Darcs and GADTs

Ganesh Sittampalam
Credit Suisse

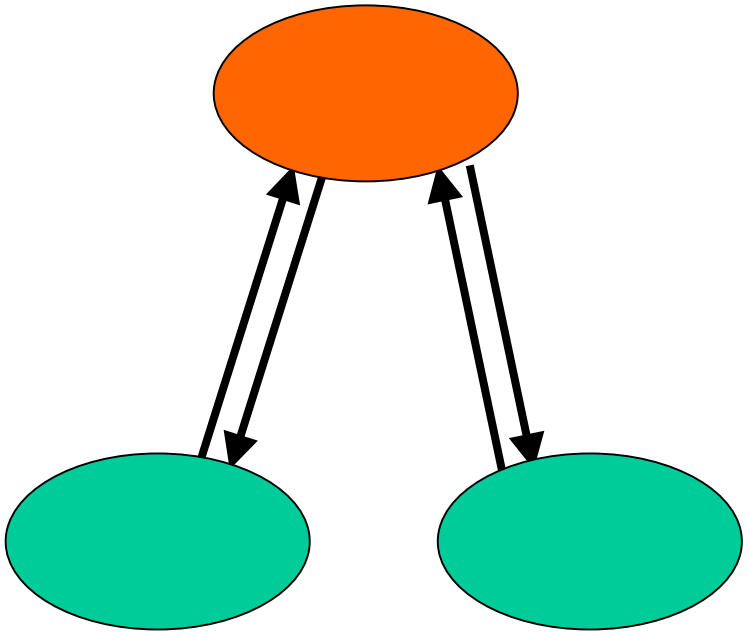
Outline

- Darcs
- GADTs
- How GADTs help darcs

Version control systems

- Keep a history of changes to a project
- A basis for collaboration
  - Lock based
  - Merge based

Centralised VCS



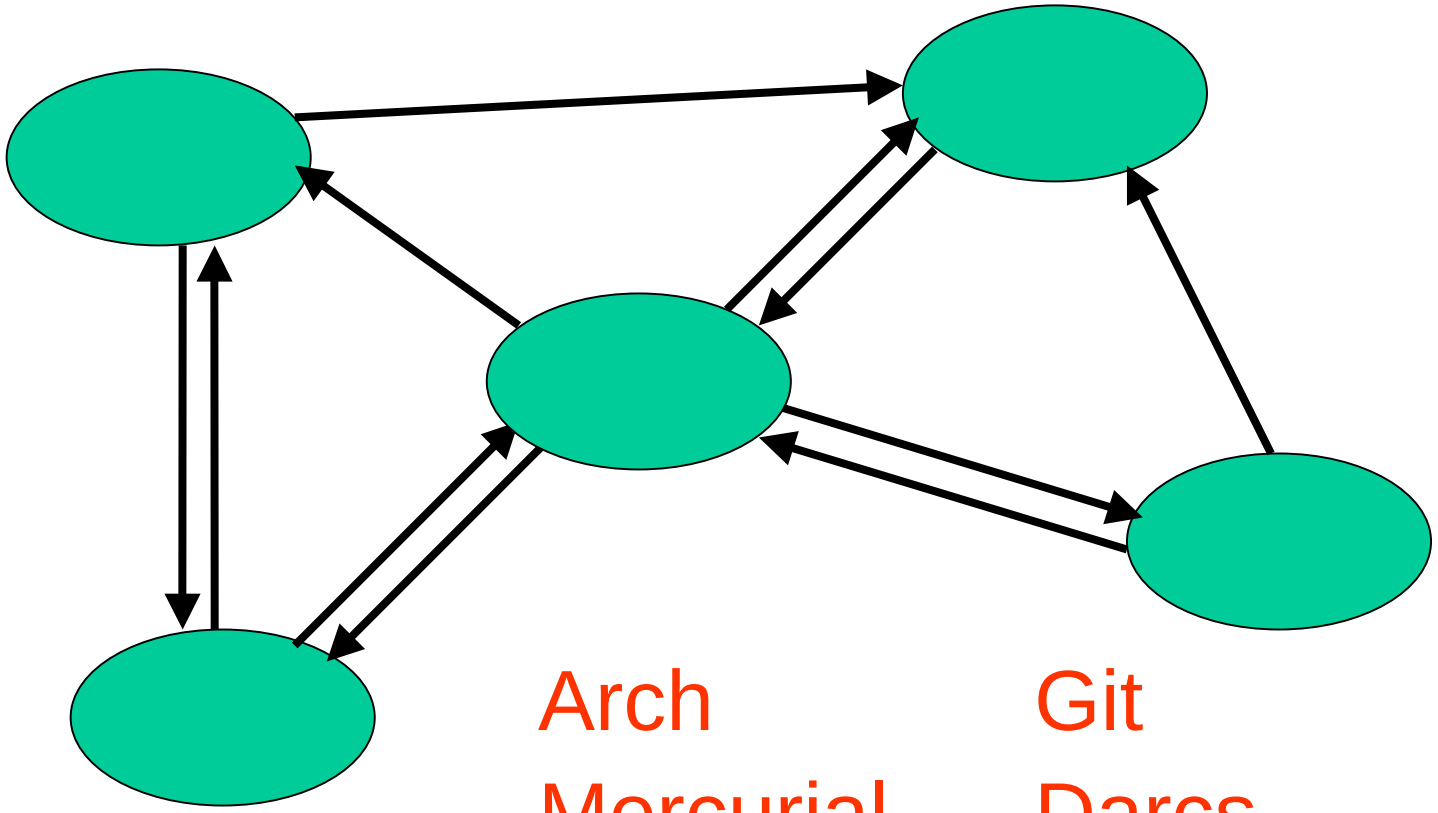
CVS

Subversion (SVN)

Clearcase

…

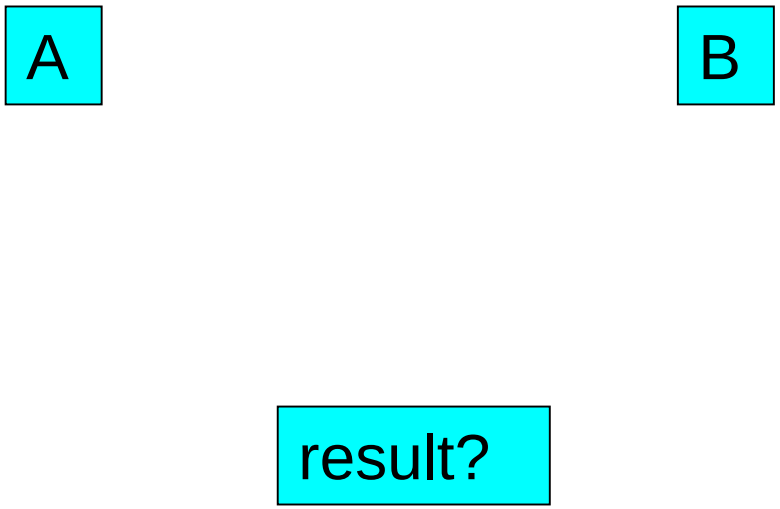
Distributed VCS



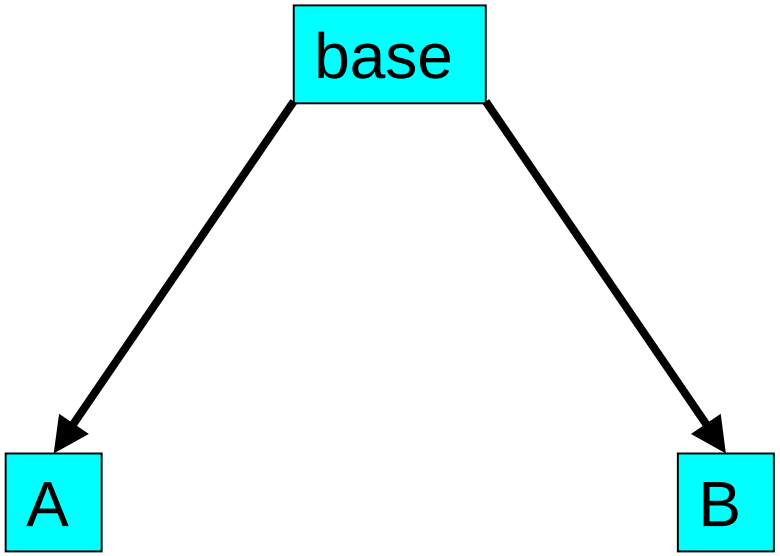
Arch
Mercurial
Bzr

Git
Darcs
...

Tree-based merging



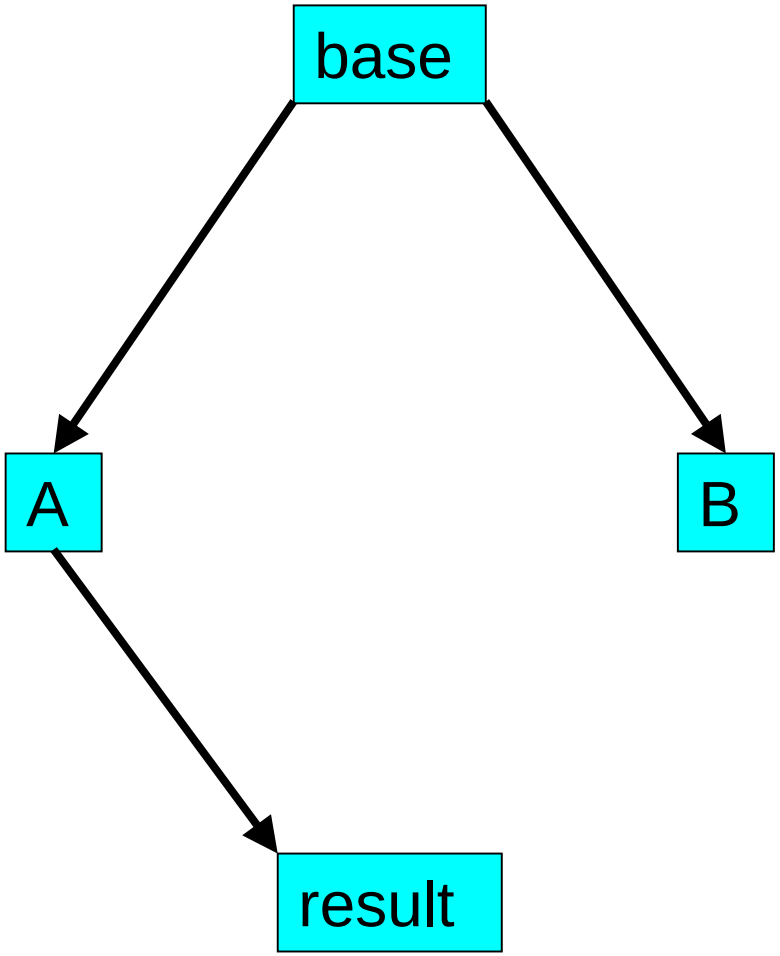
Tree-based merging



- Find base

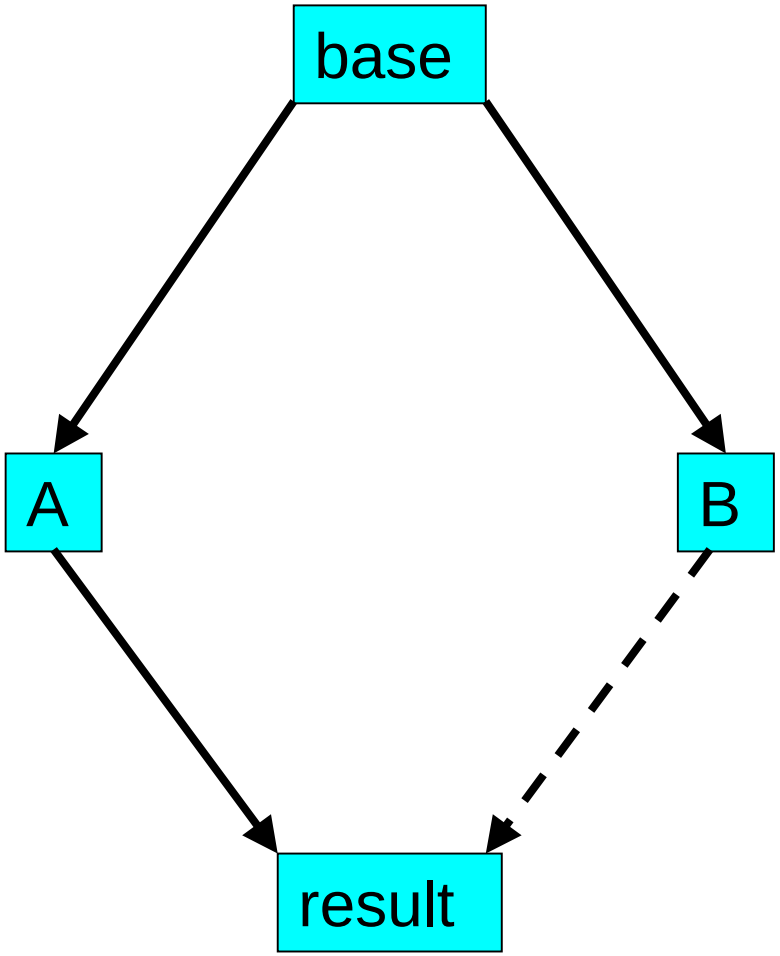
result?

Tree-based merging



- Find base
- Calculate diffs
- Adjust offsets
- Apply

Tree-based merging



- Find base
- Calculate diffs
- Adjust offsets
- Apply

Tree-based merging

- Base can be hard to find
  - Can have to artificially construct it
- Ignores the intermediate revisions
  - Not compositional
- Can choose algorithm at time of merge

Patch-based merging

- Each revision is viewed as a “patch” against the previous version
- Merge algorithm is locked in when patch is created
  - “Intelligent” patch types like token replace
- Sequences of revisions merged by merging individual patches
  - Results repeatable and compositional

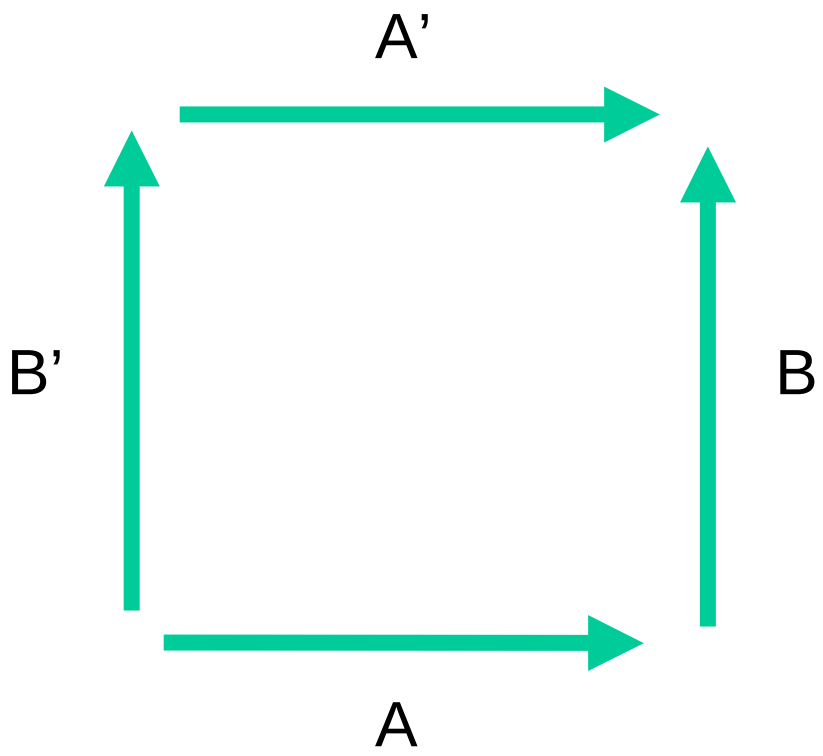
Darcs

- Repo is conceptually a set of patches
- Representation places them in an order
  - Only some orders are valid

- Push and pull patches between repos
- Branches are just repos with different sets of patches
  - Merge = take the union

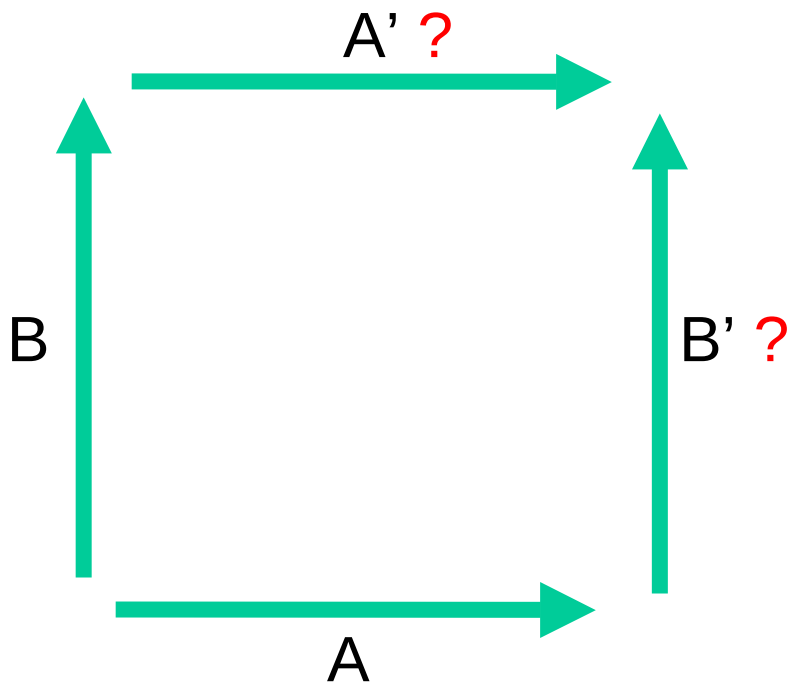
Commutation

AB ⟷ B'A'

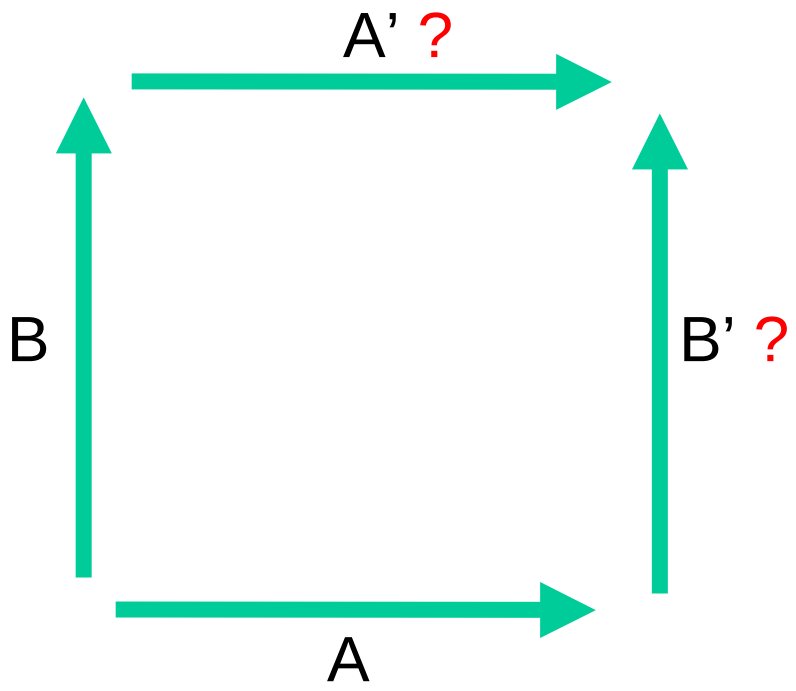


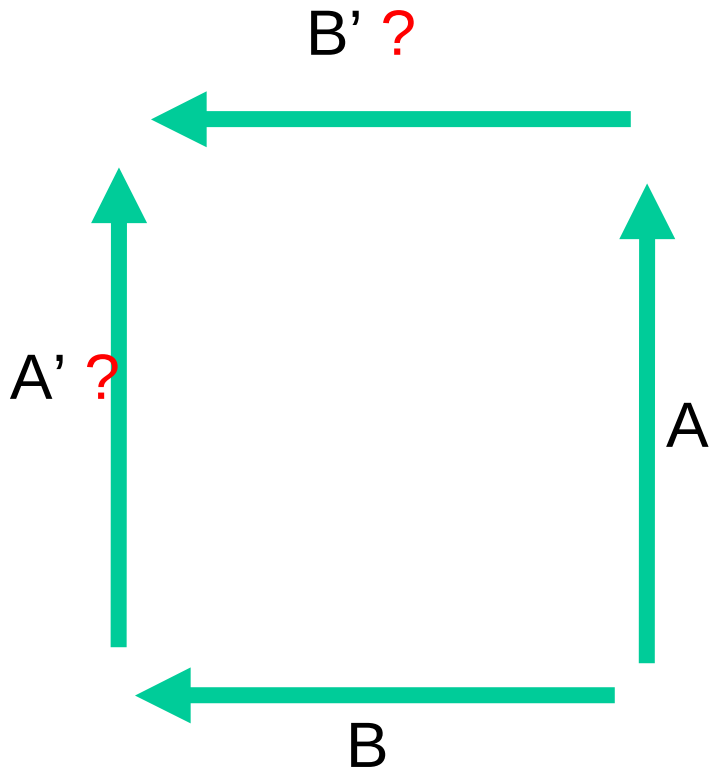
Makes *cherry-picking* easy

Merging

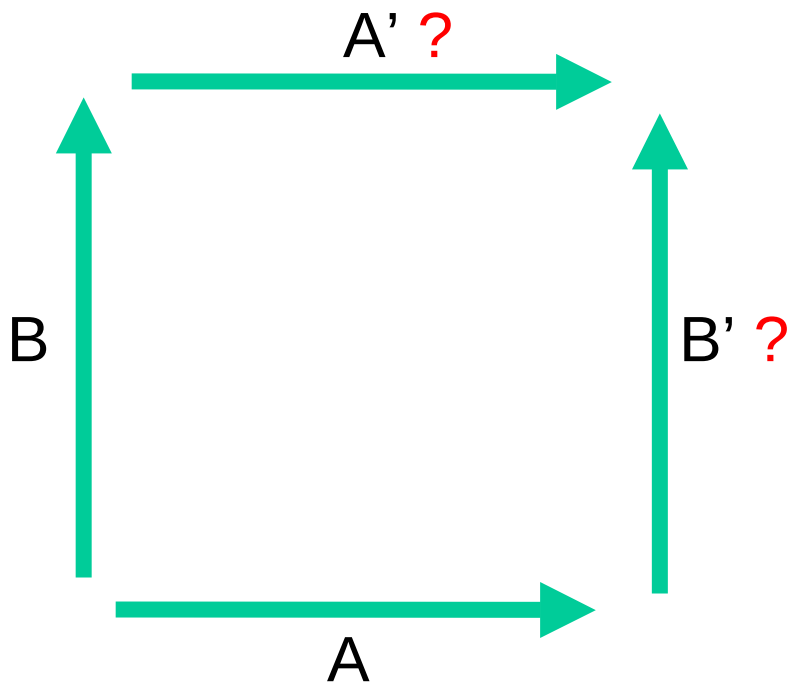


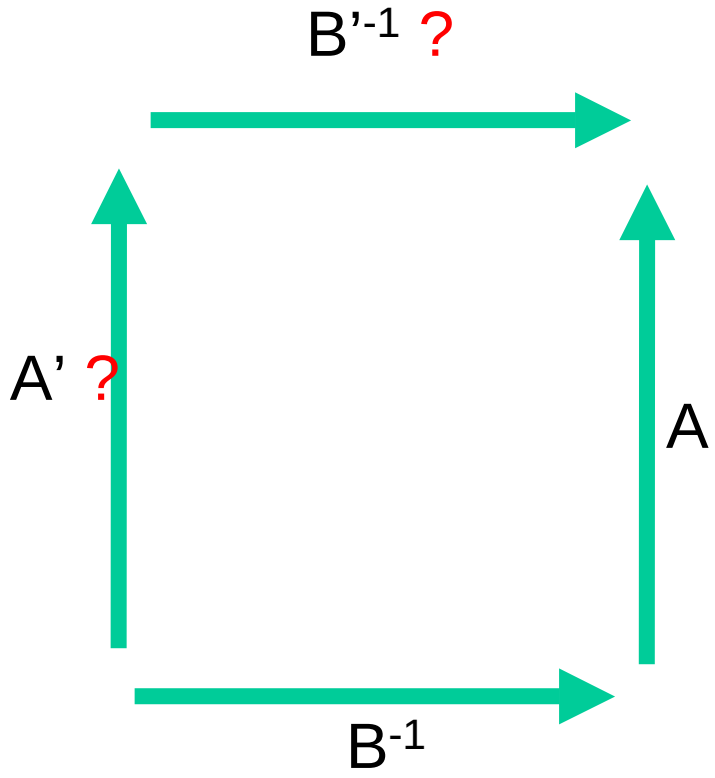
Merging





Merging

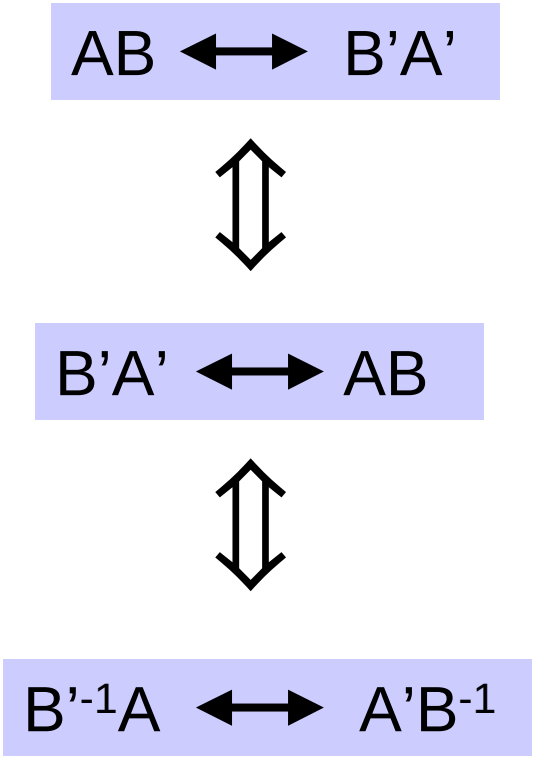




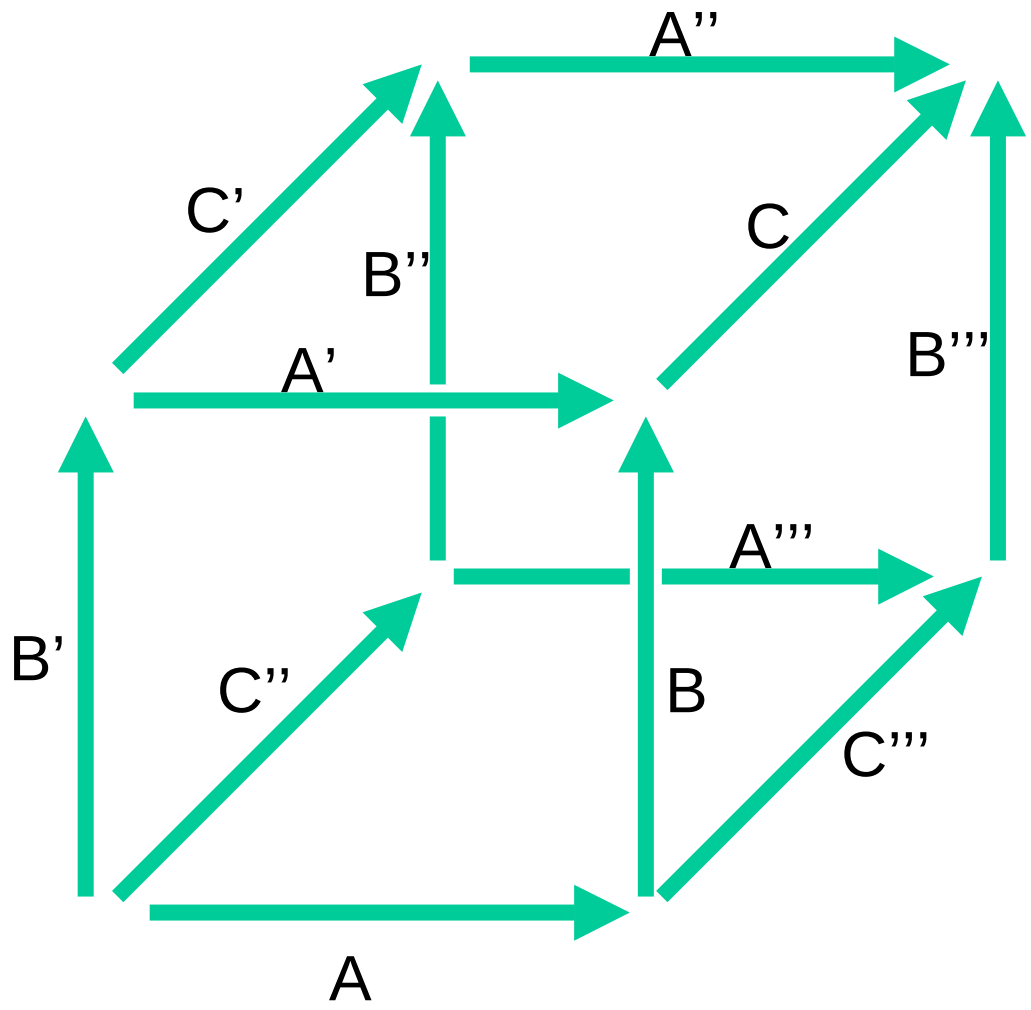
Patch theory

- Set of rules about how patches behave
- Not yet properly understood
  - Very few theorems
- Notation a bit confusing

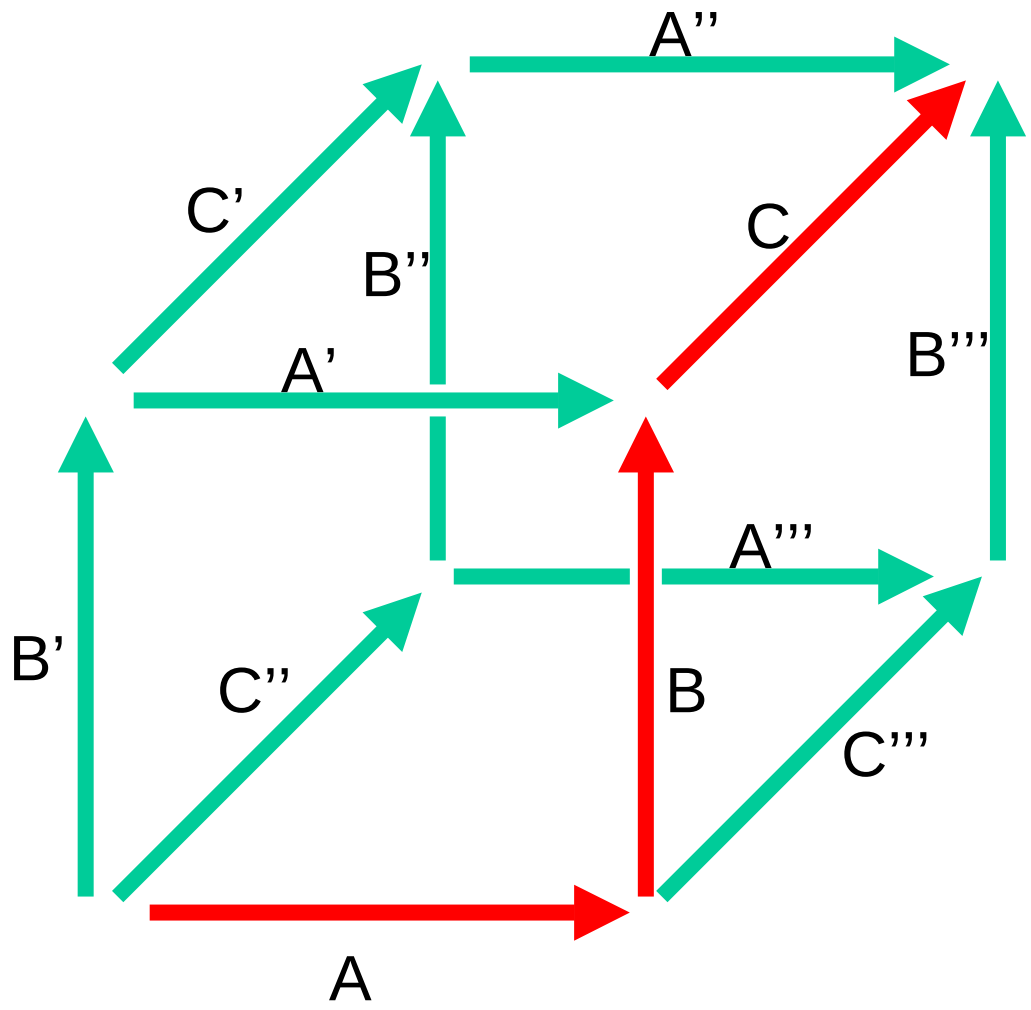
Some basic properties



“Permutivity”

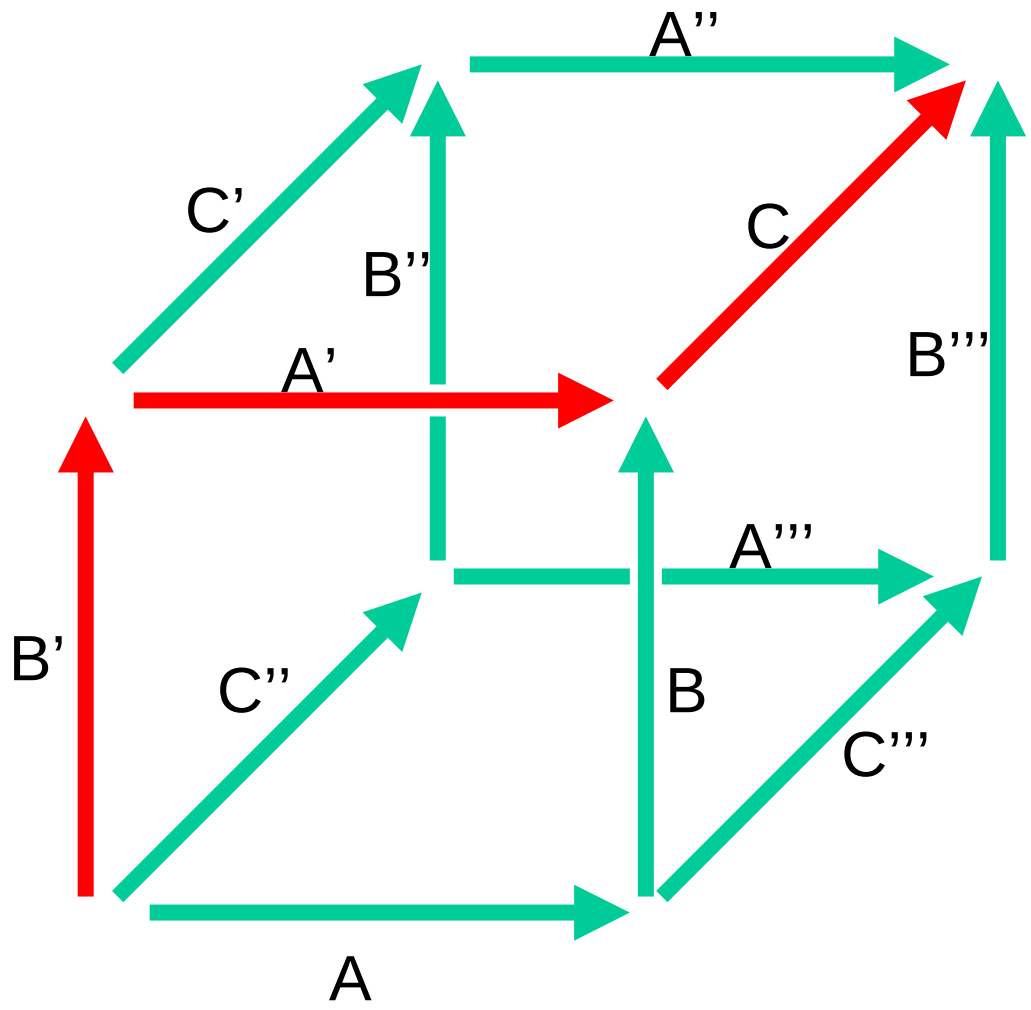


“Permutivity”



ABC

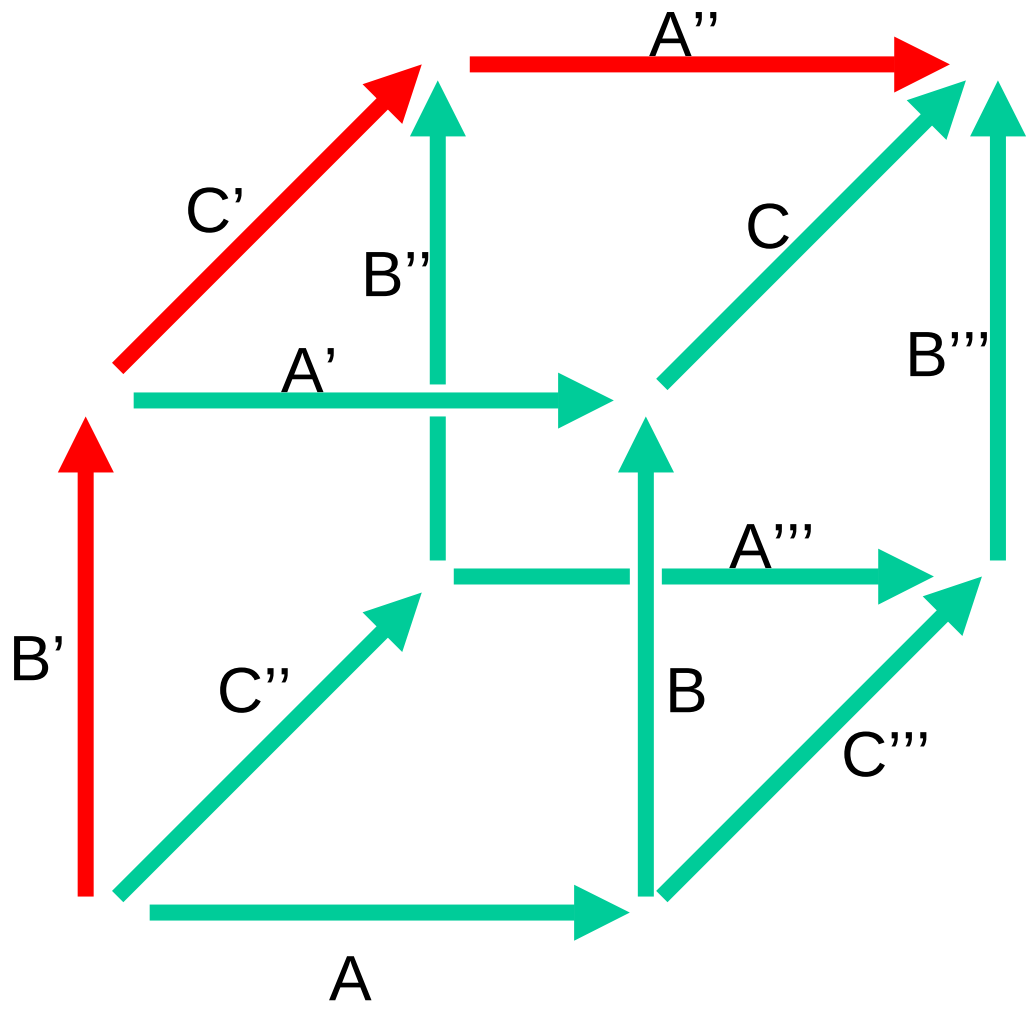
“Permutivity”



ABC

B’A’C

“Permutivity”

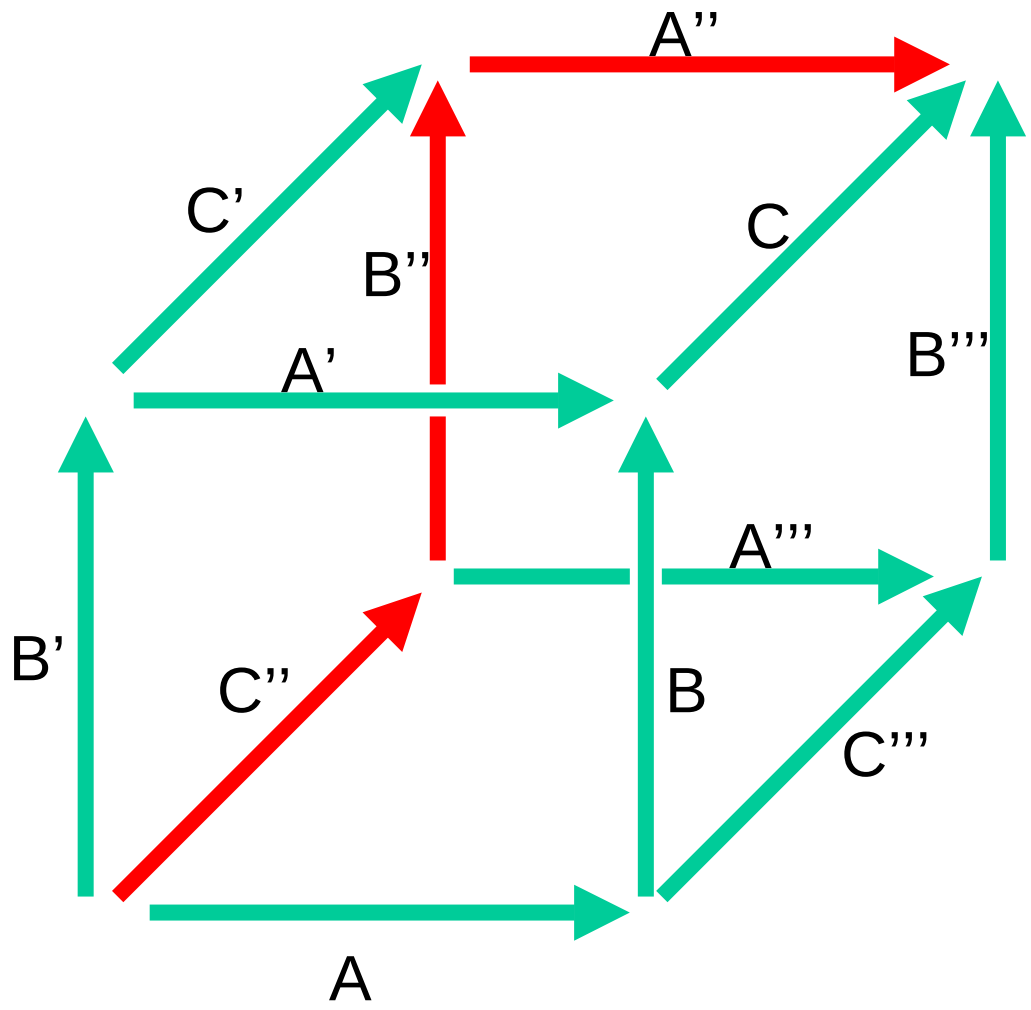


ABC

B’A’C

B’C’A’’

“Permutivity”



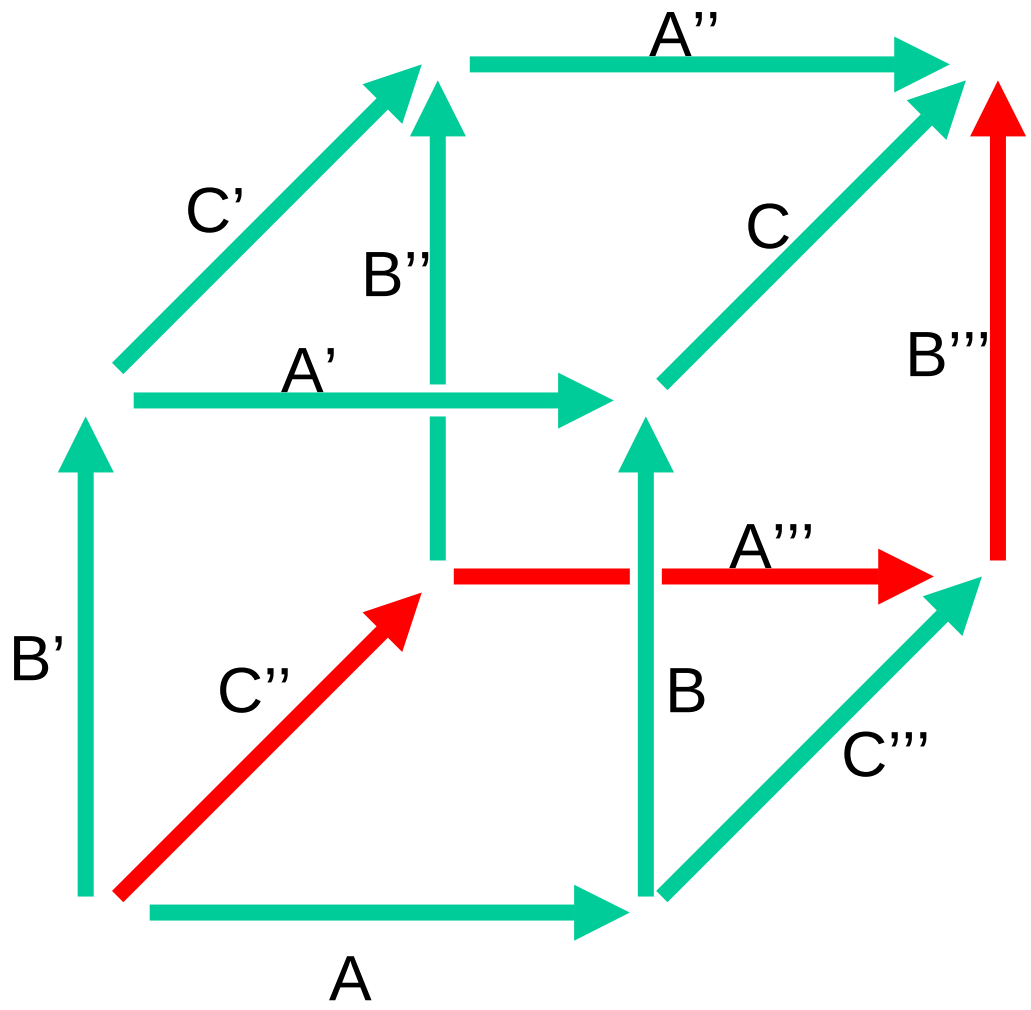
ABC

B’A’C

B’C’A’’

C’’B’’A’’

“Permutivity”



ABC

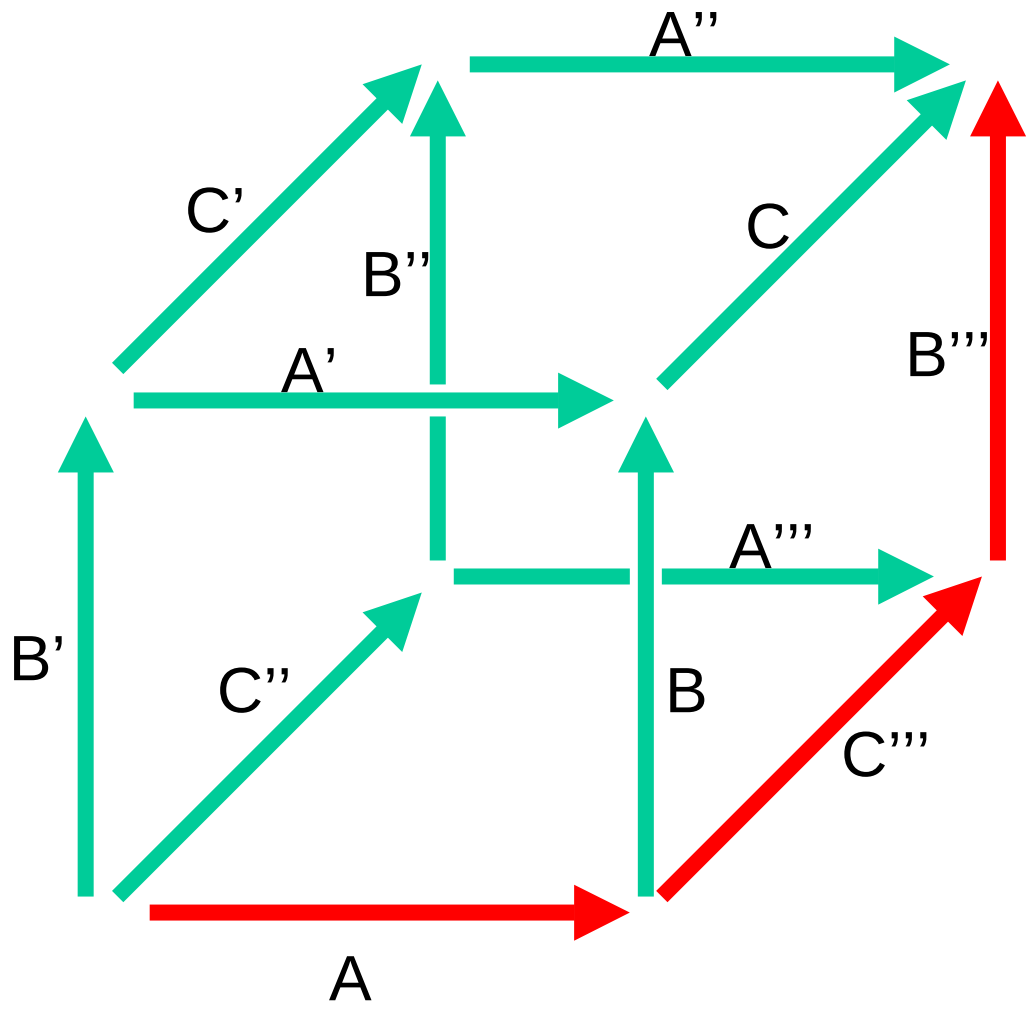
B’A’C

B’C’A’’

C’’B’’A’’

C’’A’’’B’’’

“Permutivity”



ABC

B’A’C

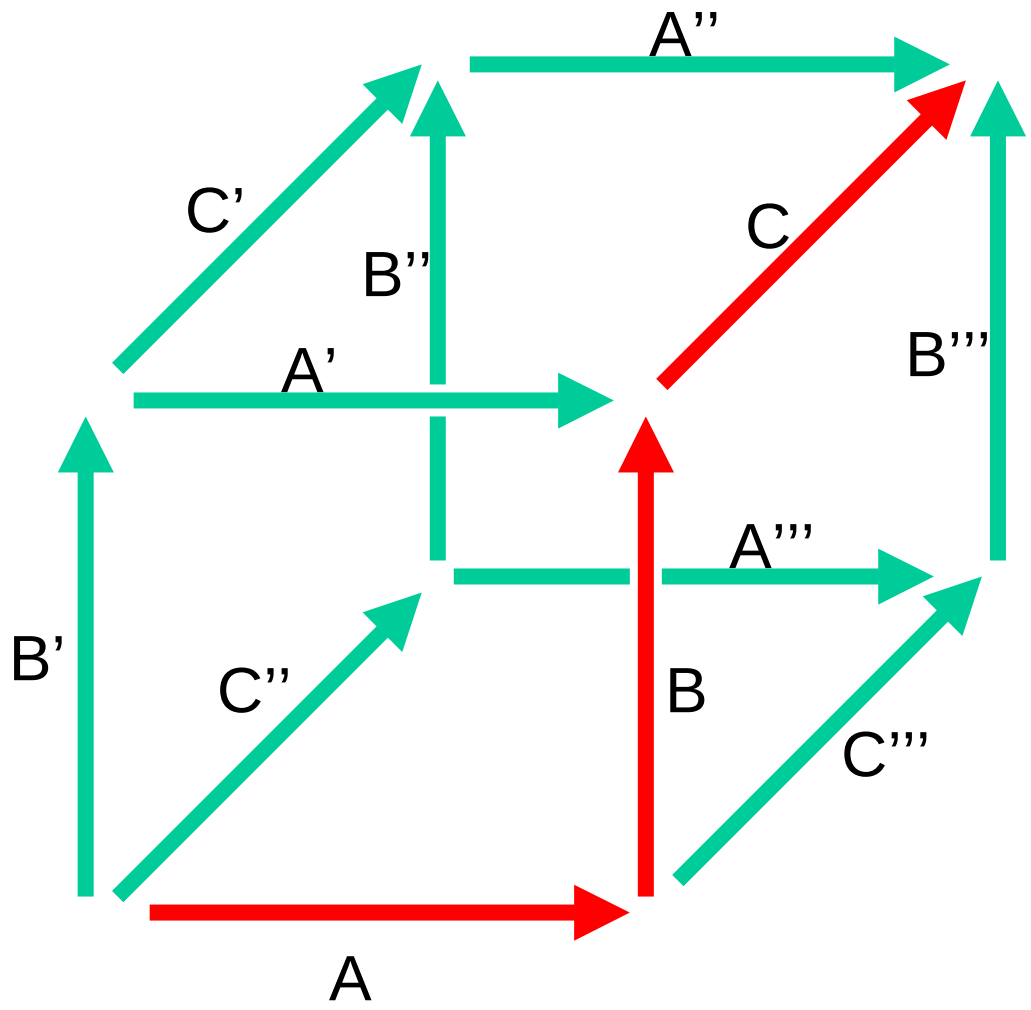
B’C’A’’

C’’B’’A’’

C’’A’’’B’’’

A’’’’C’’’B’’’

“Permutivity”



ABC

B’A’C

B’C’A’’

C’’B’’A’’

C’’A’’’B’’’

A’’’’C’’’B’’’

A’’’’B’’’’C’’’’

Some GHC extensions

ADT

```
data Expr t =
     Const t
   | Times (Expr t) (Expr t)
```

Times (Const ‘a’) (Const ‘b’) ??

Algebraic = unrestricted sums and products

ADT

with new syntax

```
data Expr t where
    Const :: t → Expr t
    Times :: Expr t → Expr t → Expr t
```

Times (Const ‘a’) (Const ‘b’) ??

Algebraic = unrestricted sums and products

GADTs

```
data Expr t where
    Const :: t → Expr t
    Times :: Expr Int → Expr Int → Expr Int
```

Times (Const ‘a’) (Const ‘b’) 

*Generalised algebraic = **restricted** sums and products*

“Existential” types

data Number = forall x . Num x $\Rightarrow$ Number x

Number (5 :: Int)
Number (5.0 :: Double)
…

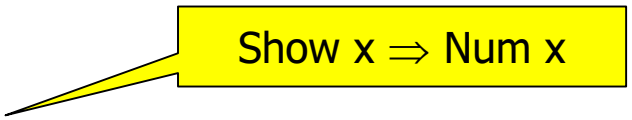
We can construct Number using any x in Num

"Existential" types

data Number = forall x . Num x ⇒ Number x

inc :: Number → Number
inc (Number a) = Number (a+1)

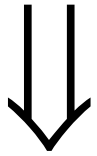
instance Show Number where
  show (Number a) = show a



When we use Number, we only know that x is in Num

Existential types, GADT style

data Number = forall x . Num x $\Rightarrow$ Number x



data Number where

Number :: Num x $\Rightarrow$ x $\rightarrow$ Number

Putting these types to work

Working with patches

AB ⟷ B'A'

```
commute :: Patch → Patch → Maybe (Patch, Patch)

data Patch where
 Null :: Patch
 Seq :: Patch → Patch → Patch
 ...
```

Working with patches

(AB)C ⟷ ?

```
commute :: Patch → Patch → Maybe (Patch, Patch)
commute (Seq a b) c
= do (a', c')  ← commute a c
     (b', c'') ← commute b c'
     return (Seq a' b', c'')
```

Working with patches

(AB)C ⟷ ?

```
commute :: Patch → Patch → Maybe (Patch, Patch)
commute (Seq a b) c
= do (b', c')  ← commute b c
     (a', c'') ← commute a c'
     return (Seq a' b', c'')
```

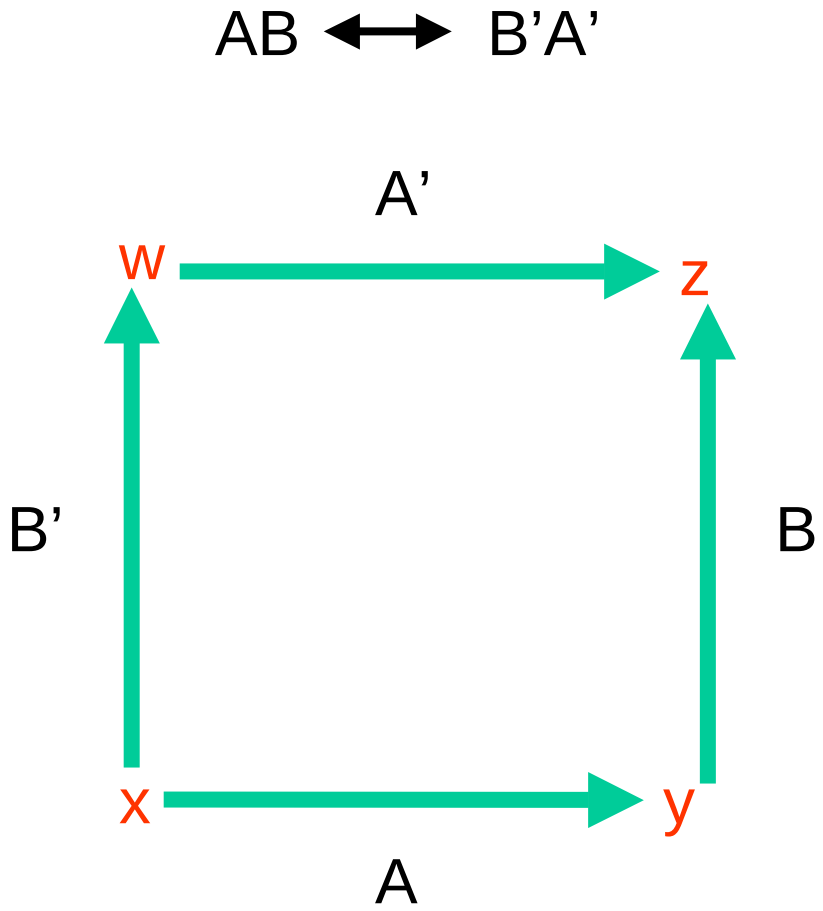
GADT

```
data Patch x y where
 Null :: Patch x x
 Seq :: Patch x y → Patch y z → Patch x z
 …

commute :: Patch x y → Patch y z
                → Maybe (Patch x w, Patch w z)
```

Phantom types represent the repository state

What's going on?

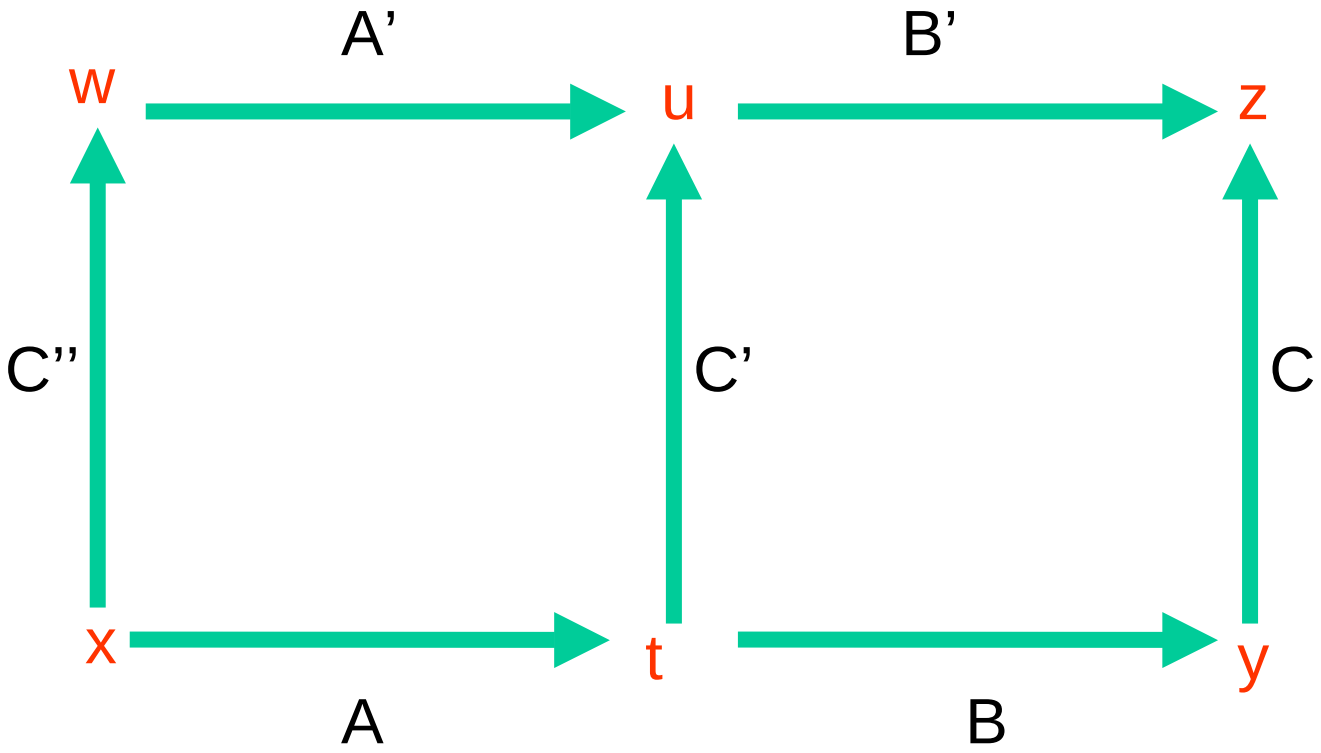


commute ::
Patch x y →
Patch y z →

Maybe
(Patch x w,
Patch w z)

Safer commute

commute :: Patch x y → Patch y z
→ Maybe (Patch x w, Patch w z)



So is it actually useful?

- Upcoming darcs 2.0 release
  - Rewrite of conflict handling code
  - Conditionally compiled with GADTs
  - Very positive influence:
    - Conversion process found at least one bug in existing code
    - “very valuable in exploratory development”

Limitations

- There have to be “unsafe” operations in some places
  - e.g. when constructing “primitive” patches
  - Cordon those off to a small number of cases, and use comments to explain ourselves
- Working with GADTs can be painful
  - Extra type signatures
  - Confusing error messages

The end

Sealing

```
commute :: Patch x y → Patch y z
        → Maybe (Patch x w, Patch w z)
```

Sealing

```
commute :: forall x y w z
         . Patch x y → Patch y z
         → Maybe (Patch x w, Patch w z)
```

Sealing

```
commute :: forall x y w z
                . Patch x y → Patch y z
               → Maybe (Patch x w, Patch w z)

data CommutePair x y where
   CommutePair :: Patch x y → Patch y z
                  → Patch x z
```

Sealing

```
commute :: forall x z
                . CommutePair x z
               → Maybe (CommutePair x z)

data CommutePair x y where
   CommutePair :: Patch x y → Patch y z
                           → Patch x z
```

Data structures

```
data FL p x y where
   NilFL :: FL p x x
   ConsFL :: FL p x y → FL p y z → FL p x z

data Tree p x where
   NilTree :: Tree x
   SeqTree :: p x y → Tree y → Tree x
   ParTree :: Tree x → Tree x → Tree x
```

DAGs

- Based on Martin Erwig's inductive graphs