q3, r3» <~>c «r4, q4» ∧
             «p1, r2» <~>c «r4, p6» ∧
             «p6, q2» <~>c «q4, p3»)))).
Proof with auto.
intros.
dependent destruction catchCommuteDetails;
destruct commute2;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails;
destruct commute3;
try (abstract (dependent destruction catchCommuteDetails));
dependent destruction catchCommuteDetails.
(*  case 8/1/3 *)
admit.
(*  case 8/1/4 *)
admit.
(*  case 8/1/8 *)
admit.
(*  case 8/2/3 *)
admit.
(*  case 8/2/4 *)
admit.
(*  case 8/2/8 *)
admit.
(*  case 8/3/3 *)
admit.
(*  case 8/3/4 *)
admit.
(*  case 8/3/8 *)
admit.
(*  case 8/4/3 *)
admit.
(*  case 8/4/4 *)
```

```
admit.
(*  case 8/4/8 *)
admit.
(*  case 8/6/1 *)
admit.
(*  case 8/6/2 *)
admit.
(*  case 8/6/6 *)
admit.
(*  case 8/8/3 *)
admit.
(*  case 8/8/4 *)
admit.
(*  case 8/8/8 *)
admit.
Qed.
End catches_commute_consistent_1_8.
```

coqdoc

printing <˜>u $\leftrightsquigarrow_u$ printing <˜?˜>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <˜> $\leftrightsquigarrow$ printing <˜?˜> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ catch_patch_universe

## 14 Catch Patch Universe

```
Module Export catch_patch_universe.
Require Import hunks.
Require Import catches.
Require Import catches_definition.
Require Import patch_universes.
Require Import names.
Definition HunkCatchType := Catch HunkInvertiblePatchlike.
Definition HunkCatchPartPatchUniverse := CatchPartPatchUniverse HunkInvertiblePatchlike.
Definition HunkCatchPatchUniverseInv := CatchPatchUniverseInv HunkInvertiblePatchlike.
Definition HunkCatchPatchUniverse := CatchPatchUniverse HunkInvertiblePatchlike.
Lemma hunkCatchCommuteInSequenceConsistent :
    ∀ {o oq oqr oqrs ou our ours : NameSet}
           {qs : Sequence HunkCatchPatchUniverse o oq}
           {r : HunkCatchType oq oqr}
           {s : HunkCatchType oqr oqrs}
           (*  ts omitted for now *)
           {us : Sequence HunkCatchPatchUniverse o ou}
           {r' : HunkCatchType ou our}
           {s' : HunkCatchType our ours}
           {vs : Sequence HunkCatchPatchUniverse ours oqrs}

           (s_NilConsOK : ConsOK s [])
           (r_s_NilConsOK : ConsOK r (s :> []))
           (qs_r_s_NilAppendOK : AppendOK qs (r :> s :> []))
```

```
             (s'_vsConsOK : ConsOK s' vs)
             (r'_s'_vsConsOK : ConsOK r' (s' :> vs))
             (us_r'_s'_vsAppendOK : AppendOK us (r' :> (s' :> vs))),
      («qs :+> r :> s :> []» <~~>* «us :+> r' :> s' :> vs»)
  → (catch_name r = catch_name r')
  → (catch_name s = catch_name s')
  → (r <~?~> s)
  → (r' <~?~> s').
Proof.
apply (@commuteInSequenceConsistent HunkCatchType HunkCatchPartPatchUniverse
HunkCatchPatchUniverseInv HunkCatchPatchUniverse).
Qed.
```

$[p]\left([a]\,[b] + [c]\,[d]\right) = [p]\,[a]\,[b]\left[b^{-1}a^{-1}, \{: a, a : b\}, : c\right]\left[\epsilon, \{: a, a : b\}, c : d\right]$

You can trace a sequence $pabb^{-1}a^{-1}ab$

XXXXXXXXXXXXXXXXXXXXXX

XXX Want to show that catches inhabit a patch universe

XXX This all need to be updated properly for catches

XXX At some point we need to say that equality on catches is only up to commutation of their internals

**Lemma 14.1 (catch-commute-unique)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, j \in (\mathbf{P} \times \mathbf{P}) \cup \{\text{fail}\}, k \in (\mathbf{P} \times \mathbf{P}) \cup \{\text{fail}\} \cdot$
$(\langle p, q\rangle \leftrightarrow j) \wedge (\langle p, q\rangle \leftrightarrow k) \Rightarrow j = k$

**Explanation**
*This is Lemma 7.1 restated for catches.*

**Proof**
Catch commute is a load of patch commutes. Each of those patch commutes has a unique result by Lemma 7.1. ■

XXX This needs to be strengthened to only talk about catches adjacent in a repo:

**Lemma 14.2 (catch-commute-self-inverse)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, p' \in \mathbf{P}, q' \in \mathbf{P} \cdot$
$(\langle p, q\rangle \leftrightarrow \langle q', p'\rangle) \Leftrightarrow (\langle q', p'\rangle \leftrightarrow \langle p, q\rangle)$

**Explanation**
*This is Lemma 7.2 restated for catches.*

**Proof**
XXX Need names/numbers for the rules.

If two catches commutes by the patch/conflicting-patch rule, then they commute back by the same rule.

If two catches commutes by the conflictor/conflicting-patch rule, then they commute back by the patch/conflicting-conflictor rule, and vice-versa.

For the conflictor/conflicting-conflictor rule, in the forward direction we have
$\left\langle \left[\overline{rs}, W, x\right], \left[\overline{t}, \{\overline{t}^{-1}x\} \cup Y, z\right]\right\rangle \leftrightarrow \left\langle \left[\overline{rt'}, \overline{s'}Y, \overline{s'}z\right], \left[\overline{s'}, \{z\} \cup \overline{t}^{-1}W, \overline{t}^{-1}x\right]\right\rangle$

Adding an identity (XXX need a lemma that this is an identity) and some parentheses to the right hand side gives us
$\left\langle \left[\overline{rt'}, (\overline{s'}Y), (\overline{s'}z)\right], \left[\overline{s'}, \left\{\overline{s'}^{-1}(\overline{s'}z)\right\} \cup (\bar{t}^{-1}W), (\bar{t}^{-1}x)\right] \right\rangle$
and by the conflictor/conflicting-conflictor rule we get
$\left\langle \left[\overline{rt'}, (\overline{s'}Y), (\overline{s'}z)\right], \left[\overline{s'}, \left\{\overline{s'}^{-1}(\overline{s'}z)\right\} \cup (\bar{t}^{-1}W), (\bar{t}^{-1}x)\right] \right\rangle \leftrightarrow$
$\left\langle \left[\overline{rs}, \bar{t}(\bar{t}^{-1}W), \bar{t}(\bar{t}^{-1}x)\right], \left[\bar{t}, \left\{(\bar{t}^{-1}x)\right\} \cup \overline{s'}^{-1}(\overline{s'}Y), \overline{s'}^{-1}(\overline{s'}z)\right] \right\rangle$
if $\left\langle \overline{t'}, \overline{s'} \right\rangle \leftrightarrow \left\langle \bar{s}, \bar{t} \right\rangle$
$N\left(\bar{r}^{-1}\right) \subseteq N\left(\bar{t}^{-1}W\right)\right)$
$N\left(\overline{t'}^{-1}\right) \cap N\left(\bar{t}^{-1}W\right)\right) = \emptyset$
XXX Show side conditions OK
Finally, simplifying the result (XXX need a lemma for that too) gives us
$\left\langle \left[\overline{rt'}, (\overline{s'}Y), (\overline{s'}z)\right], \left[\overline{s'}, \left\{\overline{s'}^{-1}(\overline{s'}z)\right\} \cup (\bar{t}^{-1}W), (\bar{t}^{-1}x)\right] \right\rangle \leftrightarrow \left\langle \left[\overline{rs}, W, x\right], \left[\bar{t}, \left\{(\bar{t}^{-1}x)\right\} \cup Y, z\right] \right\rangle$
and we are back where we started, as required.

XXX patch/patch

XXX conflictor/patch

XXX patch/conflictor

XXX conflictor/conflictor ■

XXX We need to do something about this one. We probably need to add the cases for inverse conflicting patches in the commute rules. This will have the side-effect that we won't need to strengthen the other lemmata.

**Lemma 14.3 (catch-commute-square)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, \forall p' \in \mathbf{P}, q' \in \mathbf{P} \cdot$
$(\langle p, q \rangle \leftrightarrow \langle q', p' \rangle) \Leftrightarrow \left(\left\langle q'^{-1}, p \right\rangle \leftrightarrow \left\langle p', q^{-1} \right\rangle\right)$

**Explanation**
*This is Lemma 7.3 restated for catches.*

**Proof**
XXX ■

XXX This needs to be strengthened to only talk about catches adjacent in a repo:

**Lemma 14.4 (catch-commute-preserves-commute)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, r \in \mathbf{P}, p' \in \mathbf{P}, q' \in \mathbf{P}, r' \in \mathbf{P} \cdot$
$(\langle p, qr \rangle \leftrightarrow \langle q'r', p' \rangle) \Rightarrow \left(\left(q \underset{?}{\leftrightarrow} r\right) \Leftrightarrow \left(q' \underset{?}{\leftrightarrow} r'\right)\right)$

**Explanation**
*This is Lemma* **??** *restated for catches.*

**Proof**
XXX ■

XXX This needs to be strengthened to only talk about catches adjacent in a repo:

**Lemma 14.5 (catch-commute-consistent)**
$\forall p \in \mathbf{P}, q \in \mathbf{P}, r \in \mathbf{P}, p' \in \mathbf{P}, q' \in \mathbf{P}, r' \in \mathbf{P} \cdot$
$(\langle q, r \rangle \leftrightarrow \langle r', q' \rangle) \Rightarrow ((\langle p, qr \rangle \leftrightarrow \langle \_, p' \rangle) \Leftrightarrow (\langle p, r'q' \rangle \leftrightarrow \langle \_, p' \rangle))$

**Explanation**
*This is Lemma* **??** *restated for catches.*

**Proof**
XXX ■

```
End catch_patch_universe.
```

coqdoc

printing <~>u $\leftrightsquigarrow_u$ printing <~?~>u $\overset{?}{\leftrightsquigarrow}_u$ printing □u $\epsilon_u$

printing <~> $\leftrightsquigarrow$ printing <~?~> $\overset{?}{\leftrightsquigarrow}$ printing □ $\epsilon$ hunks

## 15 Hunks

```
Module Export hunks.
```

End-to-end proof for hunks.

```
Require Import Arith.
Require Import Compare_dec.
Require Import Setoid.

Require Import unnamed_patches.
Require Import names.
Require Import named_patches.
Require Import patch_universes.
Require Import invertible_patchlike.
Require Import invertible_patch_universe.

(*  The offset, remove and add values are the number of lines
   the hunk skips over, removes and adds respectively. *)
Inductive Hunk : Set
    := MkHunk : ∀ (offset remove add : nat), Hunk.

Definition hunkInverse (x : Hunk) : Hunk :=
    match x with
    MkHunk xOff xRemove xAdd ⇒ MkHunk xOff xAdd xRemove
    end.
Notation "p ^u" := (hunkInverse p).

(*  This isn't the best definition of hunk commute that we can make,
   but it is a simple and consistent definition *)
Definition hunkCommute (x y y' x' : Hunk) : Prop :=
    match x, y, y', x' with
    MkHunk xOff xRemove xAdd, MkHunk yOff yRemove yAdd,
    MkHunk yOff' yRemove' yAdd', MkHunk xOff' xRemove' xAdd' ⇒
    (xRemove = xRemove') ∧ (xAdd = xAdd') ∧
    (yRemove = yRemove') ∧ (yAdd = yAdd') ∧
    (
        (
            (xOff + xAdd < yOff) ∧
            (xOff' = xOff) ∧
            (yOff' = yOff - xAdd + xRemove)
        )
```

```
          ∨
          (
               (yOff + yRemove < xOff) ∧
               (xOff' = xOff - yRemove + yAdd) ∧
               (yOff' = yOff)
          )
     )
     end.
Notation "« p , q » <~>u « q' , p' »" := (hunkCommute p q q' p').

Definition HunkCommutable : Hunk → Hunk → Prop
 := fun (p : Hunk) ⇒ fun (q : Hunk) ⇒
     (∃ q' : Hunk, ∃ p' : Hunk,
     «p, q» <~>u «q', p'»).
Notation "p <~?~>u q" := (HunkCommutable p q).

Lemma HunkCommuteUnique :
        ∀ (p : Hunk) (q : Hunk)
                 (p' : Hunk) (q' : Hunk)
                 (p'' : Hunk) (q'' : Hunk),
        «p, q» <~>u «q', p'»
    → «p, q» <~>u «q'', p''»
    → (p' = p'') ∧ (q' = q'').
Proof.
intros.
unfold hunkCommute in H.
unfold hunkCommute in H0.
destruct p.
destruct q.
destruct p'.
destruct q'.
destruct p''.
destruct q''.
destruct H as [H1 H2].
split; f_equal; omega.
Qed.

Lemma HunkCommutable_dec :
        ∀ (p : Hunk) (q : Hunk),
        { q' : Hunk &
        { p' : Hunk &
        «p, q» <~>u «q', p'» }}
        +
        {~(p <~?~>u q)}.
Proof with auto.
intros.
unfold HunkCommutable.
destruct p as [pOff pRemove pAdd].
destruct q as [qOff qRemove qAdd].
unfold hunkCommute.
case (le_gt_dec qOff (pOff + pAdd)).
     case (le_gt_dec pOff (qOff + qRemove)).
          (*  This is the "do not commute" case *)
          right.
        intro.
        destruct H as [p' [q' H]].
        destruct p'.
        destruct q'.
        omega.
    (*  Commute option 2 *)
    left.
    ∃ (MkHunk qOff qRemove qAdd).
    ∃ (MkHunk (pOff - qRemove + qAdd) pRemove pAdd).
    split...
    split...
    split...
(*  Commute option 1 *)
left.
∃ (MkHunk (qOff - pAdd + pRemove) qRemove qAdd).
∃ (MkHunk pOff pRemove pAdd).
split...
split...
split...
Qed.

Lemma HunkCommuteSelfInverseOneWay :
      ∀ (p : Hunk) (q : Hunk)
             (p' : Hunk) (q' : Hunk),
      («p, q» <~>u «q', p'») →
      («q', p'» <~>u «p, q»).
Proof with auto.
intros.
destruct p.
destruct q.
destruct p'.
destruct q'.
unfold hunkCommute in H.
unfold hunkCommute.
omega.
Qed.

Lemma HunkCommuteSelfInverse :
      ∀ (p : Hunk) (q : Hunk)
             (p' : Hunk) (q' : Hunk),
      («p, q» <~>u «q', p'») ↔
      («q', p'» <~>u «p, q»).
Proof.
intros.
split; apply HunkCommuteSelfInverseOneWay.
Qed.

Lemma HunkCommuteSquare :
      ∀ (p : Hunk) (q : Hunk)
             (p' : Hunk) (q' : Hunk),
      «p, q» <~>u «q', p'» →
      « q'^u, p» <~>u «p', q^u».
Proof.
intros.
```

```
destruct p.
destruct q.
destruct p'.
destruct q'.
unfold hunkCommute in H.
unfold hunkCommute.
unfold hunkInverse.
omega.
Qed.

Lemma HunkCommuteConsistent1 :
       ∀ (p1 : Hunk)
              (q1 : Hunk)
              (r1 : Hunk)
              (q2 : Hunk)
              (r2 : Hunk)
              (q3 : Hunk)
              (r3 : Hunk)
              (p3 : Hunk)
              (p5 : Hunk),
       «q1, r1» <~>u «r2, q2»
    → «p1, q1» <~>u «q3, p5»
    → «p5, r1» <~>u «r3, p3»
    → ∃ r4 : Hunk,
       ∃ q4 : Hunk,
       ∃ p6 : Hunk,
       «q3, r3» <~>u «r4, q4» ∧
       «p1, r2» <~>u «r4, p6» ∧
       «p6, q2» <~>u «q4, p3».
Proof with auto.
intros.
destruct p1 as [p1Offset p1Remove p1Add].
destruct q1 as [q1Offset q1Remove q1Add].
destruct r1 as [r1Offset r1Remove r1Add].
destruct q2 as [q2Offset q2Remove q2Add].
destruct r2 as [r2Offset r2Remove r2Add].
destruct p3 as [p3Offset p2Remove p2Add].
destruct q3 as [q3Offset q3Remove q3Add].
destruct r3 as [r3Offset r3Remove r3Add].
destruct p5 as [p5Offset p5Remove p5Add].
unfold hunkCommute in *.
intuition.

(*  Case 1: p changes before q, q changes before r *)
∃ (MkHunk (r3Offset - q3Add + q3Remove) r3Remove r3Add).
∃ (MkHunk q3Offset q3Remove q3Add).
∃ (MkHunk p1Offset p1Remove p1Add).
subst.
intuition.

(*  Case 3: q changes before p, p changes before r *)
∃ (MkHunk (r3Offset - q3Add + q3Remove) r3Remove r3Add).
∃ (MkHunk q3Offset q3Remove q3Add).
∃ (MkHunk p1Offset p1Remove p1Add).
```

```
subst.
intuition.
(*  Case 4: q changes before r, r changes before p *)
∃ (MkHunk (r3Offset - q3Add + q3Remove) r3Remove r3Add).
∃ (MkHunk q3Offset q3Remove q3Add).
∃ (MkHunk (p1Offset - r1Remove + r1Add) p1Remove p1Add).
subst.
intuition.
(*  Case 5: p changes before r, r changes before q *)
∃ (MkHunk r3Offset r3Remove r3Add).
∃ (MkHunk (q3Offset - r3Remove + r3Add) q3Remove q3Add).
∃ (MkHunk p1Offset p1Remove p1Add).
subst.
intuition.
(*  Case 6: r changes before p, p changes before q *)
∃ (MkHunk r3Offset r3Remove r3Add).
∃ (MkHunk (q3Offset - r3Remove + r3Add) q3Remove q3Add).
∃ (MkHunk (p1Offset - r1Remove + r1Add) p1Remove p1Add).
subst.
intuition.
(*  Case 8: r changes before q, q changes before p *)
∃ (MkHunk r3Offset r3Remove r3Add).
∃ (MkHunk (q3Offset - r3Remove + r3Add) q3Remove q3Add).
∃ (MkHunk (p1Offset - r1Remove + r1Add) p1Remove p1Add).
subst.
intuition.
Qed.
Lemma HunkCommuteConsistent2 :
      ∀ (p3 : Hunk)
             (q3 : Hunk)
             (r3 : Hunk)
             (q4 : Hunk)
             (r4 : Hunk)
             (q1 : Hunk)
             (r1 : Hunk)
             (p1 : Hunk)
             (p5 : Hunk),
      «q3, r3» <~>u «r4, q4»
   → «r3, p3» <~>u «p5, r1»
   → «q3, p5» <~>u «p1, q1»
   → ∃ r2 : Hunk,
      ∃ q2 : Hunk,
      ∃ p6 : Hunk,
      «q1, r1» <~>u «r2, q2» ∧
      «q4, p3» <~>u «p6, q2» ∧
      «r4, p6» <~>u «p1, r2».
Proof with auto.
intros.
destruct p3 as [p3Offset p3Remove p3Add].
destruct q3 as [q3Offset q3Remove q3Add].
destruct r3 as [r3Offset r3Remove r3Add].
destruct q4 as [q4Offset q4Remove q4Add].
destruct r4 as [r4Offset r4Remove r4Add].
destruct p1 as [p1Offset p1Remove p1Add].
destruct q1 as [q1Offset q1Remove q1Add].
destruct r1 as [r1Offset r1Remove r1Add].
destruct p5 as [p5Offset p5Remove p5Add].
unfold hunkCommute in *.
intuition.

(*  Case 1: q changes before r, r changes before p *)
∃ (MkHunk (r1Offset - q1Add + q1Remove) r1Remove r1Add).
∃ (MkHunk q1Offset q1Remove q1Add).
∃ (MkHunk (p3Offset - q4Add + q4Remove) p1Remove p1Add).
subst.
intuition.

(*  Case 3: q changes before p, p changes before r *)
∃ (MkHunk (r1Offset - q1Add + q1Remove) r1Remove r1Add).
∃ (MkHunk q1Offset q1Remove q1Add).
∃ (MkHunk (p3Offset - q4Add + q4Remove) p1Remove p1Add).
subst.
intuition.

(*  Case 4: p changes before q, q changes before r *)
∃ (MkHunk (r1Offset - q1Add + q1Remove) r1Remove r1Add).
∃ (MkHunk q1Offset q1Remove q1Add).
∃ (MkHunk p1Offset p1Remove p1Add).
subst.
intuition.

(*  Case 5: r changes before q, q changes before p *)
∃ (MkHunk r1Offset r1Remove r1Add).
∃ (MkHunk (q1Offset - r1Remove + r1Add) q1Remove q1Add).
∃ (MkHunk (p3Offset - q4Add + q4Remove) p1Remove p1Add).
subst.
intuition.

(*  Case 6: r changes before p, p changes before q *)
∃ (MkHunk r1Offset r1Remove r1Add).
∃ (MkHunk (q1Offset - r1Remove + r1Add) q1Remove q1Add).
∃ (MkHunk p3Offset p1Remove p1Add).
subst.
intuition.

(*  Case 8: p changes before r, r changes before q *)
∃ (MkHunk r1Offset r1Remove r1Add).
∃ (MkHunk (q1Offset - r1Remove + r1Add) q1Remove q1Add).
∃ (MkHunk p1Offset p1Remove p1Add).
subst.
intuition.
Qed.

Lemma hunkInverseInverse : ∀ (p : Hunk), (p ^u)^u = p.
Proof.
intros.
destruct p.
unfold hunkInverse.
```

```
auto.
Qed.

Definition HunkUnnamedPatch : UnnamedPatch :=
    mkUnnamedPatch
        Hunk
        hunkInverse
        hunkInverseInverse
        hunkCommute
        HunkCommuteUnique
        HunkCommutable_dec
        HunkCommuteSelfInverse
        HunkCommuteSquare
        HunkCommuteConsistent1
        HunkCommuteConsistent2.

Instance HunkPartPatchUniverse : PartPatchUniverse (NamedPatch HunkUnnamedPatch)
(NamedPatch HunkUnnamedPatch) := NamedPartPatchUniverse HunkUnnamedPatch.
Instance HunkPatchUniverseInv : PatchUniverseInv (NamedPartPatchUniverse HunkUn-
namedPatch) (NamedPartPatchUniverse HunkUnnamedPatch) := NamedPatchUniverseInv
HunkPartPatchUniverse.
Instance HunkPatchUniverse :
            PatchUniverse (NamedPatchUniverseInv HunkPartPatchUniverse)
     := NamedPatchUniverse HunkPatchUniverseInv.
Instance HunkInvertiblePatchlike : InvertiblePatchlike (NamedPatch HunkUnnamedPatch) :=
NamedInvertiblePatchlike HunkUnnamedPatch.
Instance HunkInvertiblePatchUniverse :
            InvertiblePatchUniverse (NamedPatchUniverse HunkPatchUniverseInv)
                                    (NamedInvertiblePatchlike HunkUnnamedPatch)
     := NamedInvertiblePatchUniverse.

Lemma hunkCommuteInSequenceConsistent :
    ∀ {o oq oqr oqrs ou our ours : NameSet}
          {qs : Sequence HunkPatchUniverse o oq}
          {r : NamedPatch HunkUnnamedPatch oq oqr}
          {s : NamedPatch HunkUnnamedPatch oqr oqrs}
          (*  ts omitted for now *)
          {us : Sequence HunkPatchUniverse o ou}
          {r' : NamedPatch HunkUnnamedPatch ou our}
          {s' : NamedPatch HunkUnnamedPatch our ours}
          {vs : Sequence HunkPatchUniverse ours oqrs}

          (s_NilConsOK : ConsOK s [])
          (r_s_NilConsOK : ConsOK r (s :> []))
          (qs_r_s_NilAppendOK : AppendOK qs (r :> s :> []))

          (s'_vsConsOK : ConsOK s' vs)
          (r'_s'_vsConsOK : ConsOK r' (s' :> vs))
          (us_r'_s'_vsAppendOK : AppendOK us (r' :> (s' :> vs))),
    («qs :+> r :> s :> []» <~~>* «us :+> r' :> s' :> vs»)
 → (pu_nameOf r = pu_nameOf r')
 → (pu_nameOf s = pu_nameOf s')
 → (r <~?~> s)
 → (r' <~?~> s').
```

```
Proof.
apply (@commuteInSequenceConsistent
            (NamedPatch HunkUnnamedPatch)
            HunkPartPatchUniverse
            HunkPatchUniverseInv
            HunkPatchUniverse).
Qed.
End hunks.
```

## A More catch bits

XXX This whole section wants to be merged back in as appropriate.

XXX Merge bits

**Conjecture A.1 (merge-commute-cannot-fail)**
The commute in the merge definition cannot fail.

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

**Conjecture A.2 (merge-makes-repos)**
If $(\overline{c}d) \in \mathbf{R}, (\overline{c}e) \in \mathbf{R}, d' \in \mathbf{C}, e' \in \mathbf{C}, d + e = \langle e', d' \rangle$ then $\overline{c}de' \in \mathbf{R} \wedge \overline{c}ed' \in \mathbf{R}$.

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

**Conjecture A.3 (merge-effect)**
XXX This is rubbish. Fix it. Or just rely on the spec of the effect of a repo, which we haven't given yet.

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

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

**Conjecture A.4 (merge-symmetric)**
If $(\overline{c}d) \in \mathbf{R}$, $(\overline{c}e) \in \mathbf{R}$ and $d + e = \langle e', d' \rangle$ then $e + d = \langle d', e' \rangle$.

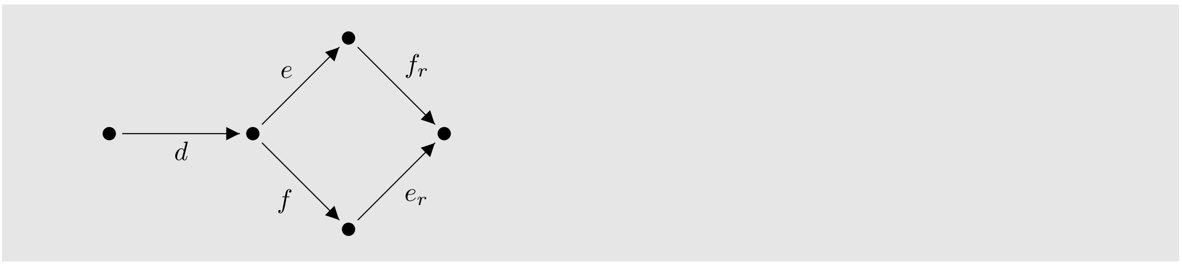
**Explanation**
*Merging $d$ and $e$ is the same as merging $e$ and $d$.*

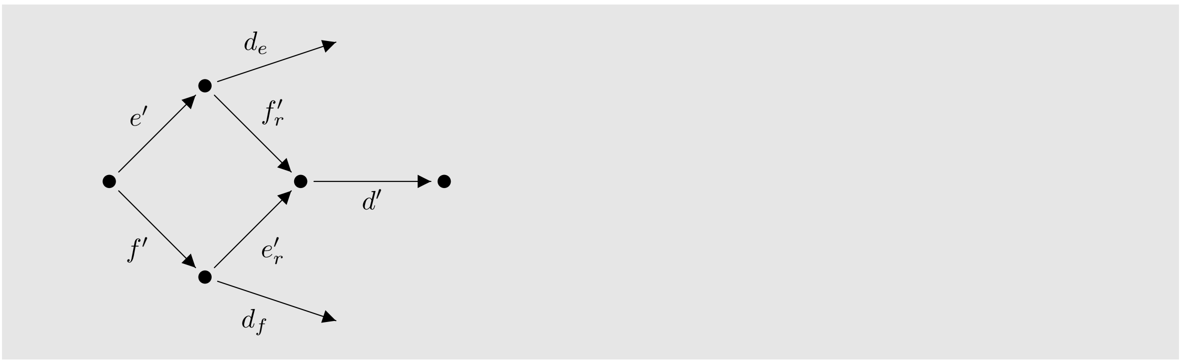
**Conjecture A.5 (merge-commute)**
If $(\overline{c}d) \in \mathbf{R}$, $(\overline{c}e) \in \mathbf{R}$ and $d + e = \langle e', d' \rangle$ then $\langle d, e' \rangle \leftrightarrow \langle e, d' \rangle$.

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

**Conjecture A.6 (merge-commute2)**
If $(\overline{c}de) \in \mathbf{R}, (\overline{c}df) \in \mathbf{R}, e + f = \langle f_r, e_r \rangle, \langle d, e \rangle \leftrightarrow \langle e', d_e \rangle, \langle d, f \rangle \leftrightarrow \langle f', d_f \rangle$ then
$\langle d_e, f_r \rangle \leftrightarrow \langle \_, d' \rangle, \langle d_f, e_r \rangle \leftrightarrow \langle \_, d' \rangle, e' + f' = \langle f'_r, e'_r \rangle, d_e + f'_r = \langle \_, d' \rangle, d_f + e'_r = \langle \_, d' \rangle$

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





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

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXX commute bits

**Definition A.1 (catch-commute)**
We extend $\leftrightarrow$ to operate on catches, as defined below.

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

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

**Unrelated**

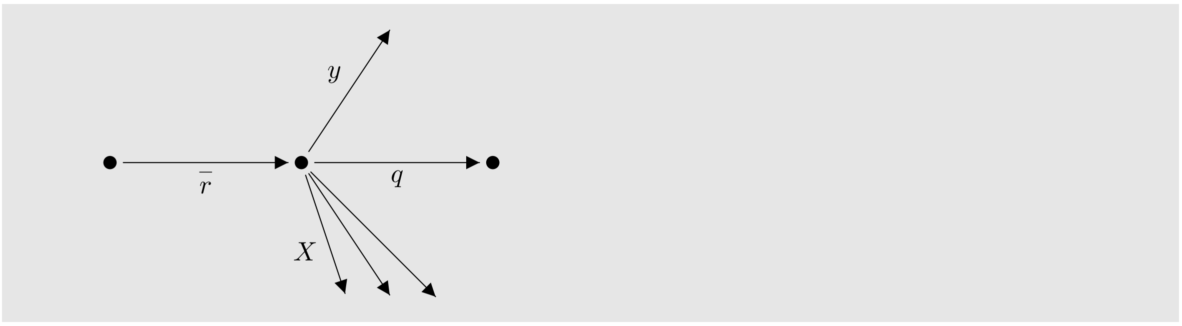
First the simple non-conflicted patch case:
$\langle [p], [q] \rangle \leftrightarrow \langle [q'], [p'] \rangle$ if $\langle p, q \rangle \leftrightarrow \langle q', p' \rangle$

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

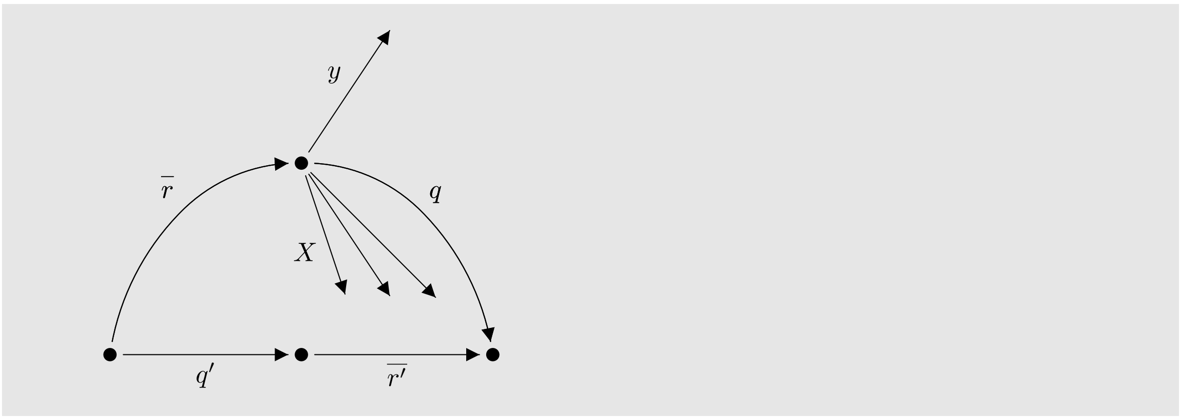
Now commute a conflictor past a patch, where everything goes smoothly:

$$\begin{aligned}\left\langle \left[\overline{r}, X, y\right], [q] \right\rangle \leftrightarrow \left\langle [q'], \left[\overline{r'}, X', y'\right] \right\rangle \text{ if } & \left\langle \overline{r}, q \right\rangle \leftrightarrow \left\langle q', \overline{r'} \right\rangle \\ & \left\langle q^{-1}, y \right\rangle \rightarrow \langle y' \rangle \\ & \left\langle q^{-1}, X \right\rangle \rightarrow \langle X' \rangle\end{aligned}$$

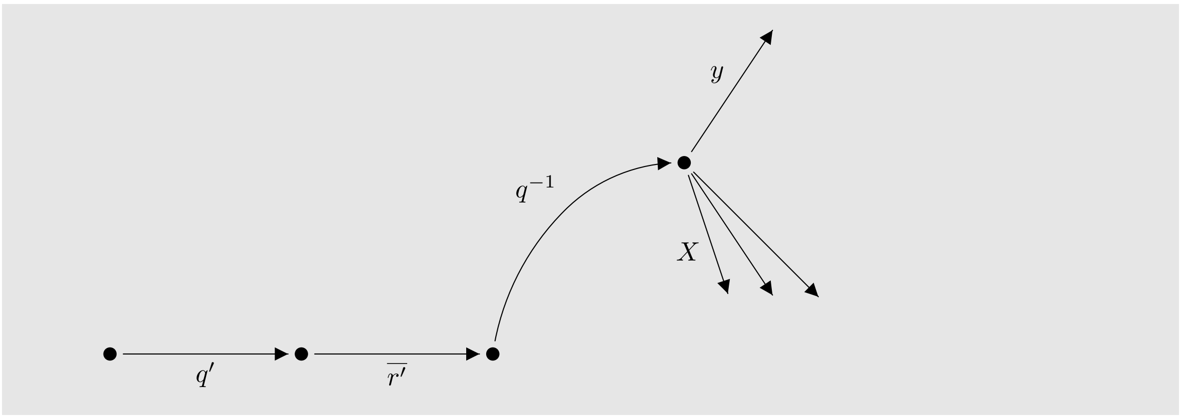
**Explanation**
*OK, now things start to look a little more daunting, but if you work through it then it's really not that bad. We start off with this situation:*



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



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



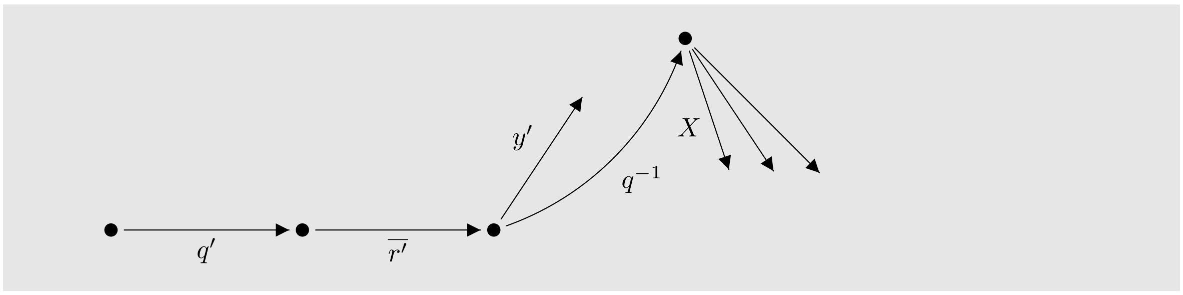
*So we need to add $q^{-1}$ into the context of $y$ and $X$. We have two ways of doing this: We can simply add $q^{-1}$ to the context, which cannot fail (by definition), or we can commute $q^{-1}$ past $y$ and $X$, which can. Being able to commute patches is good, so if possible we would prefer to use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

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

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

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

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



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

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

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

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

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

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

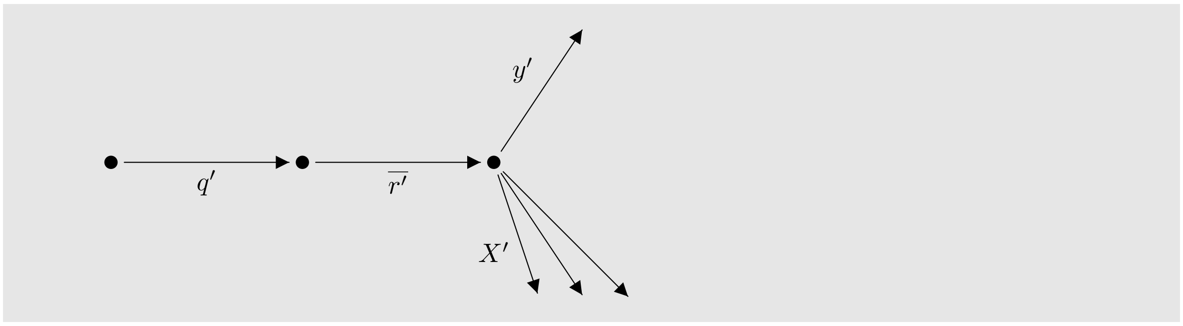
$$[s]\left[s^{-1},\{:s\},:t\right][v]\left[\epsilon,\left\{v^{-1}:s\right\},:u'\right]$$

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

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

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

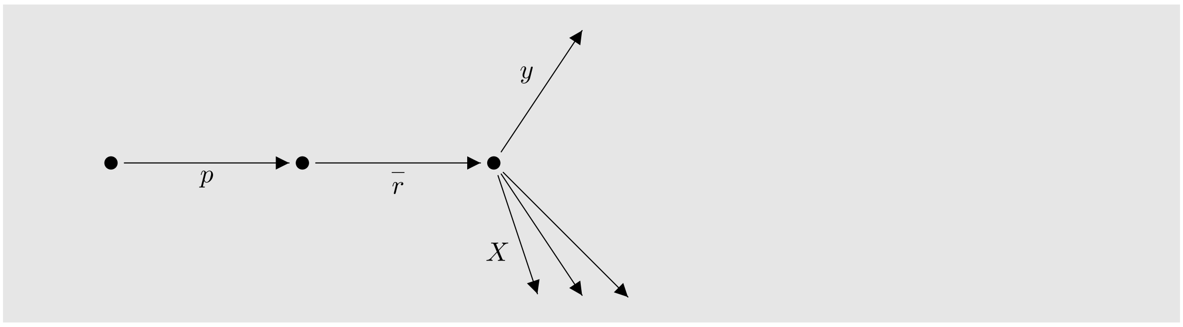
*This explains why we require* $\left\langle q^{-1},X\right\rangle \to \langle X'\rangle$, *and brings us to our destination:*



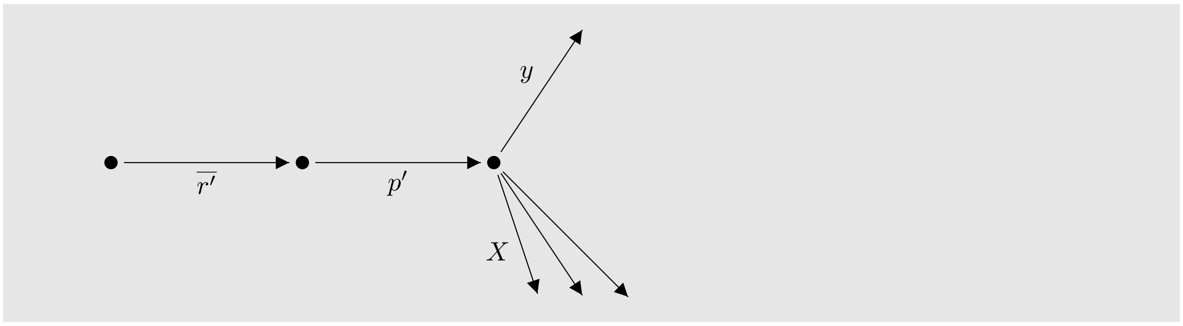
Next we have the case where the patch and conflictor start off the other way round:

$$\left\langle [p], \left[\overline{r}, X, y\right]\right\rangle \leftrightarrow \left\langle \left[\overline{r'}, X', y'\right], [p']\right\rangle \text{ if } \begin{array}{l} \left\langle p,\overline{r}\right\rangle \leftrightarrow \left\langle \overline{r'},p'\right\rangle \\ \langle p',X\rangle \to \langle X'\rangle \\ \langle p',y\rangle \to \langle y'\rangle \end{array}$$

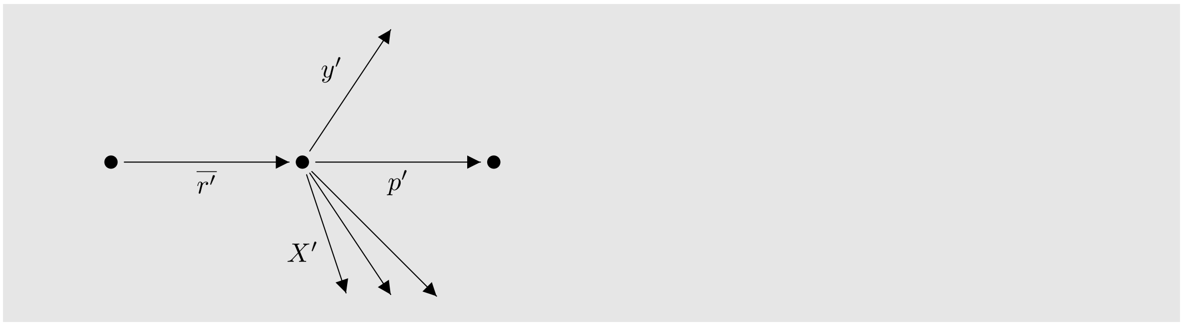
**Explanation**
*This is similar to the previous case. We start off here:*



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



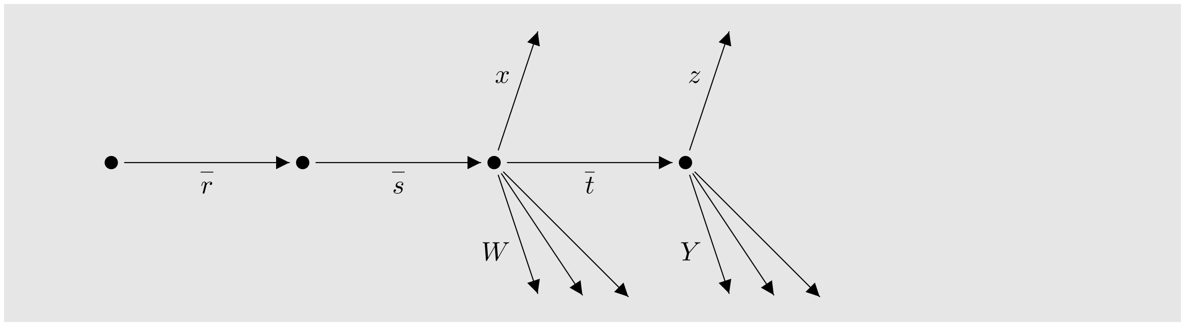
*We now need to get* $X$ *and* $y$ *into the right context, and as we want commute to be self-inverting, we'd better require that the commute pasts succeed, as we did in the previous case:*



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

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

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



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

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

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

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

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

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

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

Now, we can commute $u$ and $v$, giving us

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

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

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

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

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

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

### Conflicted merge

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

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

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

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

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

**Explanation**
*This is just the inverse of the previous case.*

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

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

### Fail

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

**Conjecture A.7 (catch-commute-conflicting-patches-succeeds)**
$\forall(\overline{c}de) \in \mathbf{R}\cdot$
$d \in \mathscr{C}(e) \Rightarrow (\langle d, e \rangle \leftrightarrow \langle \_, \_ \rangle)$

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

**Conjecture A.8 (catch-commute-unique)**
$\forall(\overline{c}de) \in \mathbf{R}, r \in (\mathbf{C} \times \mathbf{C}) \cup \{\text{fail}\} , s \in (\mathbf{C} \times \mathbf{C}) \cup \{\text{fail}\} \cdot$
$(\langle d, e \rangle \leftrightarrow r) \wedge (\langle d, e \rangle \leftrightarrow s) \Rightarrow r = s$

**Explanation**
*XXX*

**Conjecture A.9 (catch-commute-valid)**
$\forall(\overline{c}de) \in \mathbf{R}, d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$(\langle d, e\rangle \leftrightarrow \langle e', d'\rangle) \Rightarrow (\overline{c}e'd') \in \mathbf{R}$

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

**Conjecture A.10 (catch-commute-self-inverse)**
$\forall(\overline{c}de) \in \mathbf{R}, \forall d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$(\langle d, e\rangle \leftrightarrow \langle e', d'\rangle) \Leftrightarrow (\langle e', d'\rangle \leftrightarrow \langle d, e\rangle)$

> **Explanation**
> *XXX*

**Conjecture A.11 (catch-commute-square)**
$\forall(\overline{c}d) \in \mathbf{R}, (\overline{c}e) \in \mathbf{R}, \forall d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$(\left\langle d^{-1}, e\right\rangle \leftrightarrow \left\langle e', d'^{-1}\right\rangle) \Leftrightarrow (\left\langle e^{-1}, d\right\rangle \leftrightarrow \left\langle d', e'^{-1}\right\rangle)$

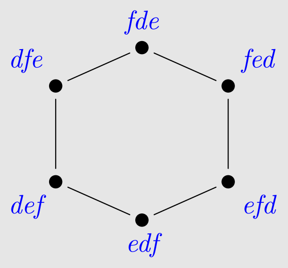
> **Explanation**
> *XXX*

**Conjecture A.12 (catch-commute-preserves-effect)**
$\forall(\overline{c}de) \in \mathbf{R}, \forall d' \in \mathbf{C}, e' \in \mathbf{C} \cdot$
$(\langle d, e\rangle \leftrightarrow \langle e', d'\rangle) \Rightarrow \mathscr{E}(de) = \mathscr{E}(e'd')$

> **Explanation**
> *XXX*

**Conjecture A.13 (catch-commute-associates)**
$\forall(\overline{c}def) \in \mathbf{R}, d_e \in \mathbf{C}, d_f \in \mathbf{C}, d_{ef} \in \mathbf{C}, e_d \in \mathbf{C}, e_f \in \mathbf{C}, e_{df} \in \mathbf{C}, f_d \in \mathbf{C}, f_e \in \mathbf{C}, f_{de} \in \mathbf{C} \cdot$
$(\langle e, f\rangle \leftrightarrow \langle f_e, e_f\rangle) \wedge (\langle d, e\rangle \leftrightarrow \langle e_d, d_e\rangle) \wedge (\langle d_e, f\rangle \leftrightarrow \langle f_d, d_{ef}\rangle)$
$\Leftrightarrow (\langle e_d, f_d\rangle \leftrightarrow \langle f_{de}, e_{df}\rangle) \wedge (\langle e_{df}, d_{ef}\rangle \leftrightarrow \langle d_f, e_f\rangle) \wedge (\langle f_{de}, d_f\rangle \leftrightarrow \langle d, f_e\rangle)$

> **Explanation**
> *XXX i.e. that commute associates, holds for catches too. Here's the diagram again for convenience:*



## B Named Patch Motivation

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

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

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

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

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

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

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

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

XXX Change the /

XXX Say key point is merge equates by name, commute by content