Git as a HIT

Dan Licata
Wesleyan University

Darcs

~~Git~~ as a HIT

Dan Licata
Wesleyan University

HITs

HITs

- *Homotopy Type Theory* is an extension of Agda/Coq based on connections with homotopy theory

  [Hofmann&Streicher,Awodey&Warren,Voevodsky,Lumsdaine,Garner&van den Berg]

HITs

- *Homotopy Type Theory* is an extension of Agda/Coq based on connections with homotopy theory

  [Hofmann&Streicher,Awodey&Warren,Voevodsky,Lumsdaine,Garner&van den Berg]

- *Higher inductive types (HITs)* are a new type former!

HITs

- *Homotopy Type Theory* is an extension of Agda/Coq based on connections with homotopy theory

  [Hofmann&Streicher,Awodey&Warren,Voevodsky,Lumsdaine,Garner&van den Berg]

- *Higher inductive types (HITs)* are a new type former!

- They were originally invented[Lumsdaine,Shulman,…] to model basic spaces (circle, spheres, the torus, …) and constructions in homotopy theory

HITs

- *Homotopy Type Theory* is an extension of Agda/Coq based on connections with homotopy theory

  [Hofmann&Streicher,Awodey&Warren,Voevodsky,Lumsdaine,Garner&van den Berg]

- *Higher inductive types (HITs)* are a new type former!

- They were originally invented[Lumsdaine,Shulman,…] to model basic spaces (circle, spheres, the torus, …) and constructions in homotopy theory

- But they have many other applications, including some programming ones!

Patches

diff

a
b
c

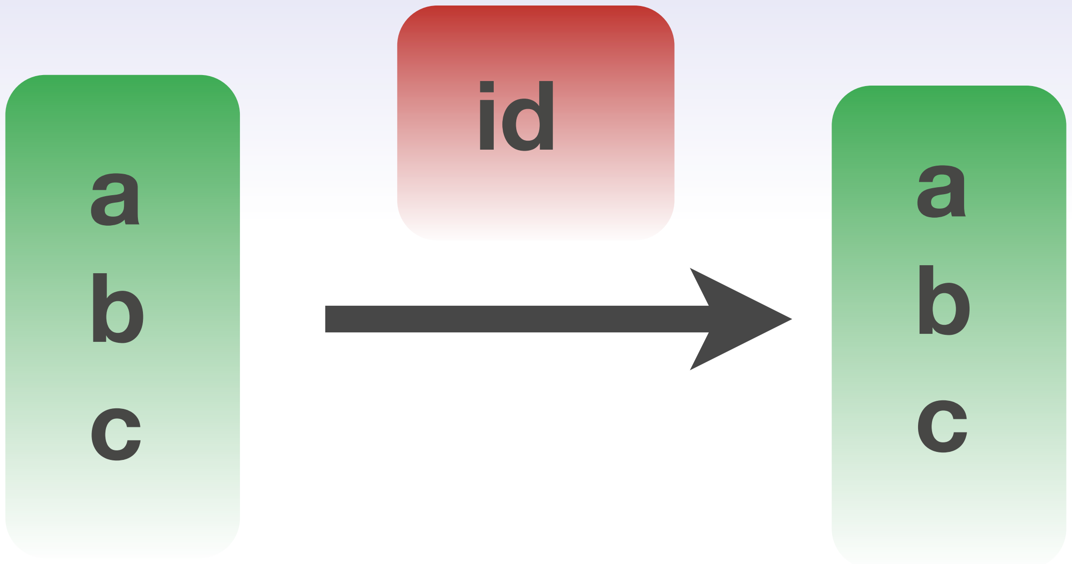
a
d
c

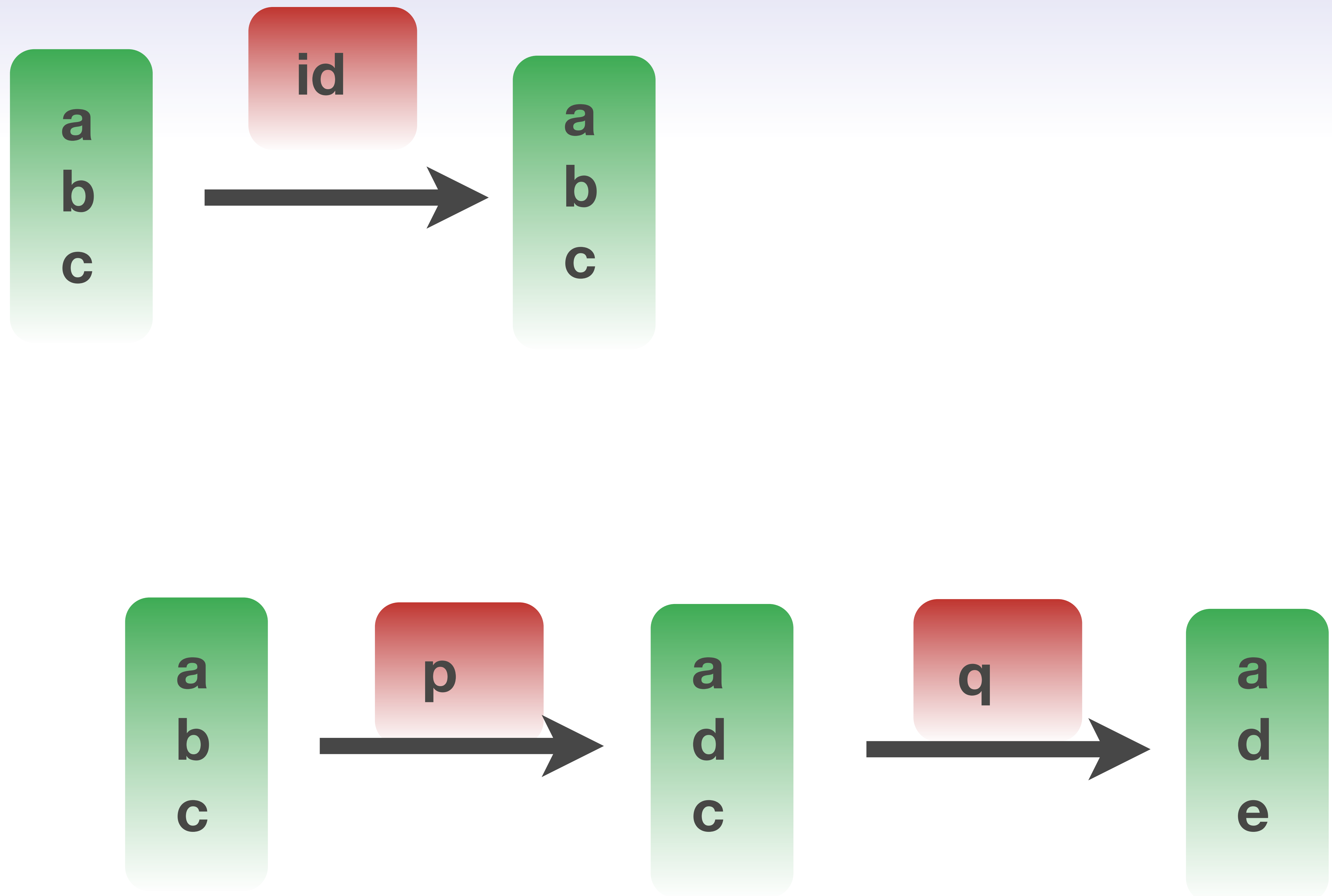
=

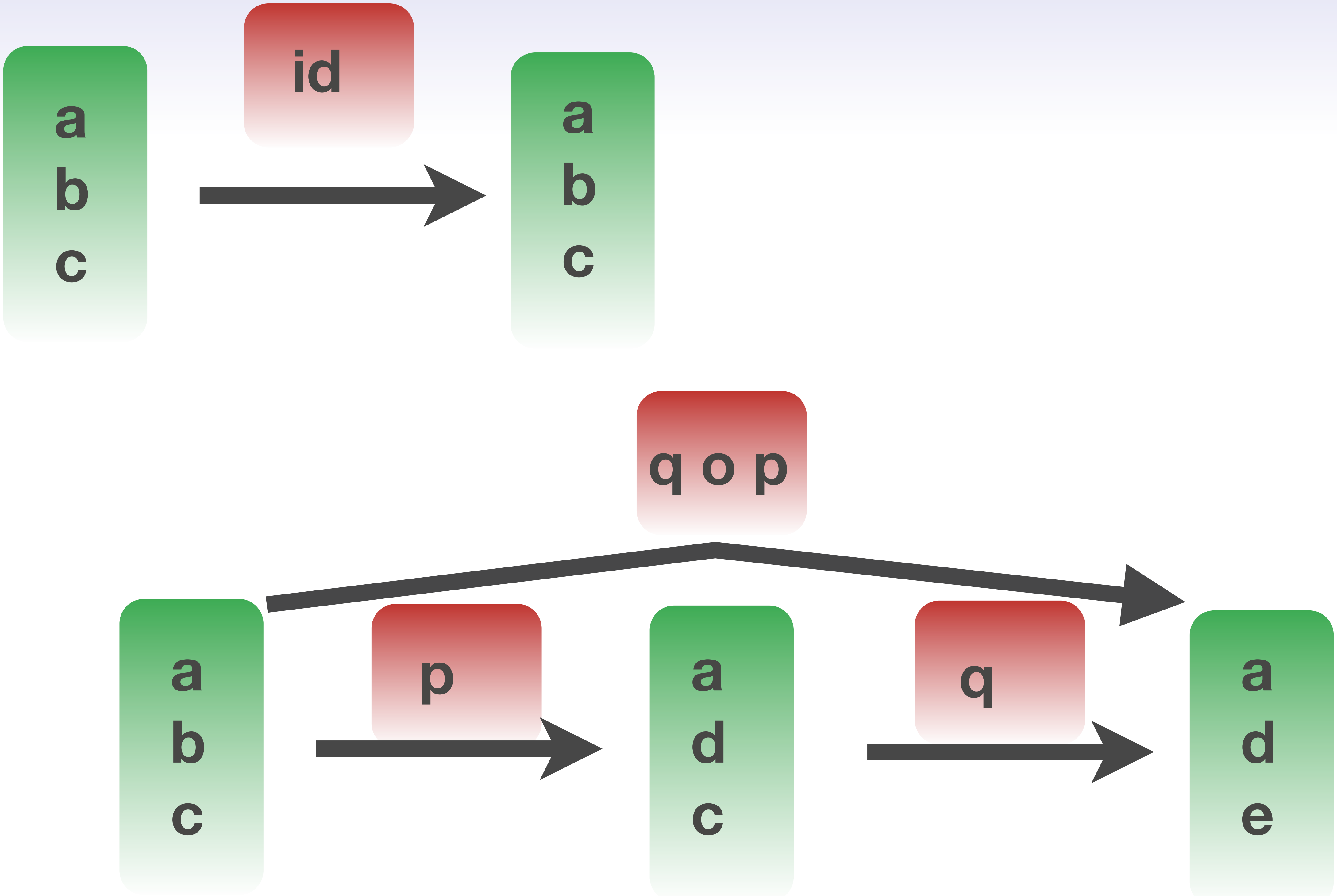
Patch

2c2
< b

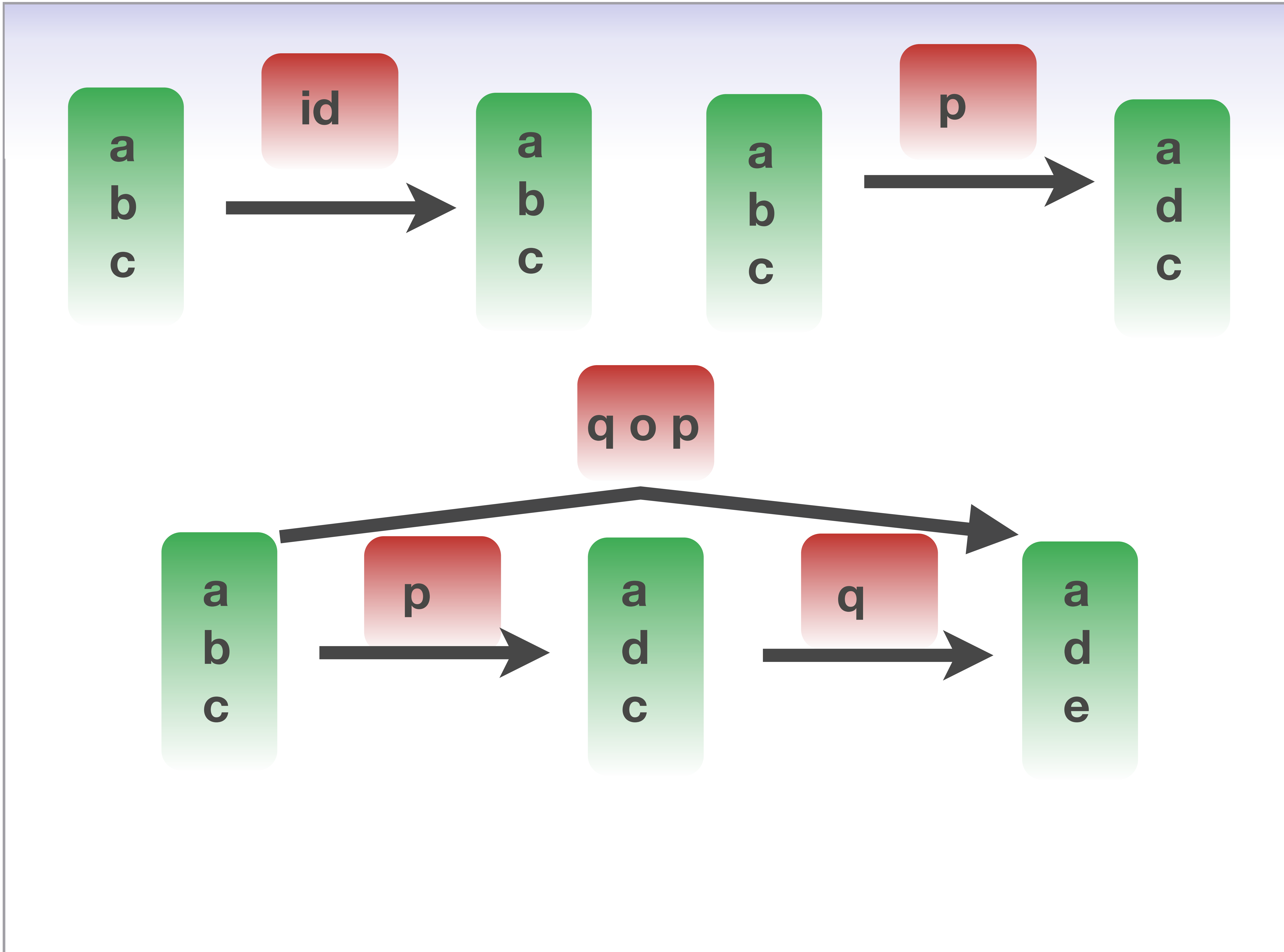
> d

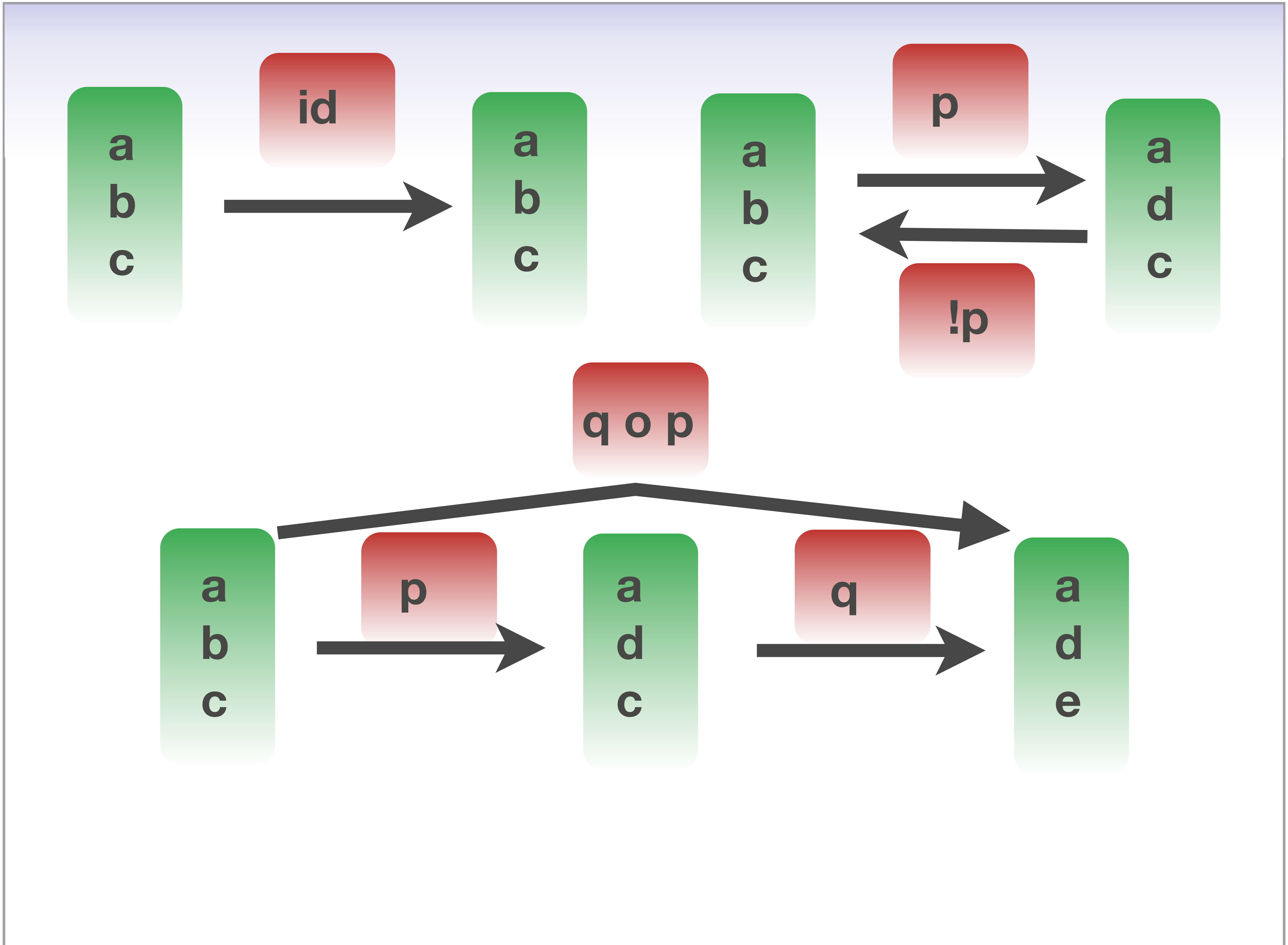

a
b
c
id
a
b
c

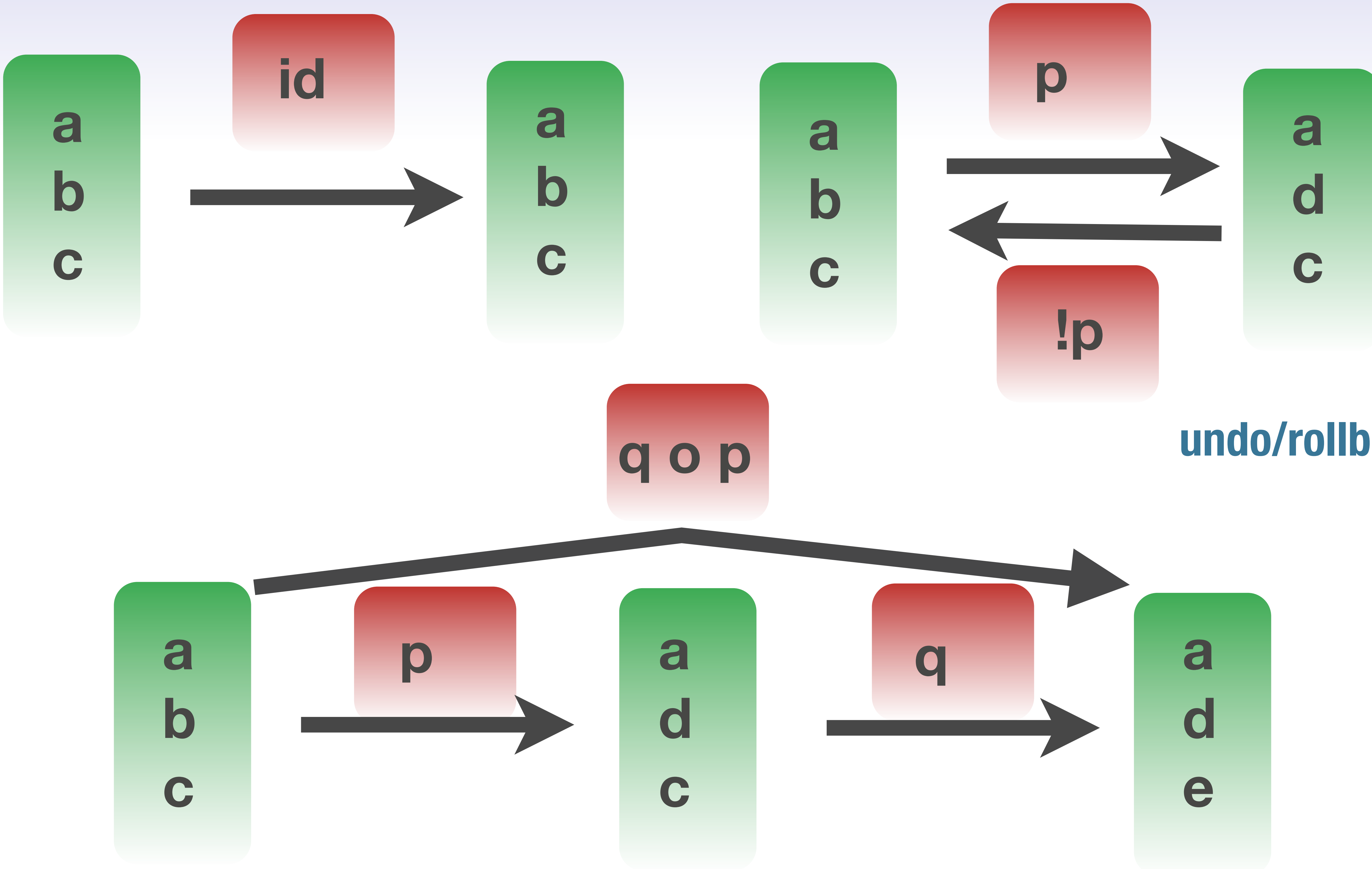

a
b
c
id
a
b
c
a
b
c
p
a
d
c
q
a
d
e



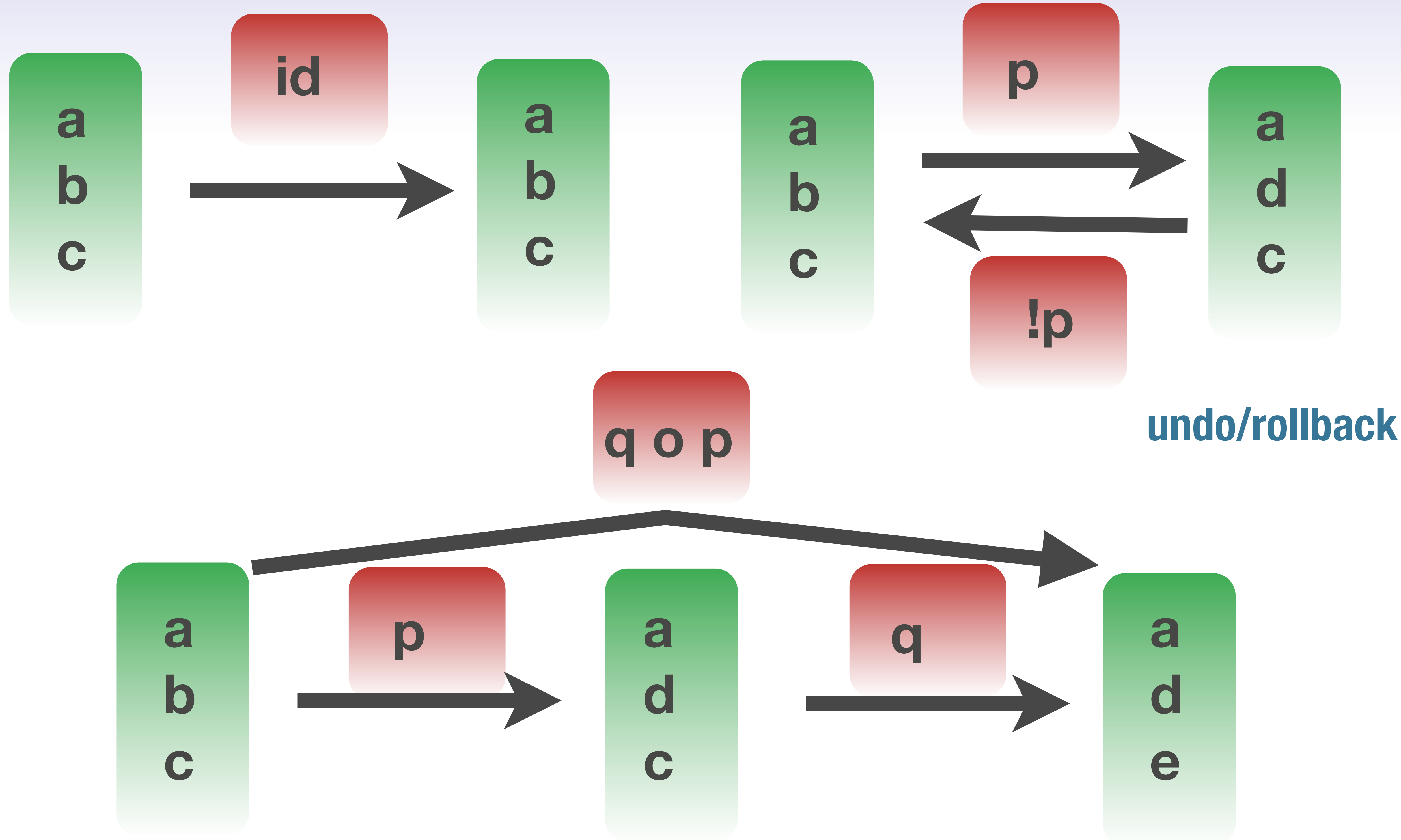
a
b
c
id
a
b
c
q o p
a
b
c
p
a
d
c
q
a
d
e


a
b
c
id
a
b
c
a
b
c
p
a
d
c
q o p
a
b
c
p
a
d
c
q
a
d
e


a
b
c
id
a
b
c
a
b
c
p
!p
a
d
c
q o p
a
b
c
p
a
d
c
q
a
d
e



undo/rollback



[Yorgey,Jacobson,…]

Simple Setup

a	u	s	s	o	i	s	

a ↔ b at 0

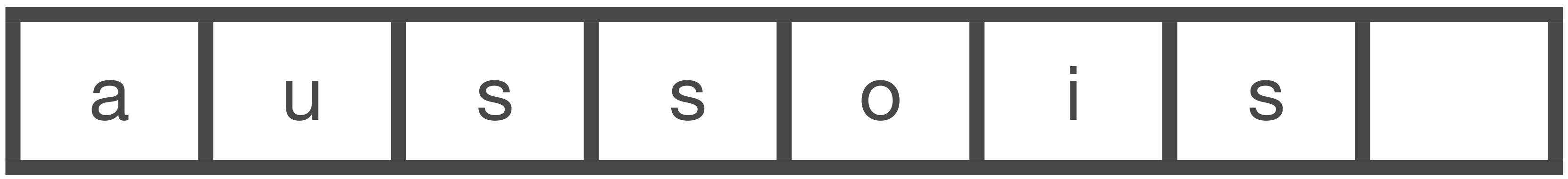
a	u	s

b	u	s

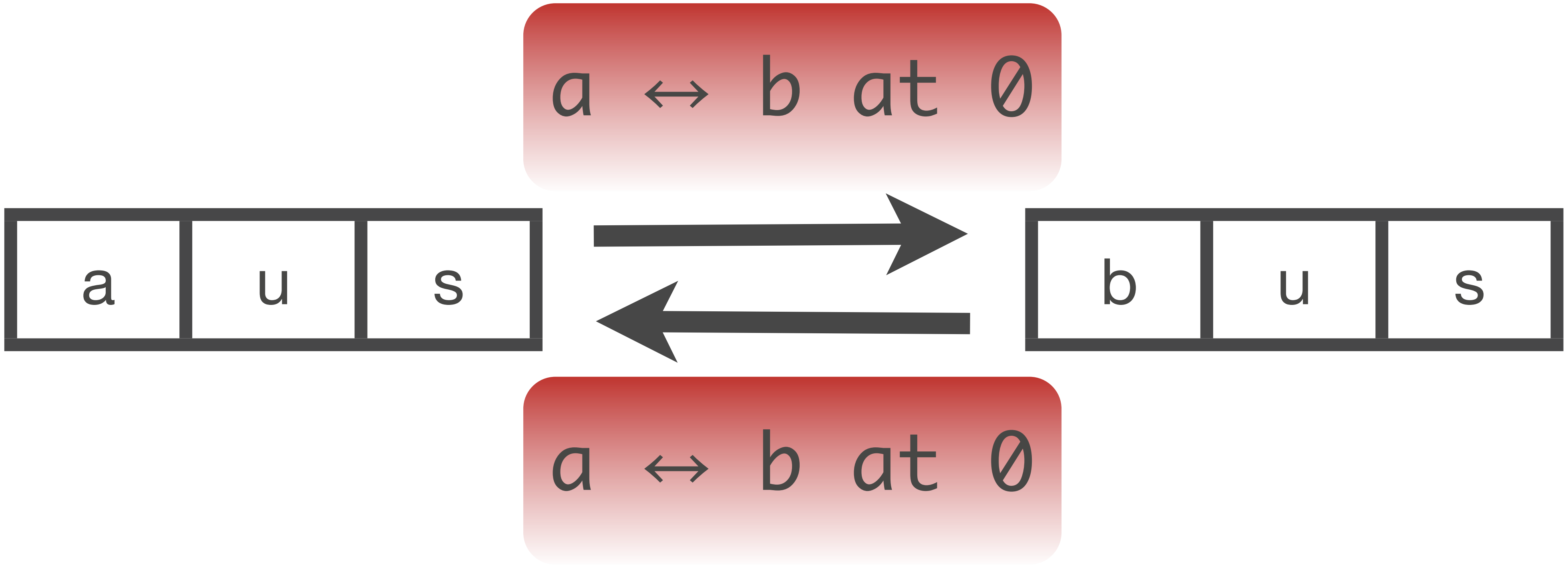
a ↔ b at 0

Simple Setup

- “Repository” is a char vector of fixed length n



- Basic patch is `a ↔ b at i` where `i<n`



Domain-Specific Language

```
data Patch : Set where
  id       : Patch
  _∘_      : Patch → Patch → Patch
  !        : Patch → Patch
  _↔_at_   : Char → Char → Fin n → Patch
```

Domain-Specific Language

```
interp : Patch → (Vec Char n → Vec Char n) ×
                 (Vec Char n → Vec Char n)
interp id = (λ x → x) , (λ x → x)
interp (q ∘ p) = fst (interp q) o fst (interp p) ,
                 snd (interp p) o snd (interp q)
interp (! p) = snd (interp p) , fst (interp p)
interp (a ↔ b at i) = swapat a b i , swapat a b i
```

Domain-Specific Language

```
interp : Patch → (Vec Char n → Vec Char n) ×
                 (Vec Char n → Vec Char n)
interp id = (λ x → x) , (λ x → x)
interp (q ∘ p) = fst (interp q) o fst (interp p) ,
                 snd (interp p) o snd (interp q)
interp (! p) = snd (interp p) , fst (interp p)
interp (a ↔ b at i) = swapat a b i , swapat a b i
```

`swapat a b i v` **permutes** `a` **and** `b` **at position** `i` **in** `v`

Domain-Specific Language

Spec: `∀ p. interp p` is a bijection:

```
  ∀ v. g (f v) = v    where (f,g)=interp p
  ∀ v. f (g v) = v
```

Domain-Specific Language

undo really un-does

Spec: `∀ p. interp p` is a bijection:

```
  ∀ v. g (f v) = v    where (f,g)=interp p
  ∀ v. f (g v) = v
```

Domain-Specific Language

undo really un-does

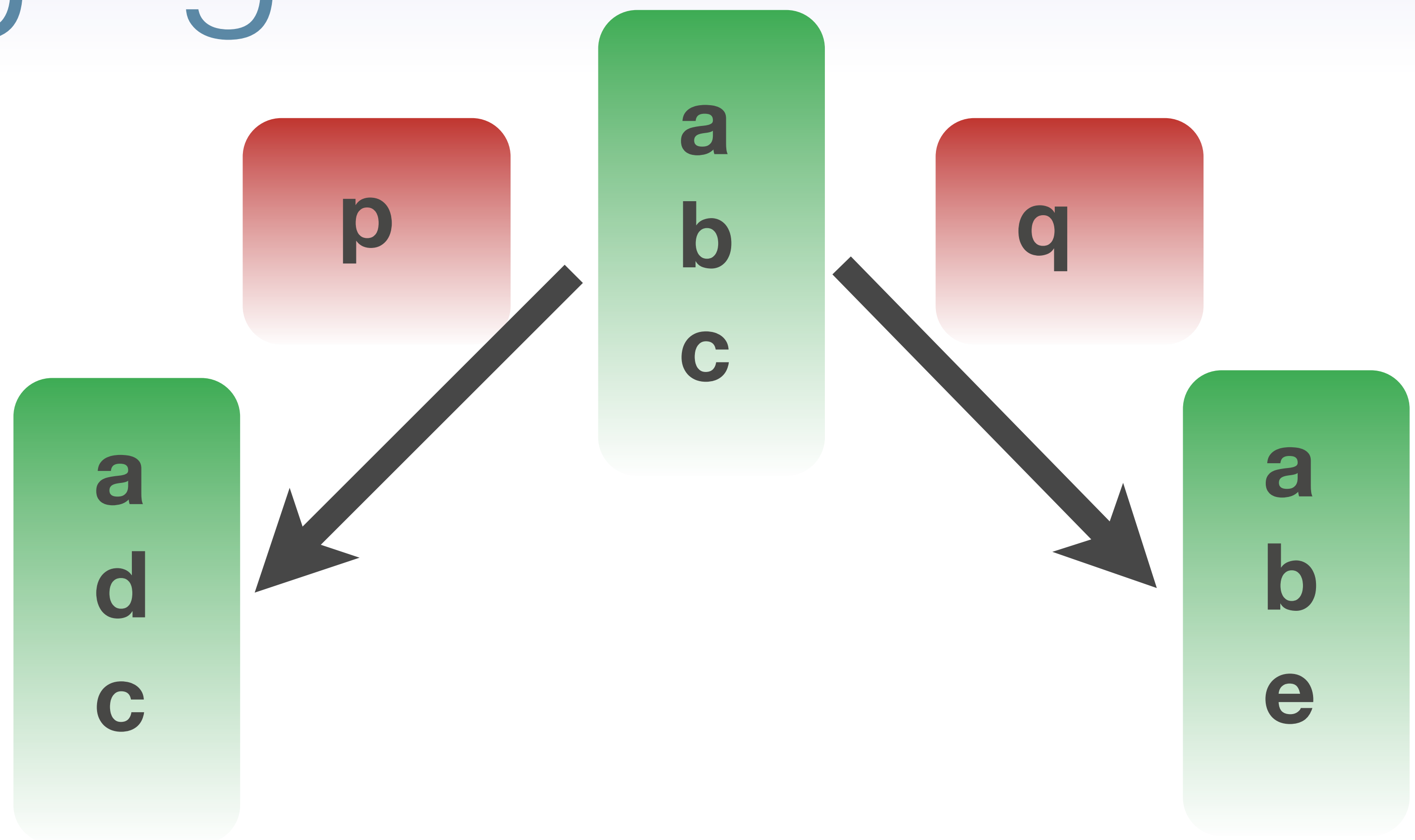
Spec: `∀ p. interp p` is a bijection:

```
∀ v. g (f v) = v    where (f,g)=interp p
∀ v. f (g v) = v
```

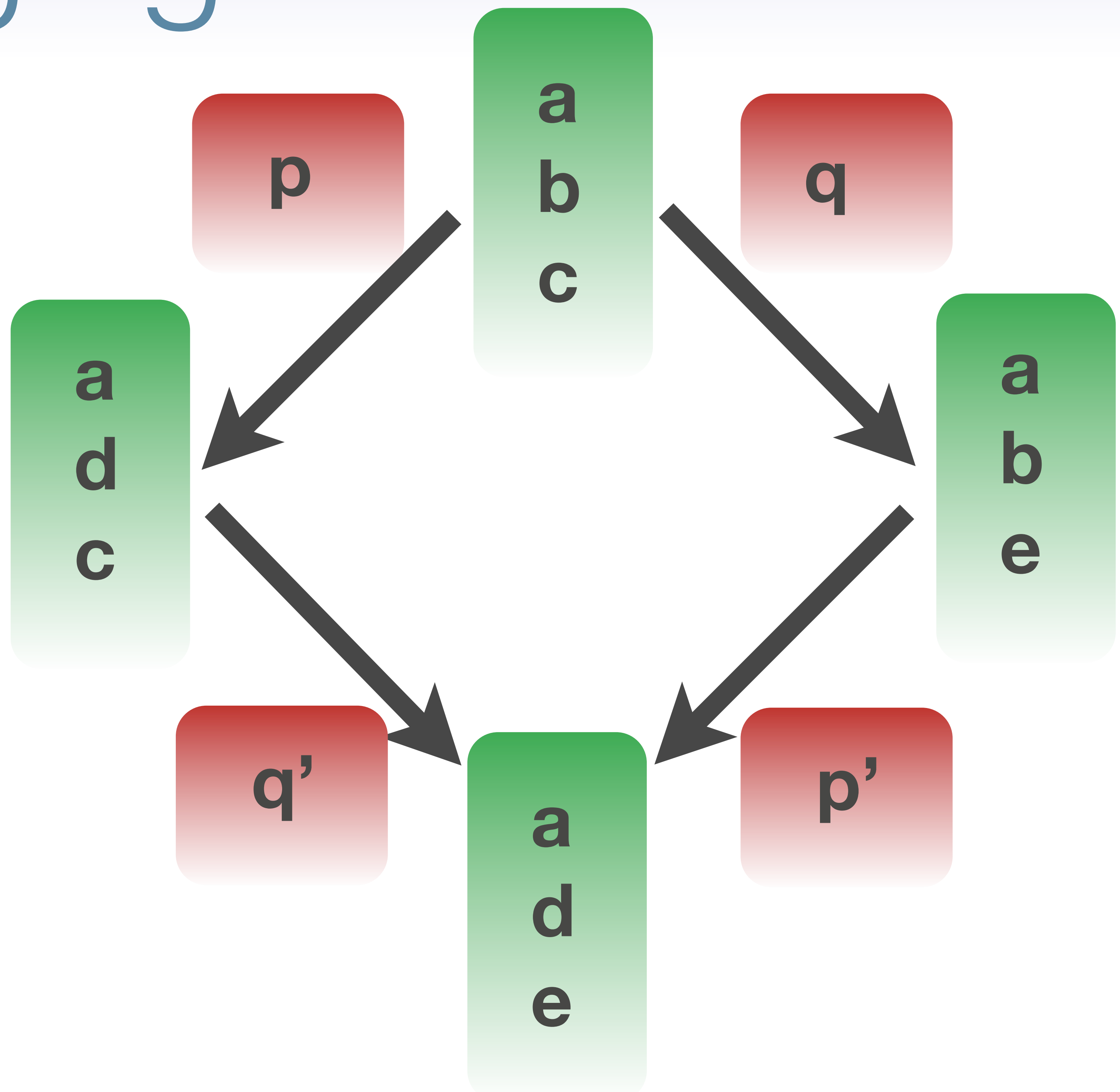
Can package this as:

```
interp : Patch →
         Bijection (Vec Char n) (Vec Char n)
```

Merging



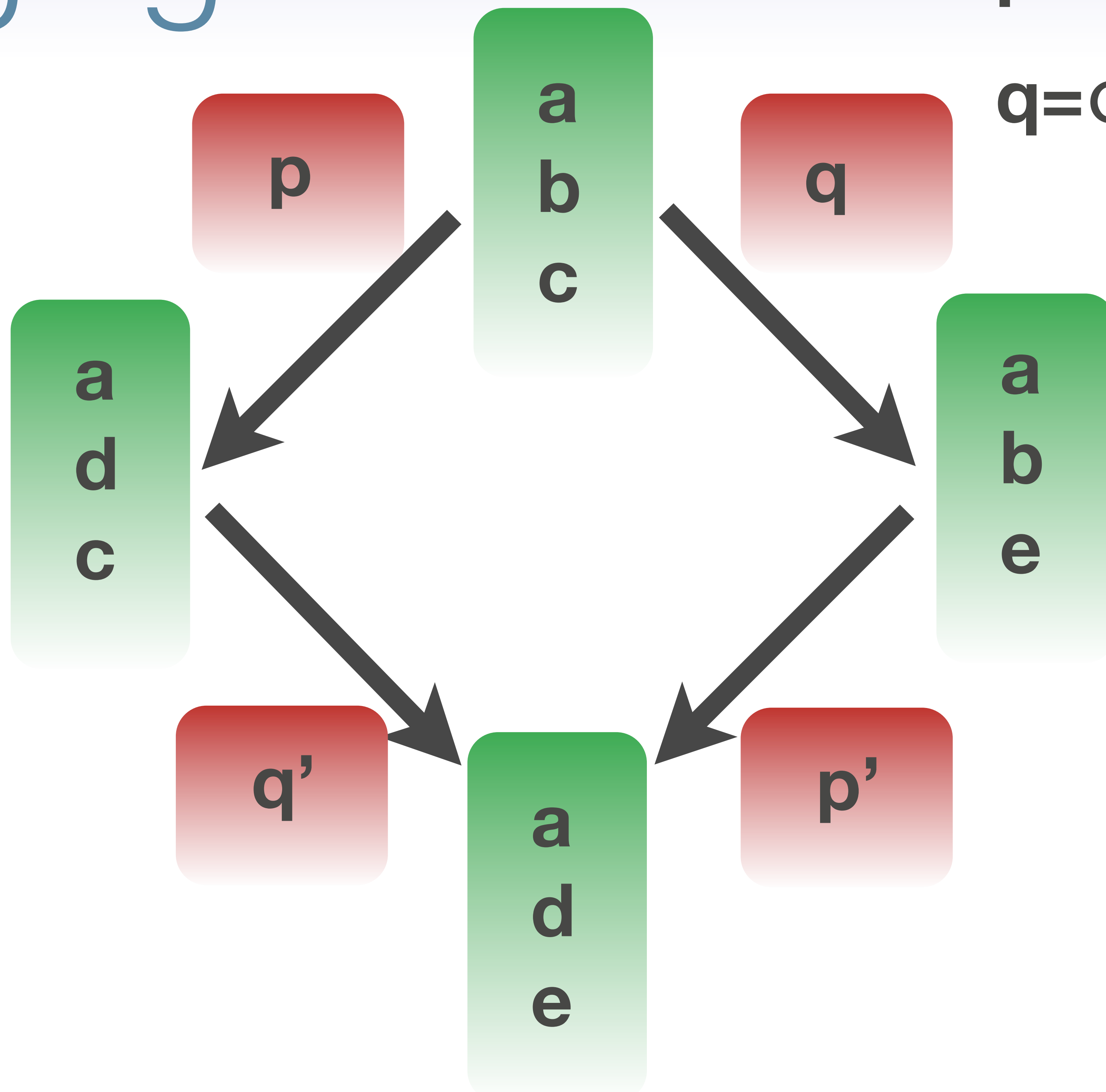
Merging



Merging

p=b↔d at 1

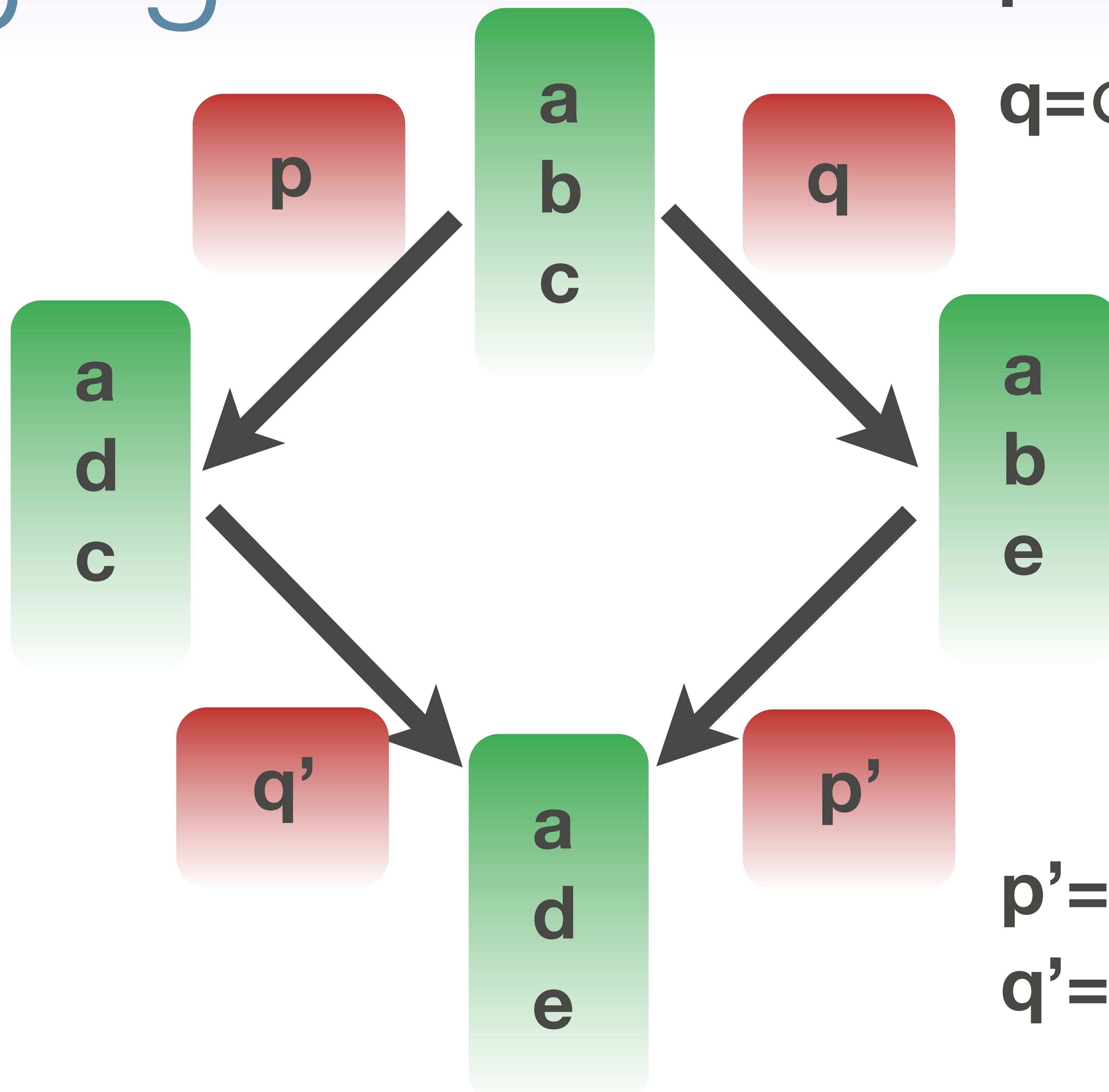
q=c↔e at 2



Merging

`p=b↔d at 1`

`q=c↔e at 2`



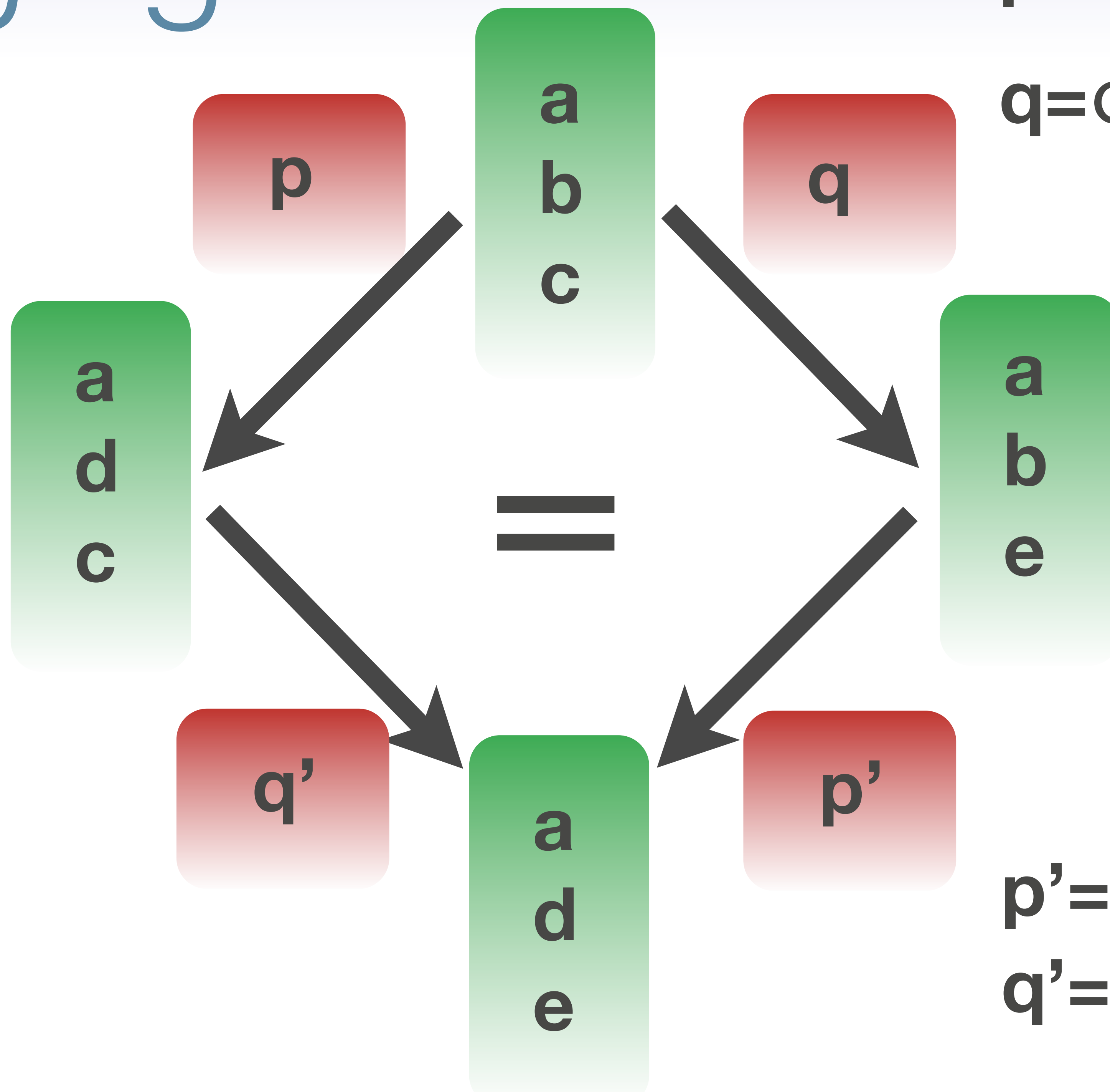
p'=p

q'=q

Merging

p=b↔d at 1

q=c↔e at 2

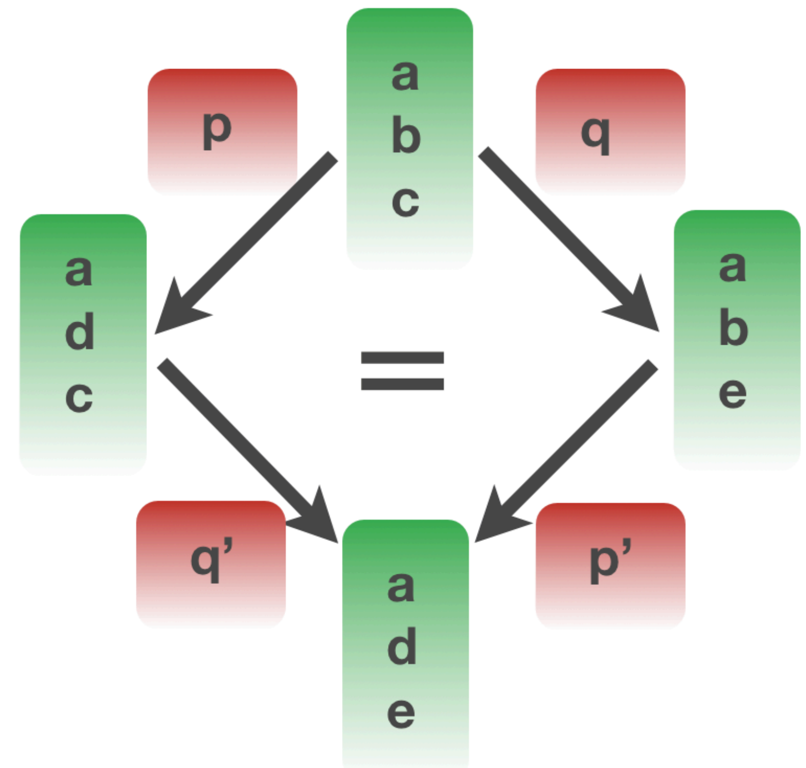


p’=p

q’=q

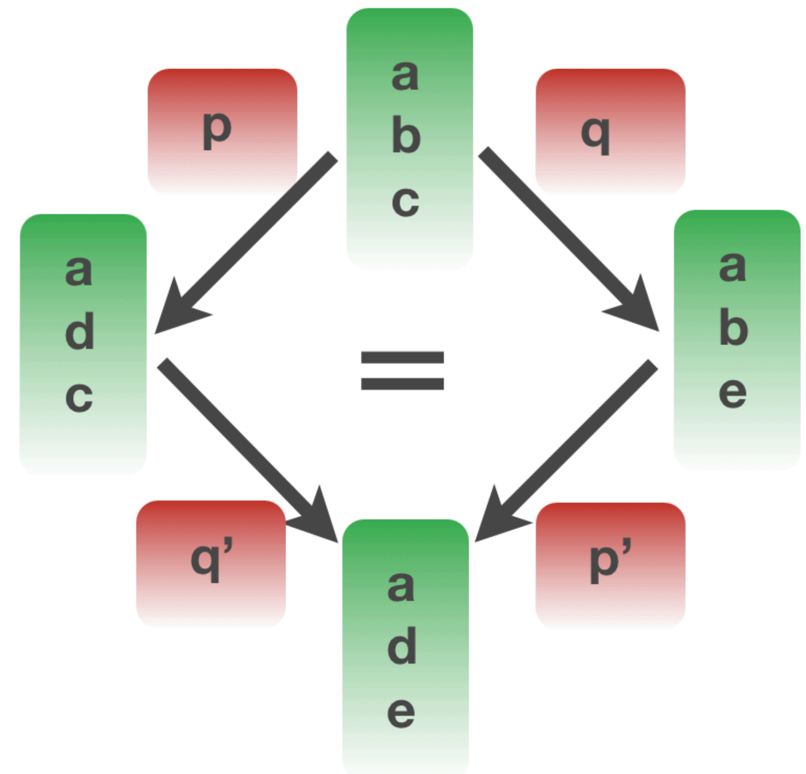
Merging

```
merge : (p q : Patch)
      → Σq’,p’:Patch.
        Maybe(q’ o p =
              p’ o q)
```



Merging

```
merge : (p q : Patch)
      → Σq’,p’:Patch.
        Maybe(q’ o p =
              p’ o q)
```



When are two patches equal?

Patch Equality

```
(a↔b at i)o(c↔d at j) =
    (c↔d at j)o(a↔b at i) if i≠j
```

Patch Equality

```
(a↔b at i)o(c↔d at j) =
    (c↔d at j)o(a↔b at i) if i≠j
(a↔a at i) = id
!(a↔b at i) = (a↔b at i)
(a↔b at i) = (b↔a at i)
```

Patch Equality

Basic Axioms:

```
(a↔b at i)o(c↔d at j) =
    (c↔d at j)o(a↔b at i) if i≠j

(a↔a at i) = id

!(a↔b at i) = (a↔b at i)

(a↔b at i) = (b↔a at i)
```

Patch Equality

Basic axioms:

```
(a↔b at i)o(c↔d at j)
=(c↔d at j)o(a↔b at i)
```

Patch Equality

Basic axioms:

```
(a↔b at i)o(c↔d at j)
=(c↔d at j)o(a↔b at i)
```

Group laws:

```
id o p = p = p o id
po(qor) = (poq)or
!p o p = id = p o !p
```

Patch Equality

Basic axioms:

`(a↔b at i)o(c↔d at j)`

`=(c↔d at j)o(a↔b at i)`

Congruence:

`p=p`

`p=q if q=p`

`p=r if p=q and q=r`

Group laws:

`id o p = p = p o id`

`po(qor) = (poq)or`

`!p o p = id = p o !p`

`!p = !p’ if p = p’`

`p o q = p’ o q’ if`
`    p = p’ and q = q’`

Patch as Quotient Type

Elements:

```
data Patch' : Set where
  id      : Patch'
  _∘_     : Patch' → Patch' → Patch'
  !       : Patch' → Patch'
  _↔_at_ : Char → Char → Fin n → Patch'
```

Equality:

```
(a↔b at i)o(c↔d at j)~
   (c↔d at j)o(a↔b at i)

...
id o p ~ p ~ p o id

po(qor) ~ (poq)or

!p o p ~ id ~ p o !p

p~p

p~q if q~p

p~r if p~q and q~r
!p ~ !p' if p ~ p'
p o q ~ p' o q' if p ~ p' and q ~ q'
```

Patch as Quotient Type

Elements:

```
data Patch' : Set where
  id      : Patch'
  _∘_     : Patch' → Patch' → Patch'
  !       : Patch' → Patch'
  _↔_at_ : Char → Char → Fin n → Patch'
```

Equality:

```
(a↔b at i)o(c↔d at j)~
   (c↔d at j)o(a↔b at i)

...
id o p ~ p ~ p o id

po(qor) ~ (poq)or

!p o p ~ id ~ p o !p

p~p

p~q if q~p

p~r if p~q and q~r
!p ~ !p’ if p ~ p’
p o q ~ p’ o q’ if p ~ p’ and q ~ q’
```

Quotient Type:

```
Patch := Patch’/~
```

Patch as Quotient Type

Elements:

```
data Patch' : Set where
  id      : Patch'
  _∘_     : Patch' → Patch' → Patch'
  !       : Patch' → Patch'
  _↔_at_  : Char → Char → Fin n → Patch'
```

Equality:

```
(a↔b at i)o(c↔d at j)~
   (c↔d at j)o(a↔b at i)

...
id o p ~ p ~ p o id

po(qor) ~ (poq)or

!p o p ~ id ~ p o !p

p~p

p~q if q~p

p~r if p~q and q~r

!p ~ !p’ if p ~ p’

p o q ~ p’ o q’ if p ~ p’ and q ~ q’
```

Quotient Type:

```
Patch := Patch’/~
```

Elimination rule:

```
interp : Patch →
         Bijection (Vec Char n) (Vec Char n)
```

define on `Patch’` as before,

then prove p ~ q implies

```
interp p = interp q
```

for all 14+ rules for ~

Patches as a HIT

1. How do you define `Patch` using a higher inductive type?

2. What is the elimination rule?

3. How do you use the elim. rule to define `interp`?

Patches as a HIT

1. **How do you define `Patch` using a higher inductive type?**
2. What is the elimination rule?
3. How do you use the elim. rule to define `interp`?

15

Higher Inductive Type

Higher Inductive Type

Type freely generated by constructors for elements, equalities, equalities between equalities, …

Higher Inductive Type

Type freely generated by constructors for elements, equalities, equalities between equalities, …

```
RepoDesc : Type
```

Higher Inductive Type

Type freely generated by constructors for elements, equalities, equalities between equalities, …

```
RepoDesc : Type

vec : RepoDesc
```

generator for element

Higher Inductive Type

Type freely generated by constructors for elements, equalities, equalities between equalities, …

```
RepoDesc : Type

vec : RepoDesc
```

generator for element

```
(a↔b at i) : vec = vec
```

generator for equality

Higher Inductive Type

Type freely generated by constructors for elements, equalities, equalities between equalities, …

```
RepoDesc : Type

vec : RepoDesc

(a↔b at i) : vec = vec
```

generator for element

generator for equality

proof-relevant!

16

Higher Inductive Type

Type freely generated by constructors for elements, equalities, equalities between equalities, …

```
RepoDesc : Type

vec : RepoDesc

(a↔b at i) : vec = vec

commute:
 (a↔b at i)o(c↔d at j)
=(c↔d at j)o(a↔b at i)
```

proof-relevant!

generator for element

generator for equality

generator for equality between equalities

16

Type: Patch

Elements:

```
id      : Patch
_∘_     : Patch → Patch → Patch
!       : Patch → Patch
_↔_at_  : Char → Char → Fin n → Patch
```

Equality:

```
(a↔b at i)o(c↔d at j)=
   (c↔d at j)o(a↔b at i)

...
id o p = p = p o id

po(qor) = (poq)or

!p o p = id = p o !p

p=p

p=q if q=p

p=r if p=q and q=r

!p = !p’ if p = p’

p o q = p’ o q’ if p = p’ and q = q’
```

Type: Patch

Elements:

```
id     : Patch
_∘_    : Patch → Patch → Patch
!      : Patch → Patch
_↔_at_ : Char → Char → Fin n → Patch
```

Equality:

```
(a↔b at i)o(c↔d at j)=
   (c↔d at j)o(a↔b at i)

...
id o p = p = p o id
po(qor) = (poq)or
!p o p = id = p o !p
p=p
p=q if q=p
p=r if p=q and q=r
!p = !p’ if p = p’
p o q = p’ o q’ if p = p’ and q = q’
```

Type: RepoDesc

Type: Patch

Elements:

```
id     : Patch
_∘_    : Patch → Patch → Patch
!      : Patch → Patch
_↔_at_ : Char → Char → Fin n → Patch
```

Equality:

```
(a↔b at i)o(c↔d at j)=
   (c↔d at j)o(a↔b at i)

...
id o p = p = p o id
po(qor) = (poq)or
!p o p = id = p o !p
p=p
p=q if q=p
p=r if p=q and q=r
!p = !p’ if p = p’
p o q = p’ o q’ if p = p’ and q = q’
```

Type: RepoDesc

Element: vec : RepoDesc

Type: Patch

Elements:

```
id       : Patch
_∘_      : Patch → Patch → Patch
!        : Patch → Patch
_↔_at_ : Char → Char → Fin n → Patch
```

Equality:

```
(a↔b at i)o(c↔d at j)=
   (c↔d at j)o(a↔b at i)

...
id o p = p = p o id
po(qor) = (poq)or
!p o p = id = p o !p
p=p
p=q if q=p
p=r if p=q and q=r
!p = !p’ if p = p’
p o q = p’ o q’ if p = p’ and q = q’
```

Type: RepoDesc

Element: vec : RepoDesc

Equality:

```
a↔b at i : vec = vec
```

Type: Patch

Elements:

```
id      : Patch
_∘_     : Patch → Patch → Patch
!       : Patch → Patch
_↔_at_  : Char → Char → Fin n → Patch
```

Equality:

```
(a↔b at i)o(c↔d at j)=
   (c↔d at j)o(a↔b at i)

...
id o p = p = p o id
po(qor) = (poq)or
!p o p = id = p o !p
p=p
p=q if q=p
p=r if p=q and q=r
!p = !p’ if p = p’
p o q = p’ o q’ if p = p’ and q = q’
```

Type: RepoDesc

Element: vec : RepoDesc

Equality:

$$\mathtt{a{\leftrightarrow}b\ at\ i : } \overbrace{\mathtt{vec = vec}}^{\mathtt{Patch}}$$

Type: Patch

Elements:

```
id      : Patch
_∘_     : Patch → Patch → Patch
!       : Patch → Patch
_↔_at_  : Char → Char → Fin n → Patch
```

Equality:

```
(a↔b at i)o(c↔d at j)=
   (c↔d at j)o(a↔b at i)

...
id o p = p = p o id
po(qor) = (poq)or
!p o p = id = p o !p
p=p
p=q if q=p
p=r if p=q and q=r
!p = !p’ if p = p’
p o q = p’ o q’ if p = p’ and q = q’
```

Type: RepoDesc

Element: vec : RepoDesc

Equality:

$$a \leftrightarrow b \text{ at } i : \overbrace{vec = vec}^{\text{Patch}}$$

Equality between equalities:

```
commute :
(a↔b at i)o(c↔d at j)=
(c↔d at j)o(a↔b at i)
```

... basic axioms only!

17

Type: Patch

Elements:

```
id      : Patch
_∘_     : Patch → Patch → Patch
!       : Patch → Patch
_↔_at_  : Char → Char → Fin n → Patch
```

Equality:

```
(a↔b at i)o(c↔d at j)=
   (c↔d at j)o(a↔b at i)
...
id o p = p = p o id
po(qor) = (poq)or
!p o p = id = p o !p
p=p
p=q if q=p
p=r if p=q and q=r
!p = !p' if p = p'
p o q = p' o q' if p = p' and q = q'
```

Type: RepoDesc

Element: vec : RepoDesc

Equality:

$$a{\leftrightarrow}b \text{ at } i : \underbrace{\text{vec} = \text{vec}}_{\text{Patch}}$$

Equality between equalities:

```
commute :
(a↔b at i)o(c↔d at j)=
(c↔d at j)o(a↔b at i)
```

… basic axioms only!

Everything else comes “for free” from the equality type!

Typed Patches

```
RepoDesc : Type
```

```
vec : RepoDesc
compressed : RepoDesc
```

generators for elements

```
a↔b at i : vec = vec
gzip : vec = compressed
```

generators for equalities

Typed Patches

```
RepoDesc : Type
```

```
vec : RepoDesc
compressed : RepoDesc
```

generators for elements

```
a↔b at i : vec = vec
gzip : vec = compressed
```

generators for equalities

$\underbrace{\texttt{vec = compressed}}_{\texttt{Patch vec compressed}}$

18

Patches as a HIT

1. How do you define `Patch` using a higher inductive type?

2. **What is the elimination rule for `RepoDesc`?**

3. How do you use the elim. rule to define `interp`?

19

RepoDesc recursion

To define a function `RepoDesc → A`
it suffices to

RepoDesc recursion

To define a function `RepoDesc → A` it suffices to

- map the element generators of `RepoDesc` to elements of `A`

RepoDesc recursion

To define a function `RepoDesc → A`
it suffices to

- map the element generators of `RepoDesc`
  to elements of `A`
- map the equality generators of `RepoDesc`
  to equalities between the corresponding elements of `A`

RepoDesc recursion

To define a function `RepoDesc → A`
it suffices to

- map the element generators of `RepoDesc` to elements of `A`
- map the equality generators of `RepoDesc` to equalities between the corresponding elements of `A`
- map the equality-between-equality generators to equalities between the corresponding equalities in `A`

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A
```

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A
```

f_1`(a↔b at i) := … : f(vec) = f(vec)`

21

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A

f₁(a↔b at i) := … : f(vec) = f(vec)

f₂(compose a b c d i j i≠j) := …
  : f₁((a↔b at i)o(c↔d at j))

  = f₁((c↔d at j)o(a↔b at j))
```

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A

f1(a↔b at i) := … : f(vec) = f(vec)

f2(compose a b c d i j i≠j) := …
   : f1((a↔b at i)o(c↔d at j))
   = f1((c↔d at j)o(a↔b at j))
```

You only specify `f` *on generators,*
not `id`*,*o*,* `!`*,group laws,congruence,…*
(1 patch and 4 basic axioms, instead of 4 and 14!)

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A

f₁(a↔b at i) := … : f(vec) = f(vec)

f₂(compose a b c d i j i≠j) := …
  : f₁((a↔b at i)o(c↔d at j))
  = f₁((c↔d at j)o(a↔b at j))
```

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A
f₁(a↔b at i) := … : f(vec) = f(vec)
f₂(compose a b c d i j i≠j) := …
  : f₁((a↔b at i)o(c↔d at j))
  = f₁((c↔d at j)o(a↔b at j))
```

Type-generic equality rules say that functions act homomorphically on `id`, `o`, `!`, …

RepoDesc recursion

To define a function $f : \mathrm{RepoDesc} \to A$
it suffices to give

$$= f_1(a \leftrightarrow b \text{ at } i) \circ f_1(c \leftrightarrow d \text{ at } j)$$

$$f(\mathrm{vec}) := \ldots : A$$

$$f_1(a \leftrightarrow b \text{ at } i) := \ldots : f(\mathrm{vec}) = f(\mathrm{vec})$$

$$f_2(\mathrm{compose}\ a\ b\ c\ d\ i\ j\ i{\neq}j) := \ldots$$

$$: f_1((a \leftrightarrow b \text{ at } i) \circ (c \leftrightarrow d \text{ at } j))$$

$$= f_1((c \leftrightarrow d \text{ at } j) \circ (a \leftrightarrow b \text{ at } j))$$

Type-generic equality rules say that functions act homomorphically on `id`, `o`, `!`, …

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A

f1(a↔b at i) := … : f(vec) = f(vec)

f2(compose a b c d i j i≠j) := …
  : f1((a↔b at i)o(c↔d at j))
  = f1((c↔d at j)o(a↔b at j))
```

RepoDesc recursion

To define a function `f : RepoDesc → A`
it suffices to give

```
f(vec) := … : A
f₁(a↔b at i) := … : f(vec) = f(vec)
f₂(compose a b c d i j i≠j) := …
  : f₁((a↔b at i)o(c↔d at j))
  = f₁((c↔d at j)o(a↔b at j))
```

All functions on RepoDesc respect patches

All functions on patches respect patch equality

Patches as a HIT

1. How do you define `Patch` using a higher inductive type?

2. What is the elimination rule for `RepoDesc`?

3. **How do you use the elim. rule to define `interp`?**

Interp

Goal is to define:

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(id) = (λx.x, …)
interp(q o p) = (interp q) o_b (interp p)
interp(!p) = !_b (interp p)
interp(a↔b at i) = swapat a b i
```

Interp

Goal is to define:

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(id) = (λx.x, …)
interp(q o p) = (interp q) o_b (interp p)
interp(!p) = !_b (interp p)
interp(a↔b at i) = swapat a b i
```

But only tool available is RepoDesc recursion:
no direct recursion over proofs of equality

25

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Need to pick A *and define*

```
f(vec) := … : A
f₁(a↔b at i) := … : f(vec) = f(vec)
f₂(compose) := …
```

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Key idea: pick A = Type *and define*

```
  f(vec) := … : Type
  f₁(a↔b at i) := … : f(vec) = f(vec)
  f₂(compose) := …
```

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Key idea: pick A = Type *and define*

```
f(vec) := Vec Char n : Type
```

f_1(a↔b at i) := … : f(vec) = f(vec)

f_2(compose) := …

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Key idea: pick A = Type *and define*

```
f(vec) := Vec Char n : Type
```

f_1`(a↔b at i) := … : Vec Char n = Vec Char n`

f_2`(compose) := …`

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Key idea: pick A = Type *and define*

```
f(vec) := Vec Char n : Type
f₁(a↔b at i) := ua(swapat a b i)
              : Vec Char n = Vec Char n
f₂(compose) := …
```

```
interp : vec = vec
          → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Key idea: pick A = Type *and define*

```
  f(vec) := Vec Char n : Type
  f₁(a↔b at i) := ua(swapat a b i)
                  : Vec Char n = Vec Char n
  f₂(compose) := …
```

Voevodky's univalence axiom ⊃ bijective types are equal

```
interp : vec = vec
      → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Key idea: pick A = Type *and define*

```
f(vec) := Vec Char n : Type
f₁(a↔b at i) := ua(swapat a b i)
              : Vec Char n = Vec Char n
f₂(compose) := <proof about swapat as before>
```

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
interp(a↔b at i) = swapat a b i
```

Key idea: pick A = Type *and define*

```
I(vec) := Vec Char n : Type
```

$\mathtt{I}_1$`(a↔b at i) := ua(swapat a b i)`

```
                  : Vec Char n = Vec Char n
```

$\mathtt{I}_2$`(compose) :=` <proof about swapat as before>

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
```

$\mathtt{interp(p)} = \mathtt{ua}^{-1}(\mathtt{I_1(p)})$

Key idea: pick A = Type *and define*

```
I(vec) := Vec Char n : Type
I₁(a↔b at i) := ua(swapat a b i)
                   : Vec Char n = Vec Char n
I₂(compose) := &lt;proof about swapat as before&gt;
```

```
interp : vec = vec
       → Bijection (Vec Char n) (Vec Char n)
```

$\mathtt{interp}(p) = ua^{-1}(I_1(p))$

Satisfies the desired equations (as propositional equalities):

$\mathtt{interp}(\mathtt{id}) = (\lambda x.x, \ldots)$

$\mathtt{interp}(q \circ p) = (\mathtt{interp}\ q) \circ_b (\mathtt{interp}\ p)$

$\mathtt{interp}(!p) = !_b\ (\mathtt{interp}\ p)$

$\mathtt{interp}(a{\leftrightarrow}b \text{ at } i) = \mathtt{swapat}\ a\ b\ i$

Summary

Summary

- `I : RepoDesc → Type` interprets `RepoDesc`'s as `Type`s, patches as bijections, satisfying patch equalities

Summary

- `I : RepoDesc → Type` interprets `RepoDesc`'s as `Types`, patches as bijections, satisfying patch equalities
- Higher inductive elim. defines functions that respect equality: you specify what happens on the generators; homomorphically extended to `id`,o,`!`,...

Summary

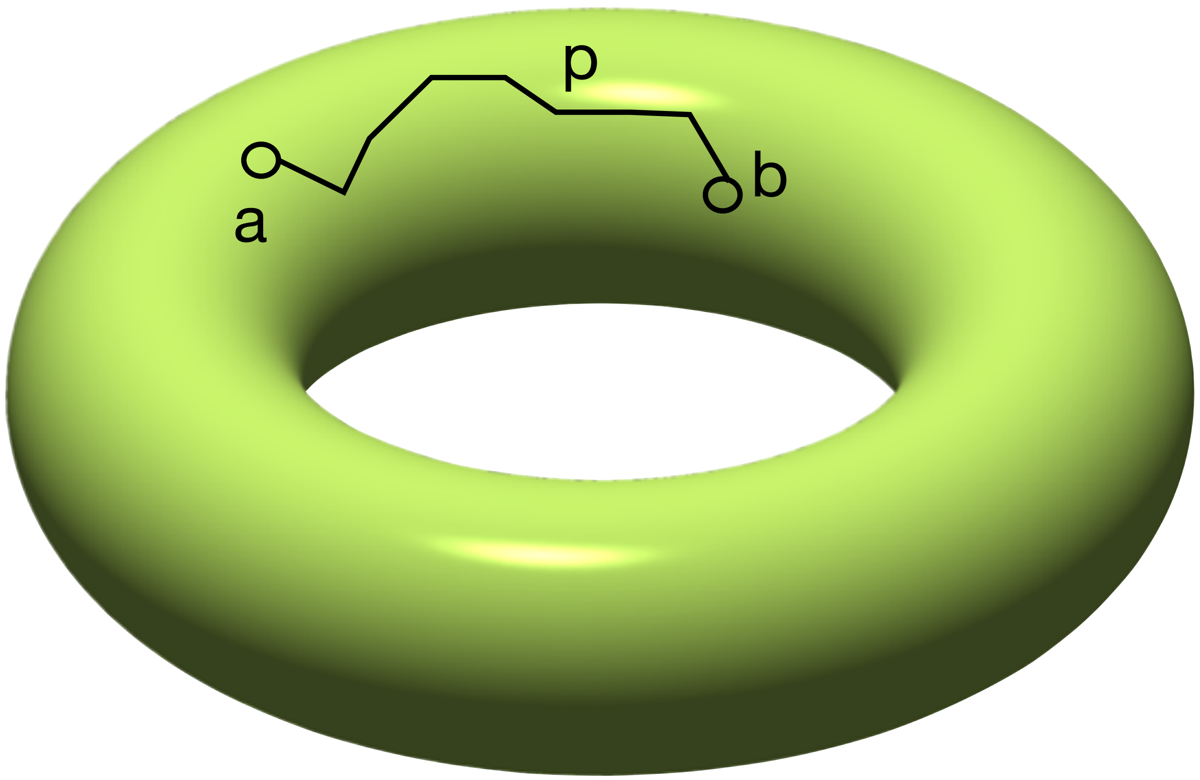
- `I : RepoDesc → Type` interprets `RepoDesc`'s as `Type`s, patches as bijections, satisfying patch equalities
- Higher inductive elim. defines functions that respect equality: you specify what happens on the generators; homomorphically extended to `id`,o,`!`,...
- Univalence lets you give a computational model of equality proofs (here, patches); guaranteed to satisfy laws

Summary

- `I : RepoDesc → Type` interprets `RepoDesc`'s as `Type`s, patches as bijections, satisfying patch equalities
- Higher inductive elim. defines functions that respect equality: you specify what happens on the generators; homomorphically extended to `id`,o,`!`,...
- Univalence lets you give a computational model of equality proofs (here, patches); guaranteed to satisfy laws
- Shorter definition and code than using quotients: 1 basic patch & 4 basic axioms of equality, instead of 4 patches & 14 equations

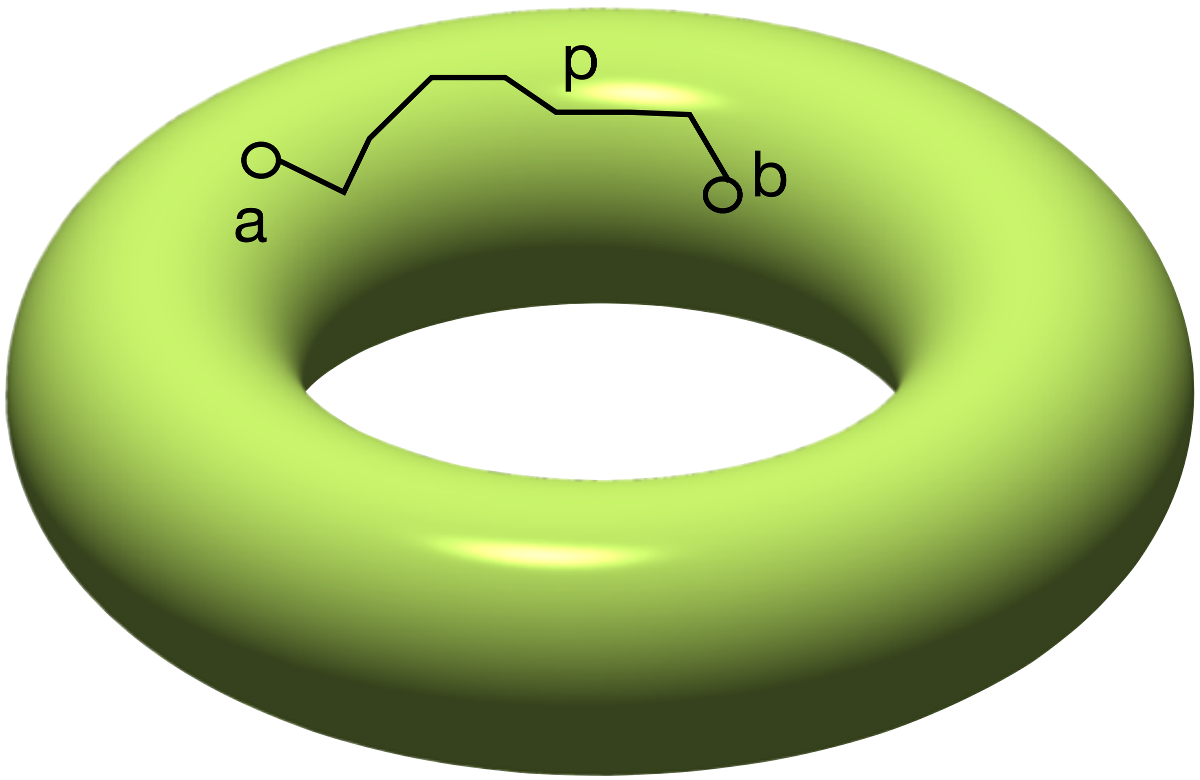
Where does this programming technique come from?

Homotopy type theory



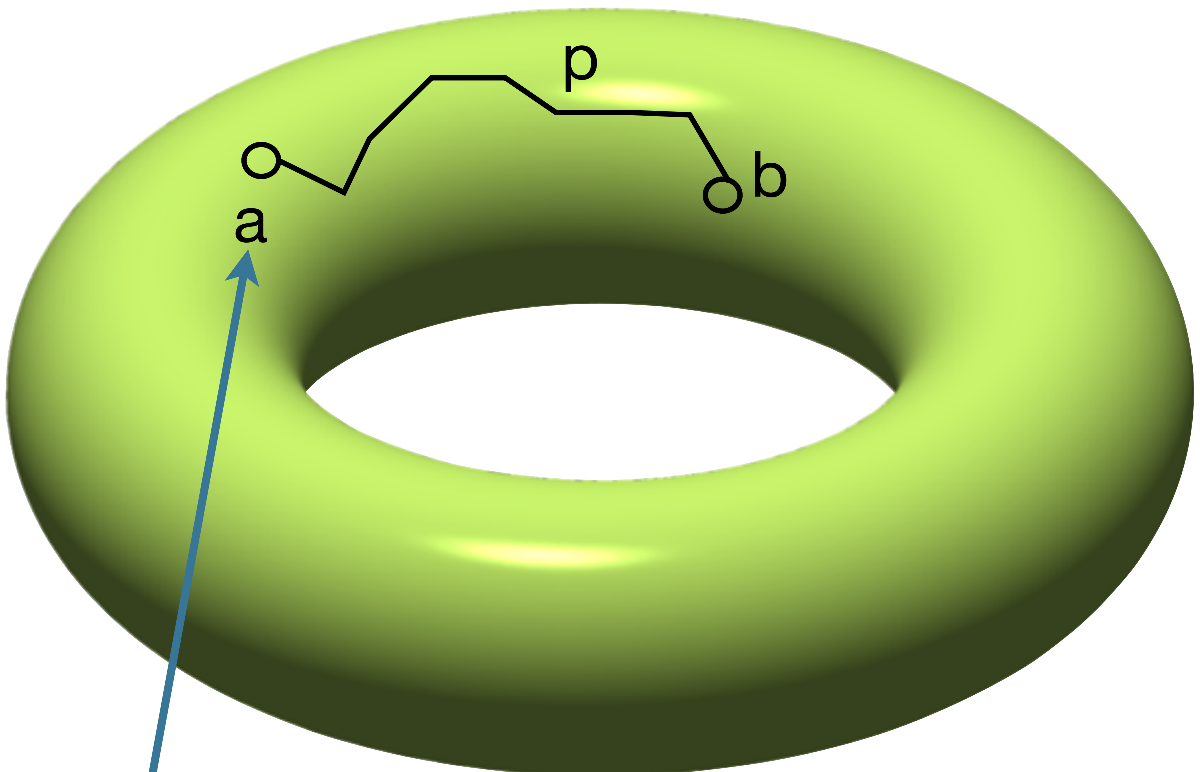
Homotopy type theory

a space is a type A



Homotopy type theory

a space is a type A

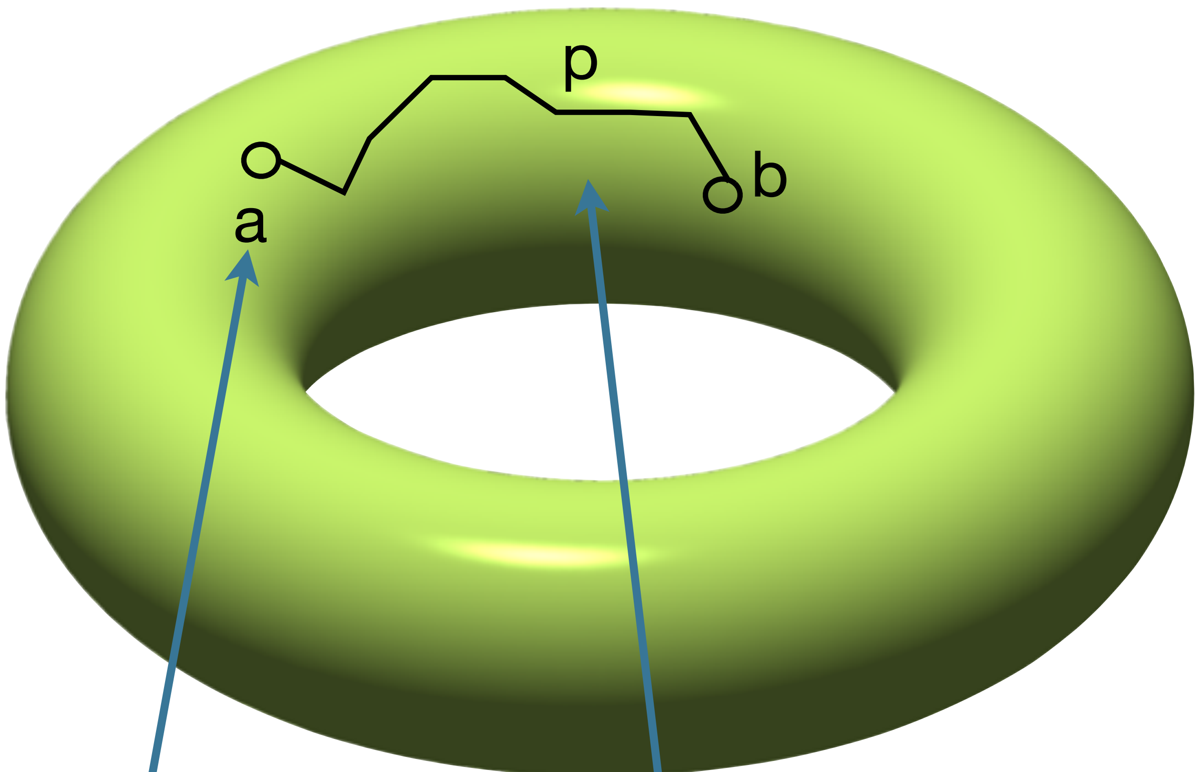


points are elements

$a : A$

Homotopy type theory

a space is a type A



points are elements

$a : A$

paths are *proofs of equality*

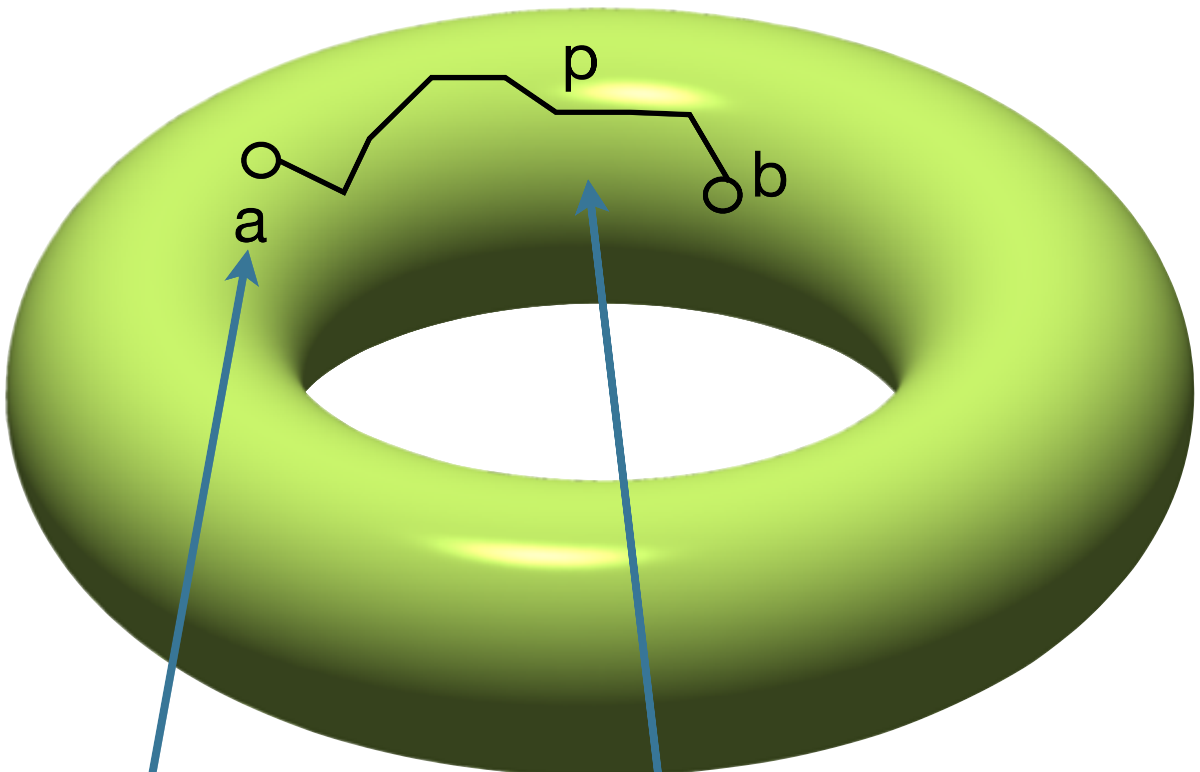
$p \; : \; a =_A b$

37

Homotopy type theory

a space is a type A

path operations



points are elements

$a : A$

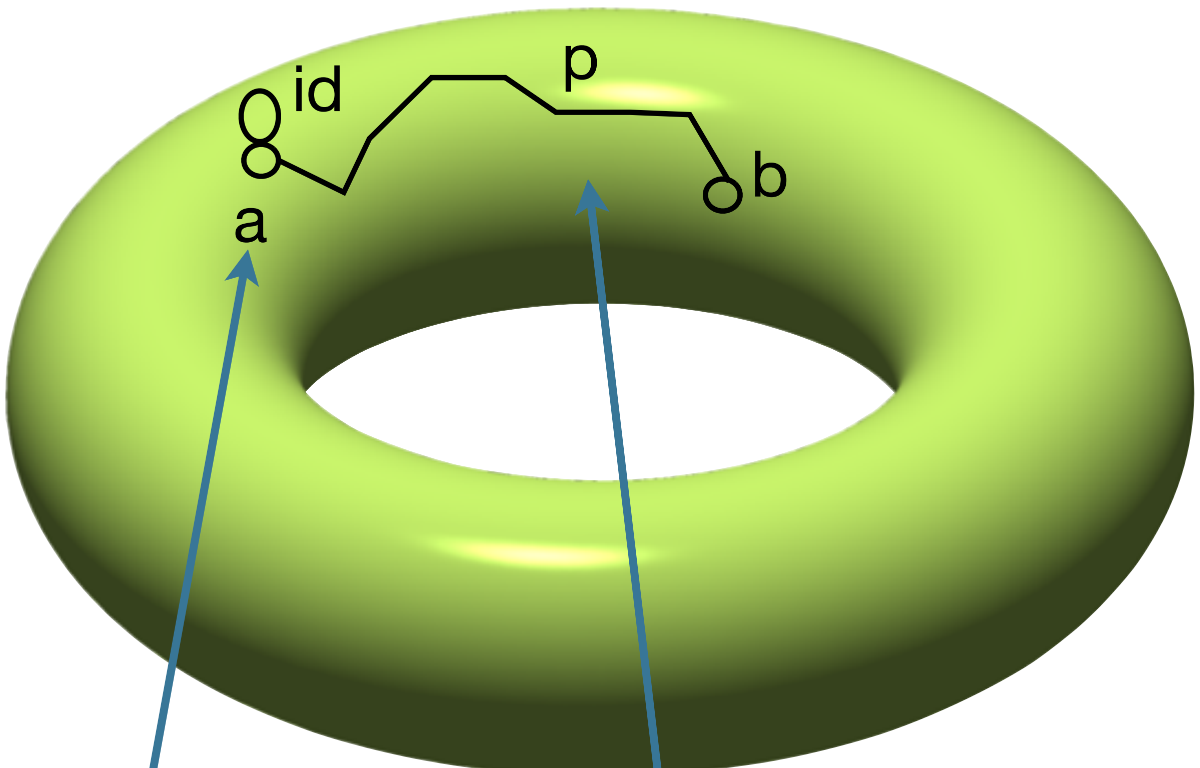
paths are

proofs of equality

$p \; : \; a =_A b$

Homotopy type theory

a space is a type A



points are elements

$a : A$

paths are *proofs of equality*

$p \ : \ a =_A b$

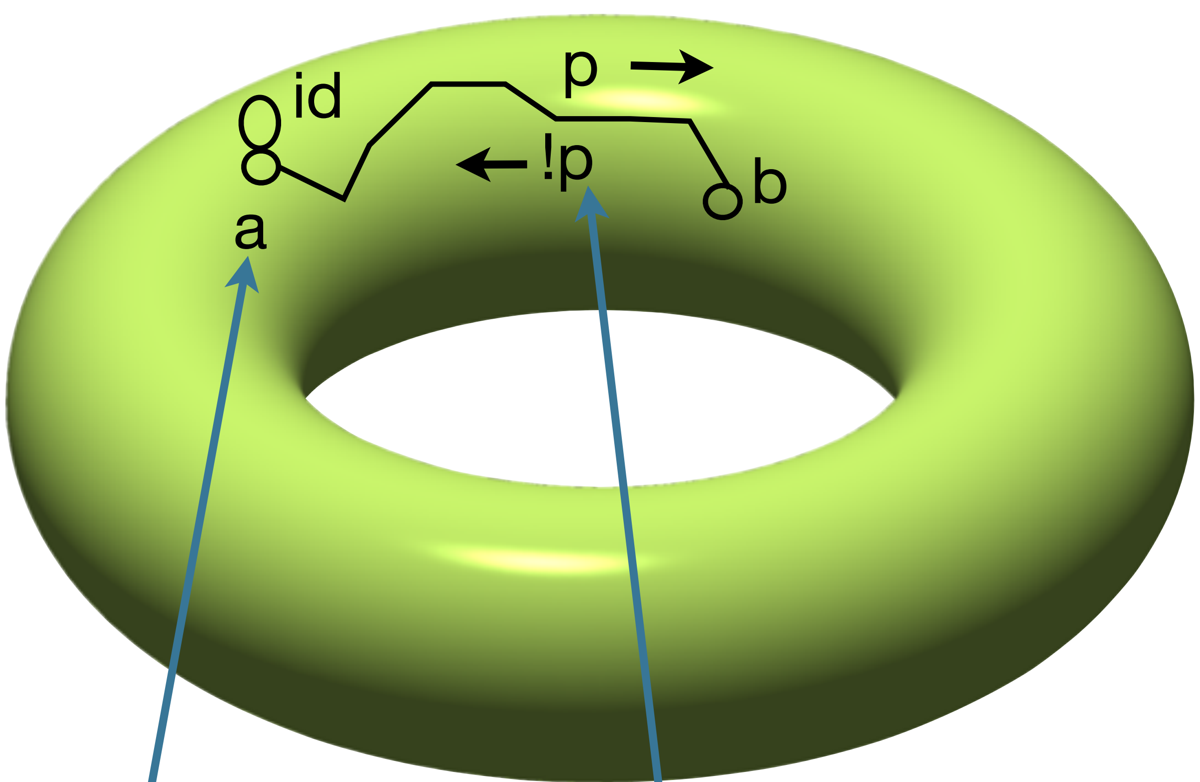
path operations

`id      : a = a (refl)`

37

Homotopy type theory

a space is a type A



points are elements

$a : A$

paths are *proofs of equality*

$p : a =_A b$

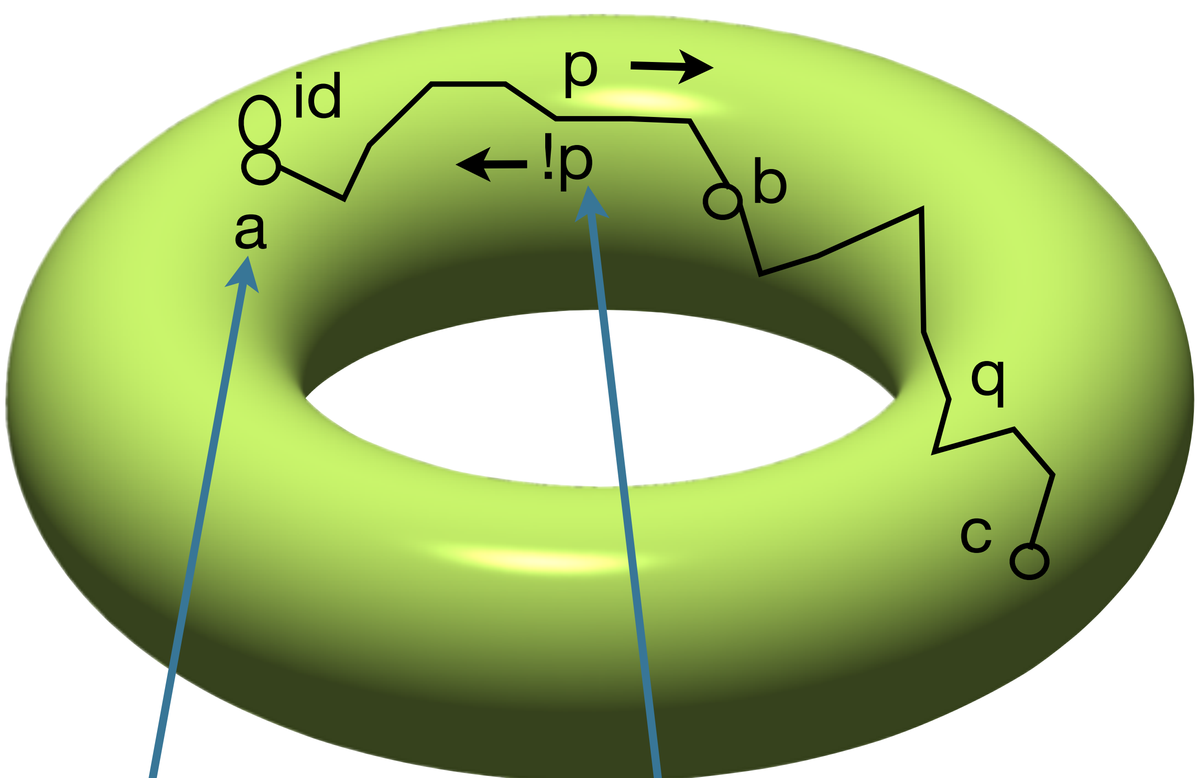
path operations

```
id    : a = a (refl)
 !p   : b = a (sym)
```


37

Homotopy type theory

a space is a type A



points are elements
$a : A$

paths are ***proofs of equality***
$p : a =_A b$

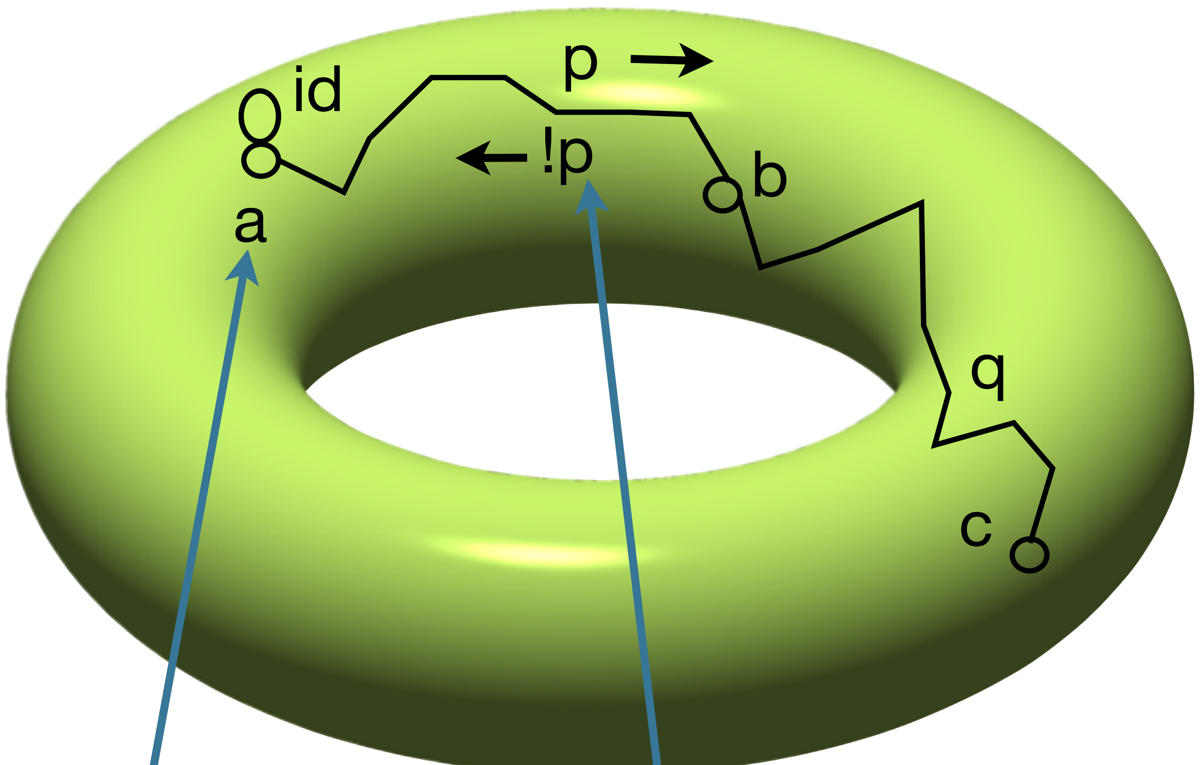
path operations

```
id    : a = a (refl)
 !p   : b = a (sym)
q o p : a = c (trans)
```


37

Homotopy type theory

a space is a type A



points are elements

$a : A$

paths are *proofs of equality*

$p : a =_A b$

path operations

id : a = a (refl)

!p : b = a (sym)

q o p : a = c (trans)

homotopies

id o p = p

!p o p = id

r o (q o p)
 = (r o q) o p

Homotopy type theory

a space is a type A

points are elements
$a : A$

paths are *proofs of equality*
$p : a =_A b$

path operations

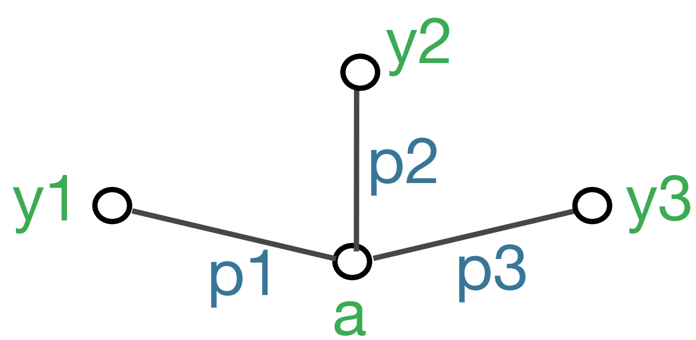
```
id     : a = a (refl)
 !p    : b = a (sym)
 q o p : a = c (trans)
```

homotopies

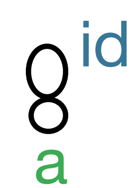
```
id o p = p
!p o p = id
r o (q o p)
   = (r o q) o p
```

Equality elimination rule

Type of equalities between *a* **and -**

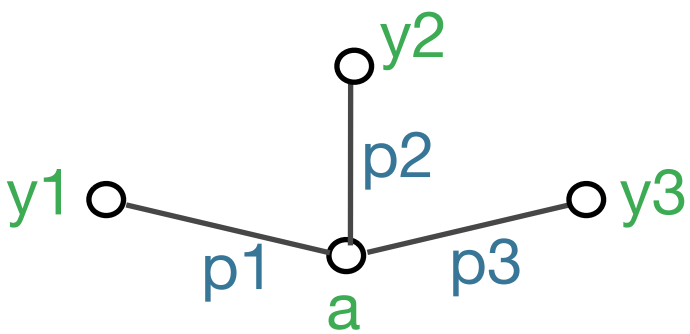


is inductively generated by

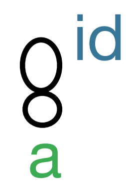


Equality elimination rule

Type of equalities between a and -



is inductively generated by



Fix a type A with element $a:A$.

For a family of types $C(y:A,\ p:a=y)$,

to give an element of

$C(y,p)$ for all y and $p:a=y$,

suffices to give an element of

$C(a,id)$

Composition and Assoc

```
_o_ : a = b → b = c → a = c
id o p = p

o-assoc : (p : a=b)(q : b=c)(r : c=d)
          → p o (q o r) = (p o q) o r
o-assoc id id id = id
```


39

Functions are functors

$f : A \to B$ has action at all levels

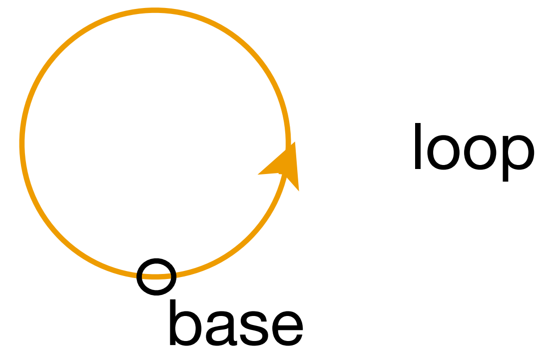
$f_1 : (a_1\ a_2 : A)$
$\to a_1 =_A a_2 \to f(a_1) =_B f(a_2)$

$f_2 : (a_1\ a_2 : A)(p\ p' : a_1 =_A a_2) \to$
$p =_{a1=a2} p' \to$
$f_1(p) =_{f(a1)=f(a2)} f_1(p')$

and so on

The Circle

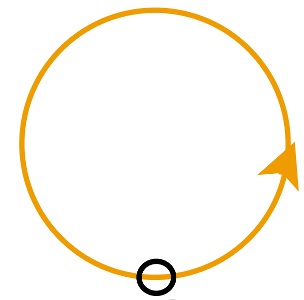
Circle S^1 is HIT generated by



The Circle

Circle S^1 is HIT generated by

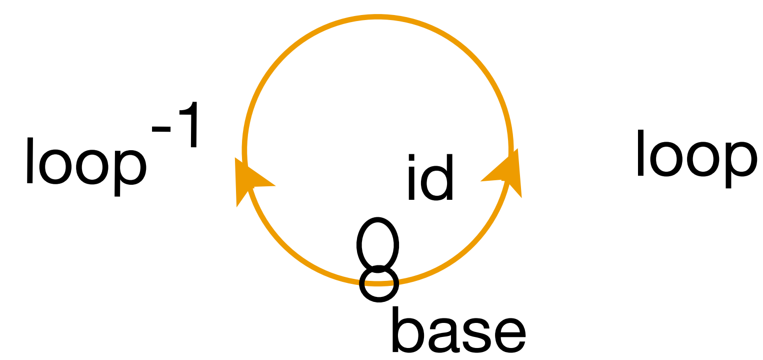
```
base : S¹
loop : base = base
```



The Circle

Circle S^1 is HIT generated by

```
base : S¹
loop : base = base
```



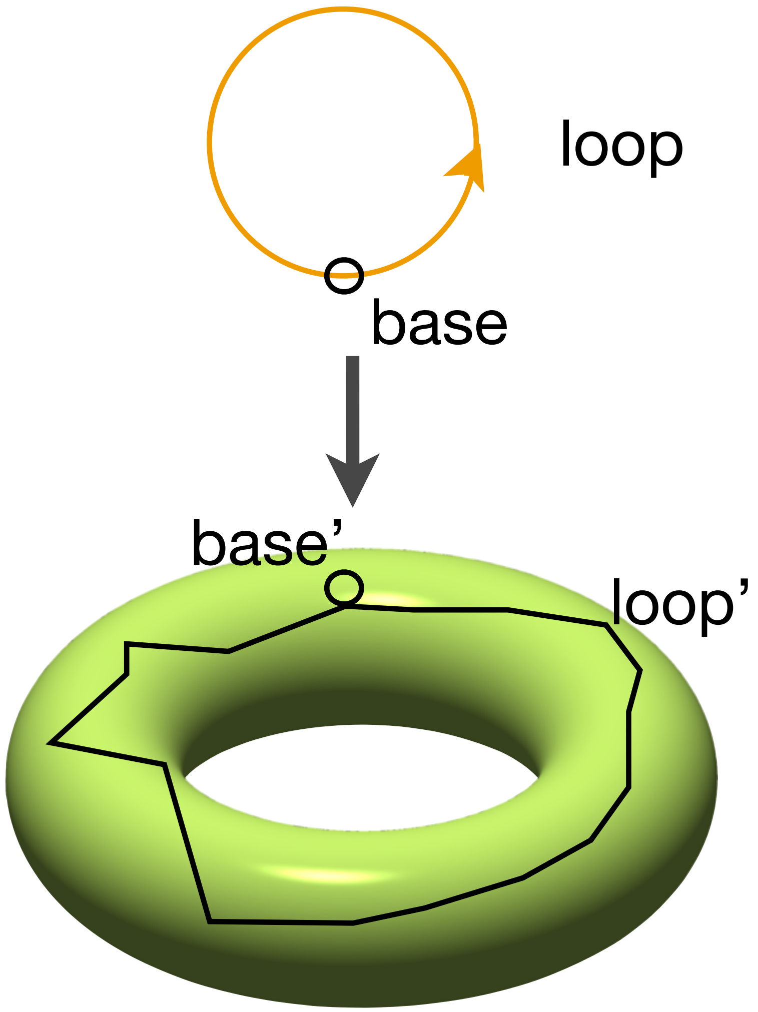
Free type: equipped with

```
id                inv : loop o loop⁻¹ = id
loop⁻¹            ...
loop o loop
```

The Circle

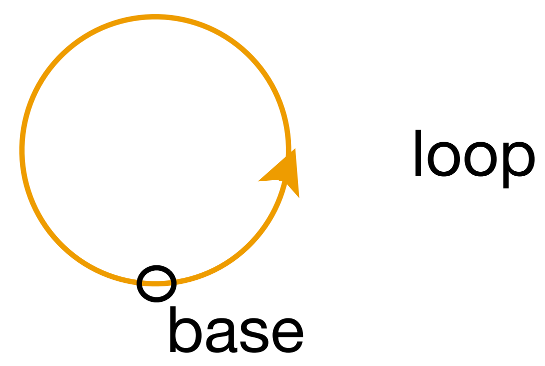
Circle recursion:
function $S^1 \to X$ determined by

```
base' : X
loop' : base' = base'
```



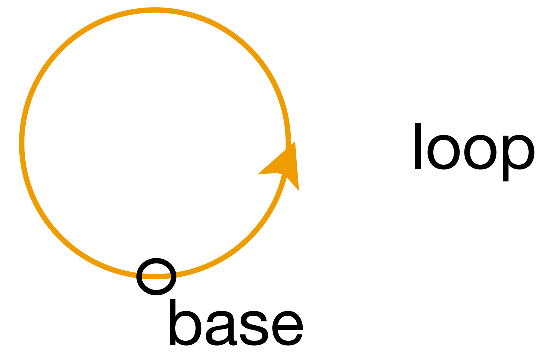
Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?



Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?

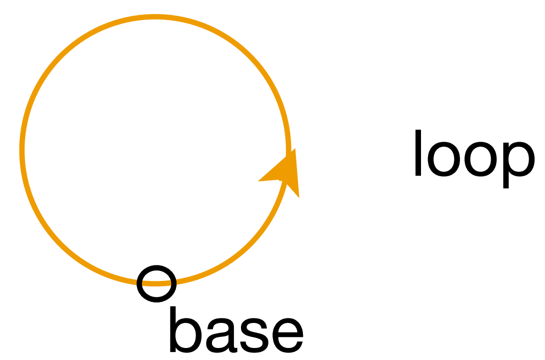



```
id
```


43

Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?

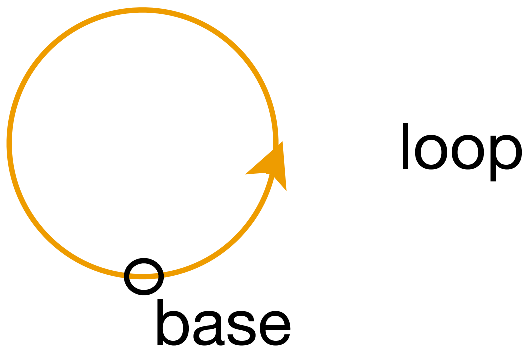



```
id
loop
```


43

Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?



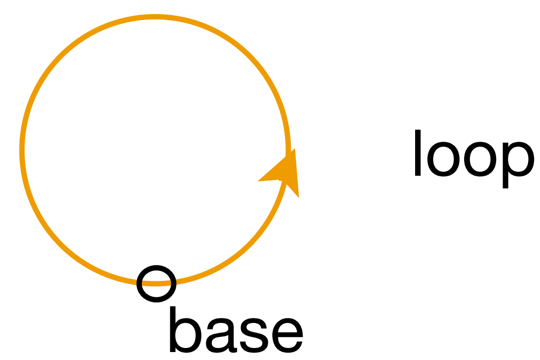
`id`

`loop`

`loop`$^{-1}$

Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?




```
id
```

```
loop
```

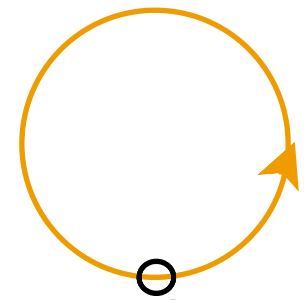
$\mathtt{loop}^{-1}$

```
loop o loop
```


43

Fundamental group of circle

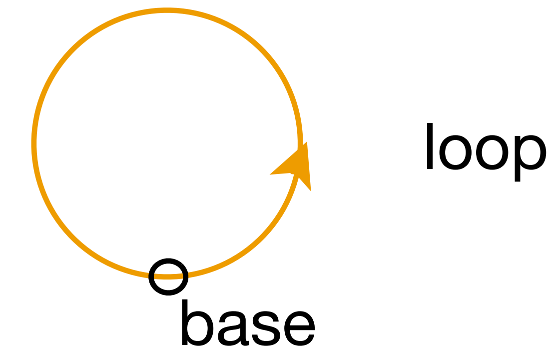
How many different loops are there on the circle, up to *homotopy*?




```
id
loop
loop⁻¹
loop o loop
loop⁻¹ o loop⁻¹
```

Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?



$\texttt{id}$

$\texttt{loop}$

$\texttt{loop}^{-1}$

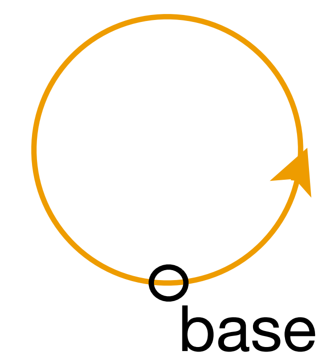
$\texttt{loop o loop}$

$\texttt{loop}^{-1}\ \texttt{o loop}^{-1}$

$\texttt{loop o loop}^{-1}$

Fundamental group of circle

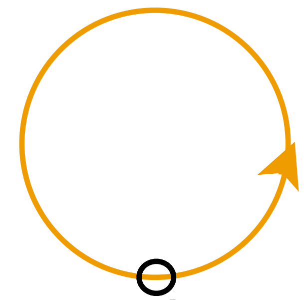
How many different loops are there on the circle, up to *homotopy*?




```
id
loop
loop⁻¹
loop o loop
loop⁻¹ o loop⁻¹
loop o loop⁻¹  = id
```

Fundamental group of circle

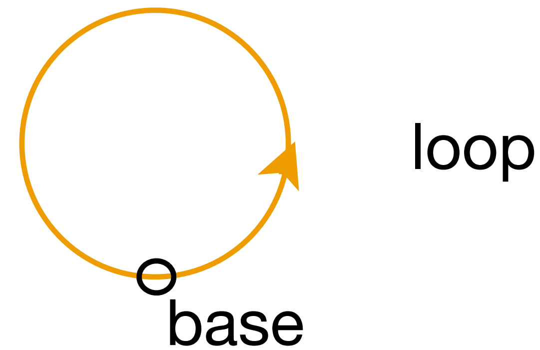
How many different loops are there on the circle, up to *homotopy*?




```
id                         0
loop
loop⁻¹
loop o loop
loop⁻¹ o loop⁻¹
loop o loop⁻¹  = id
```

Fundamental group of circle

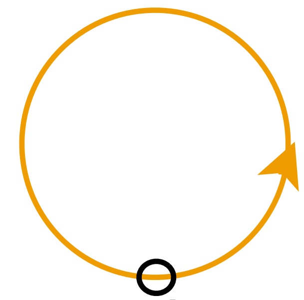
How many different loops are there on the circle, up to *homotopy*?



`id`	`0`
`loop`	`1`
$\texttt{loop}^{-1}$	
`loop o loop`	
$\texttt{loop}^{-1}$ `o` $\texttt{loop}^{-1}$	
`loop o` $\texttt{loop}^{-1}$ `= id`	

Fundamental group of circle

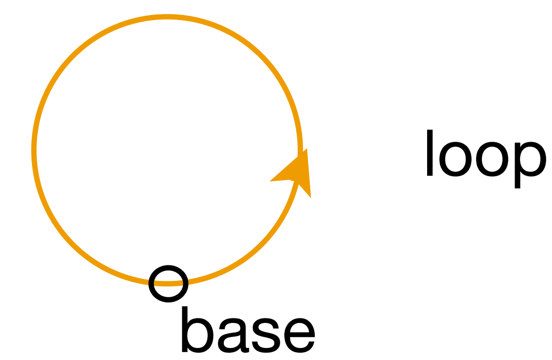
How many different loops are there on the circle, up to *homotopy*?



`id`	`0`
`loop`	`1`
$\texttt{loop}^{-1}$	`-1`
`loop o loop`	
$\texttt{loop}^{-1}\ \texttt{o}\ \texttt{loop}^{-1}$	
$\texttt{loop o loop}^{-1}\ = \texttt{id}$	

Fundamental group of circle

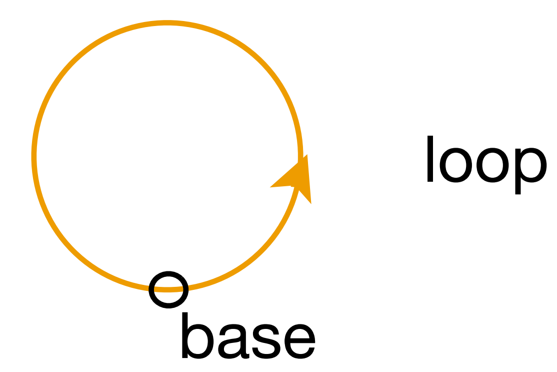
How many different loops are there on the circle, up to *homotopy*?



`id`	`0`
`loop`	`1`
`loop`$^{-1}$	`-1`
`loop o loop`	`2`
`loop`$^{-1}$ `o loop`$^{-1}$	
`loop o loop`$^{-1}$ `= id`	

Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?

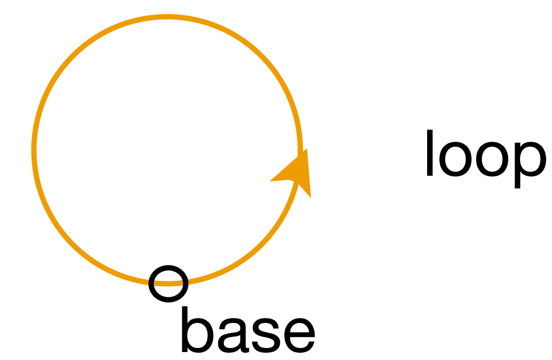


`id`	`0`
`loop`	`1`
$\texttt{loop}^{-1}$	`-1`
`loop o loop`	`2`
$\texttt{loop}^{-1}\ \texttt{o}\ \texttt{loop}^{-1}$	`-2`

$\texttt{loop o loop}^{-1} = \texttt{id}$

Fundamental group of circle

How many different loops are there on the circle, up to *homotopy*?

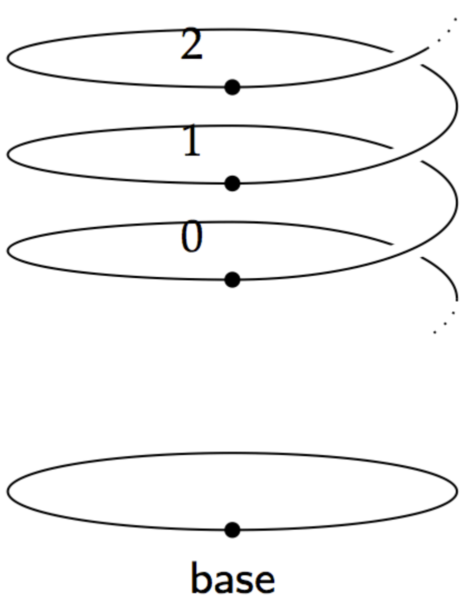


`id`	`0`
`loop`	`1`
$\texttt{loop}^{-1}$	`-1`
`loop o loop`	`2`
$\texttt{loop}^{-1}\ \texttt{o}\ \texttt{loop}^{-1}$	`-2`
$\texttt{loop o loop}^{-1} = \texttt{id}$	`0`

Fundamental group of circle

Theorem. Group of loops on the circle is isomorphic to $\mathbb{Z}$

Proof: Define universal cover

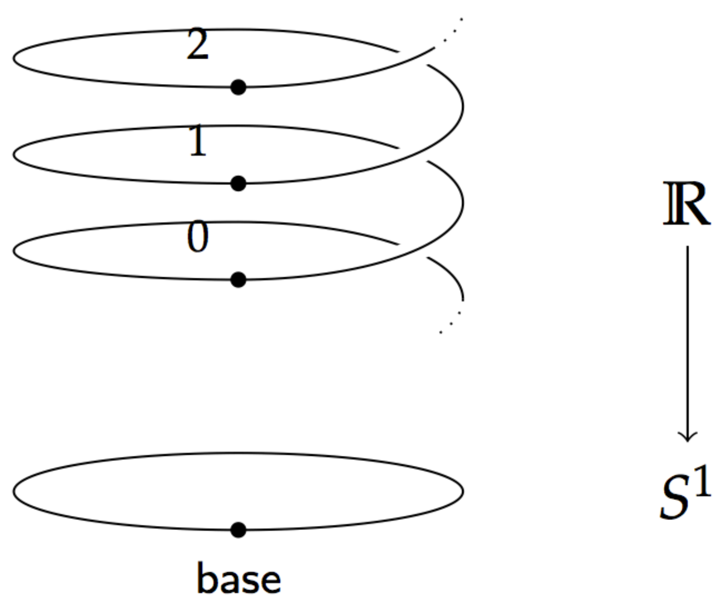


$$\begin{array}{c} \mathbb{R} \\ \downarrow \\ S^1 \end{array}$$

Fundamental group of circle

Theorem. Group of loops on the circle
is isomorphic to $\mathbb{Z}$

Proof: Define universal cover



$$\mathsf{Cover} : S^1 \to \mathsf{Type}$$

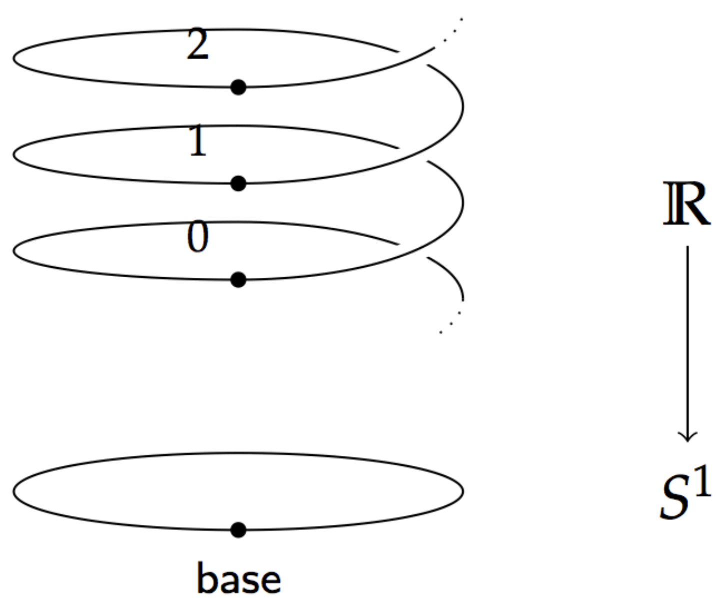
$$\mathsf{Cover}(\mathsf{base}) := \mathbb{Z}$$

$$\mathsf{Cover}_1(\mathsf{loop}) :=$$
$$\quad \mathsf{ua}(\mathsf{successor}) : \mathbb{Z} = \mathbb{Z}$$

Fundamental group of circle

Theorem. Group of loops on the circle is isomorphic to $\mathbb{Z}$

Proof: Define universal cover



$$\mathtt{Cover} : S^1 \to \mathtt{Type}$$

$$\mathtt{Cover(base)} := \mathbb{Z}$$

$$\mathtt{Cover}_1\mathtt{(loop)} :=$$
$$\quad \mathtt{ua(successor)} : \mathbb{Z} = \mathbb{Z}$$

interpret loop as "add 1" bijection

Homotopy in HoTT

$\pi_1(S^1) = \mathbb{Z}$

$\pi_{k<n}(S^n) = 0$

Hopf fibration

$\pi_2(S^2) = \mathbb{Z}$

$\pi_3(S^2) = \mathbb{Z}$

James Construction

$\pi_4(S^3) = \mathbb{Z}_?$

Freudenthal

$\pi_n(S^n) = \mathbb{Z}$

K(G,n)

Cohomology axioms

Blakers-Massey

Van Kampen

Covering spaces

Whitehead for n-types

[Brunerie, Finster, Hou, Licata, Lumsdaine, Shulman]

What's next?

- Operational semantics of HITs and univalence is still an open problem in general, though some special cases are known
- Have just started exploring programming applications
- Extensions to this example: more realistic basic patches, patches that can fail (partial bijections), implement `merge`