Mathematical and Computational Applications of Homotopy Type Theory

Dan Licata

Wesleyan University
Department of Mathematics and Computer Science


LUXURY
Sunkist Oranges
REG. U.S. PAT. OFF.
GROWN IN U.S.A.
CALIFORNIA FRUITGROWERS EXCHANGE
BOX
The
I Promised You
from California.

Kepler Conjecture (1611)

No way to pack equally-sized spheres in space has higher density than

Hales' proof (1998)

- Reduces Kepler Conjecture to proving that a function has a lower bound on 5,000 different configurations of spheres

- This requires solving 100,000 linear programming problems

- 1998 submission:
  300 pages of math
  + 50,000 LOC (revised 2006: 15,000 LOC)

Proofs can be hard to check

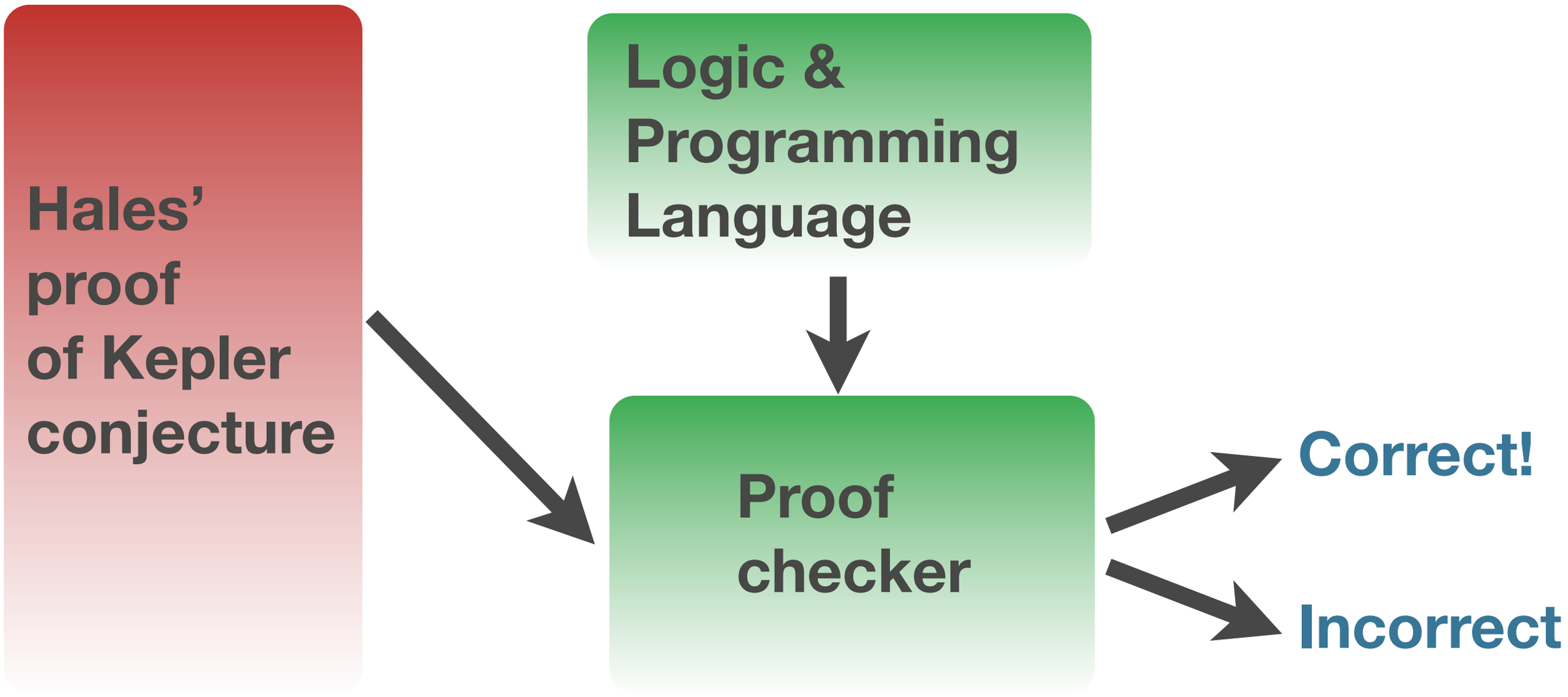
In 2003, after 4 years' work,
12 referees had checked lots of lemmas,
but gave up on verifying the proof

Proofs can be hard to check

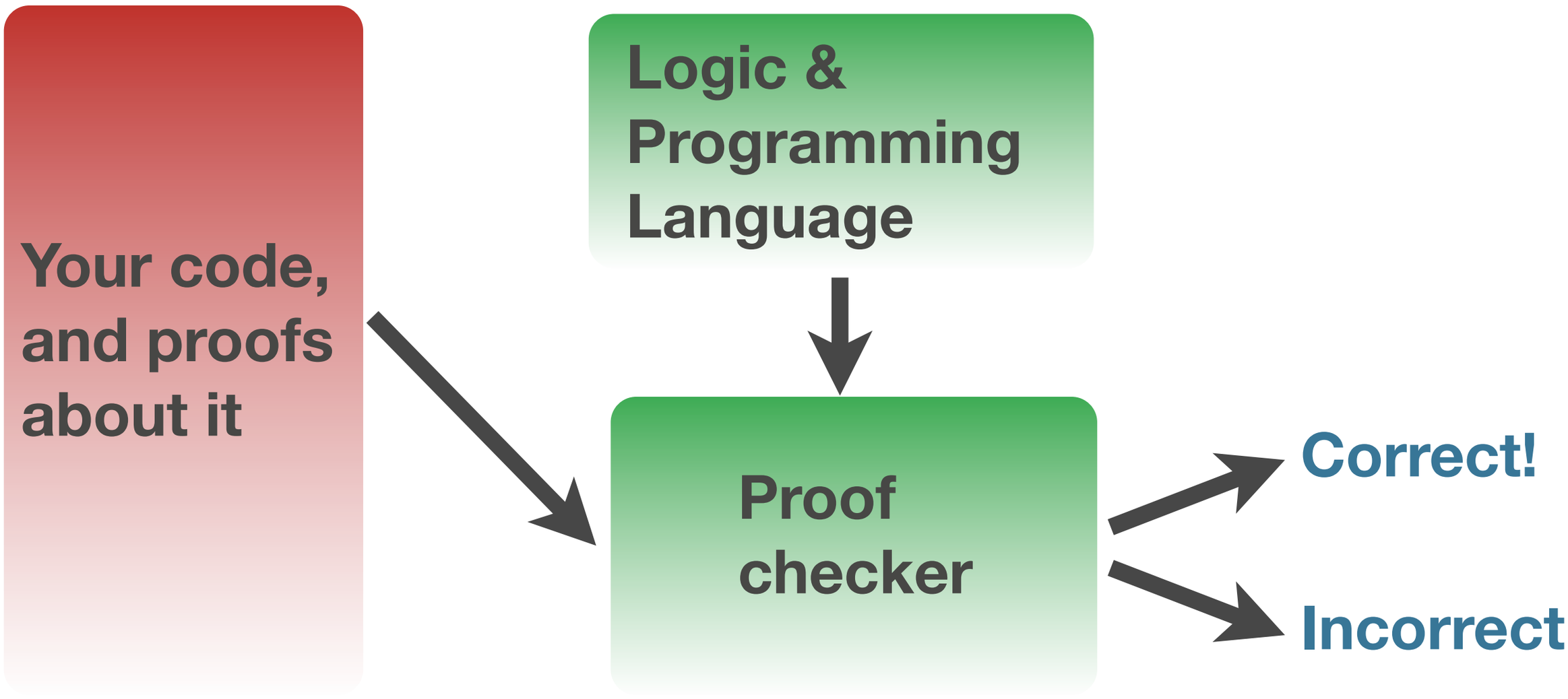
In 2003, after 4 years' work,
12 referees had checked lots of lemmas,
but gave up on verifying the proof

"This paper has brought about a change in the journal's policy on computer proof. ***It will no longer attempt to check the correctness of computer code****."*

Computer-checked math



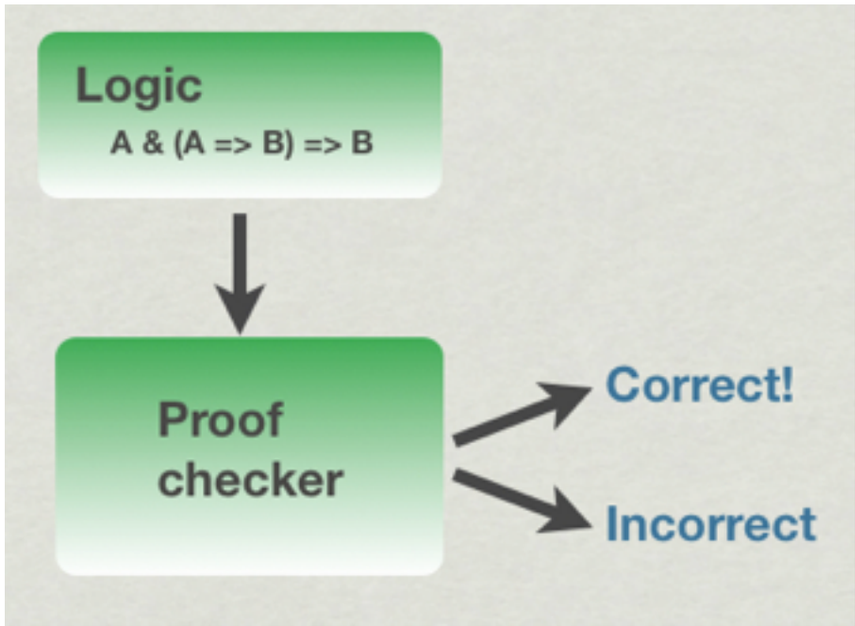
Computer-checked software



Computer-assisted proofs

Proof assistant

- Interactive proof editor
- Automated proofs
- Libraries



Kepler proof (85% done)

Informal

- 300 pages of math + 15,000 lines of code
- 15 hours to run

Computer-checked

- 350,000 lines of math + code
- >2 years to run

Kepler proof (85% done)

Informal

- 300 pages of math + 15,000 lines of code
- 15 hours to run

Computer-checked

- 350,000 lines of math + code
- >2 years to run

~5-10x longer

Kepler proof (85% done)

Informal

- 300 pages of math + 15,000 lines of code
- 15 hours to run

Computer-checked

- 350,000 lines of math + code **~5-10x longer**
- >2 years to run **~2000x slower**

Kepler proof (85% done)

Informal

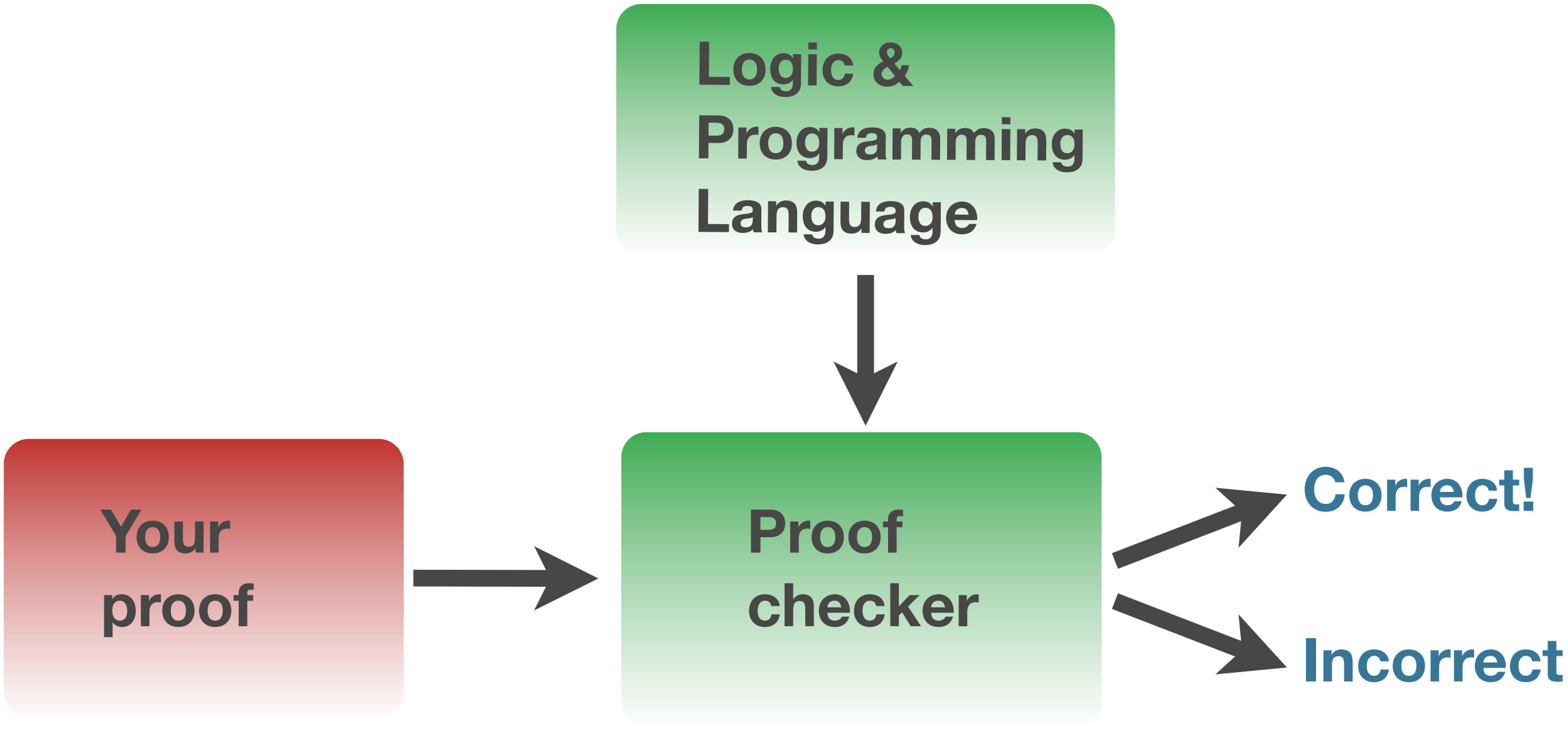
- 300 pages of math + 15,000 lines of code
- 15 hours to run

Computer-checked

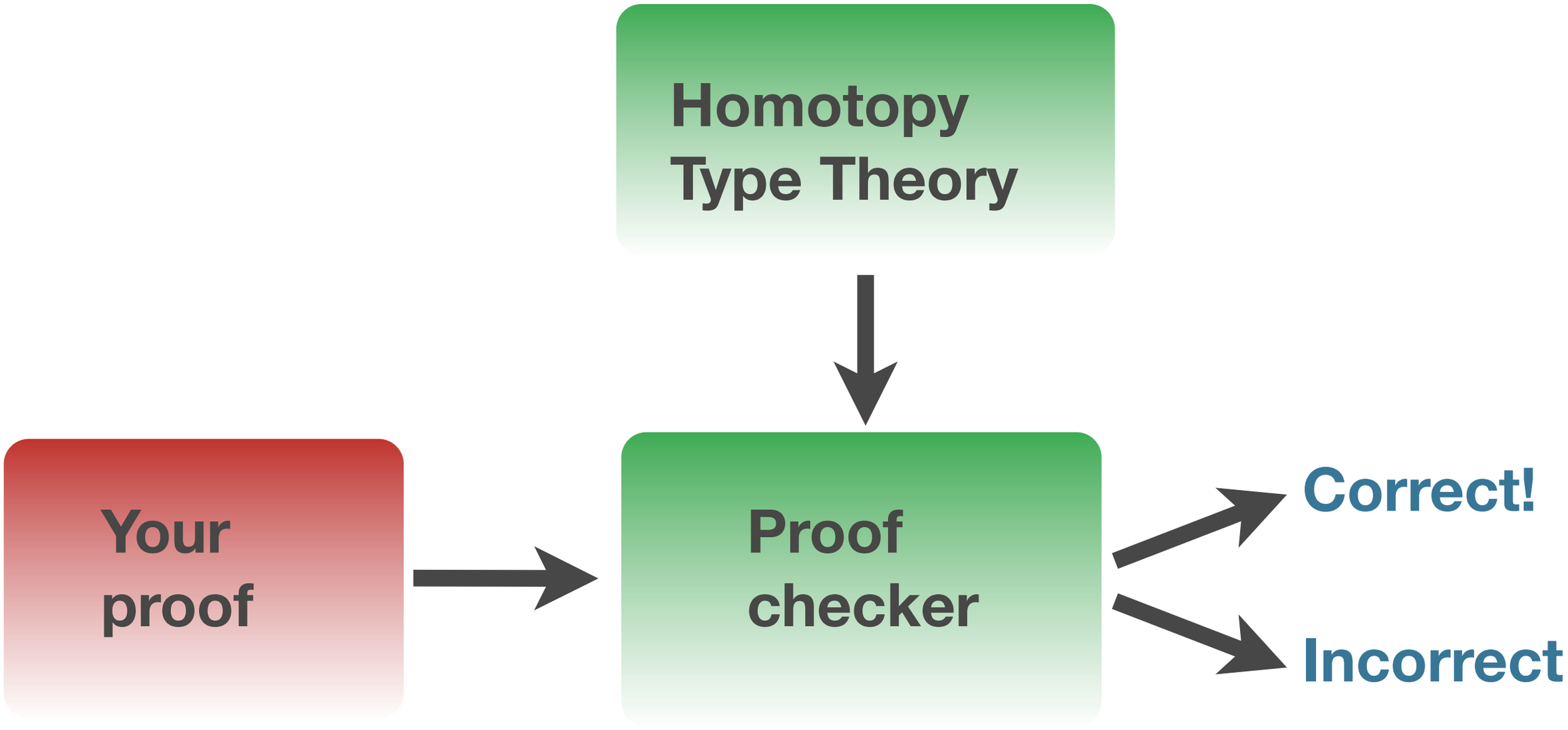
- 350,000 lines of math + code **~5-10x longer**
- >2 years to run **~2000x slower**

We have some work to do!

Homotopy Type Theory



Homotopy Type Theory



Type Theory

Basis of many successful proof assistants (Agda, Coq, NuPRL, Twelf)

- Functional programming language

```
insertsort : list<int> → list<int>
mergesort  : list<int> → list<int>
```

- **Unifies programming and proving:**
  types are rich enough to do math/verification

Propositions as Types

1. A theorem is represented by a type
2. Proof is represented by a program of that type

```
∀x. mergesort(x) = insertsort(x)
```

type **of proofs of program equality**

Propositions as Types

1. A theorem is represented by a type
2. Proof is represented by a program of that type

```
proof : ∀x. mergesort(x) = insertsort(x)
```

type **of proofs of program equality**

Propositions as Types

1. A theorem is represented by a type
2. Proof is represented by a program of that type

```
proof : ∀x. mergesort(x) = insertsort(x)
proof x = case x of
            [] => reflexivity
            (x :: xs) => ...
```

proof by case analysis represented by a function defined by cases

Type are sets?

Traditional view:

type theory	**set theory**
`<program> : <type>`	`x ∈ S`
`<prog1> = <prog2>`	`x = y`

Type are sets?

Traditional view:

type theory	**set theory**
`<program> : <type>`	`x ∈ S`
`<prog1> = <prog2>`	`x = y`

In set theory, an equation is a *proposition*:
it holds or it doesn't; we don't ask *why* 1+1=2

Type are sets?

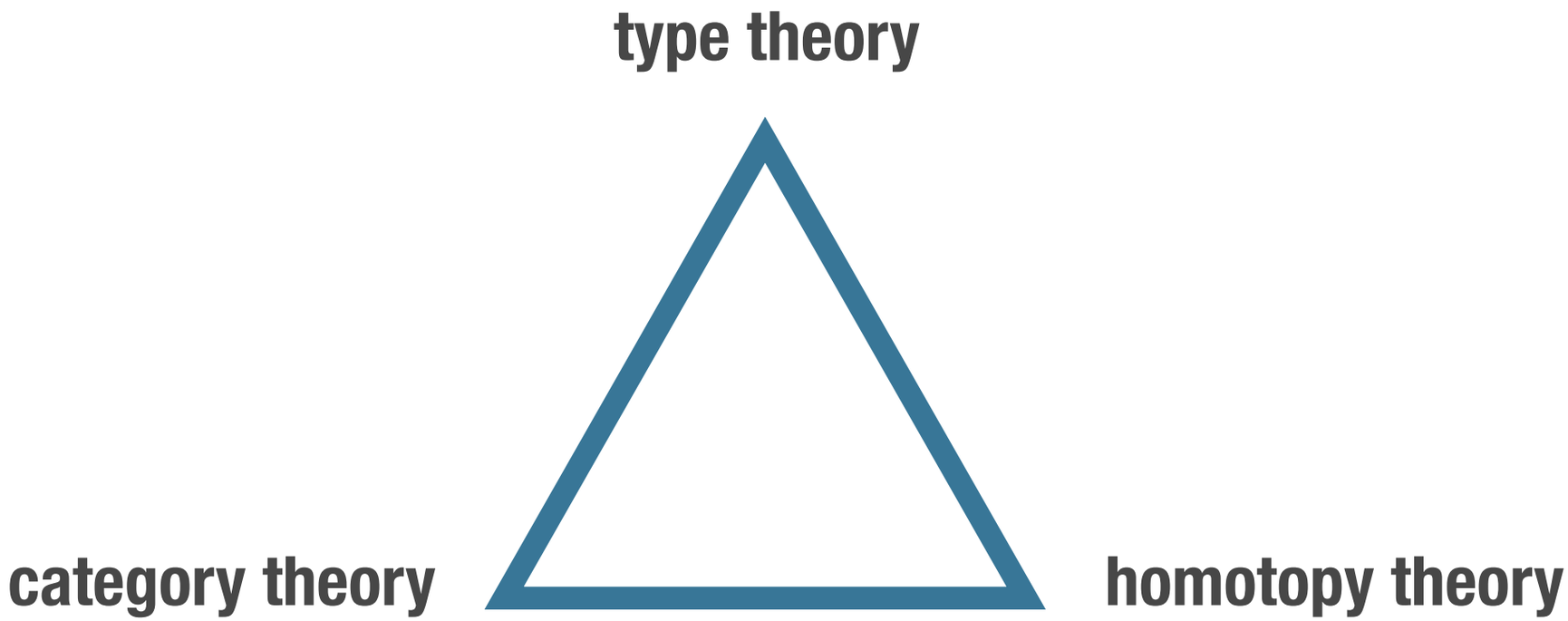
Traditional view:

type theory	**set theory**
`<program> : <type>`	`x ∈ S`
`<proof> : <prog1> = <prog2>`	`x = y`

In set theory, an equation is a *proposition*:
it holds or it doesn't; we don't ask *why* 1+1=2

In (intensional) type theory, an equation has
a non-trivial `<proof>`

Homotopy Type Theory



Types are ∞-groupoids

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

Types are ∞-groupoids

type theory	**set theory**
<program> : <type>	x ∈ S
<proof> : <prog$_1$> = <prog$_2$>	x = y

Types are ∞-groupoids

type theory	**set theory**
$\langle \text{program} \rangle : \langle \text{type} \rangle$	$x \in S$
$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$	$x = y$
$\langle \text{2-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$	

Types are ∞-groupoids

type theory	set theory
`<program> : <type>`	$x \in S$
`<proof> : <prog1> = <prog2>`	$x = y$
`<2-proof> : <proof1> = <proof2>`	
`<3-proof> : <2-proof1> = <2-proof2>`	

Types are ∞-groupoids

type theory	set theory
$\langle \text{program} \rangle : \langle \text{type} \rangle$	$x \in S$
$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$	$x = y$
$\langle \text{2-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$	
$\langle \text{3-proof} \rangle : \langle \text{2-proof}_1 \rangle = \langle \text{2-proof}_2 \rangle$	
⋮	

Types are ∞-groupoids

type theory	**set theory**
<program> : <type>	x ∈ S
<proof> : <prog$_1$> = <prog$_2$>	x = y
<2-proof> : <proof$_1$> = <proof$_2$>	
<3-proof> : <2-proof$_1$> = <2-proof$_2$>	
⋮	

Proofs, 2-proofs, 3-proofs, …
all influence how a program runs

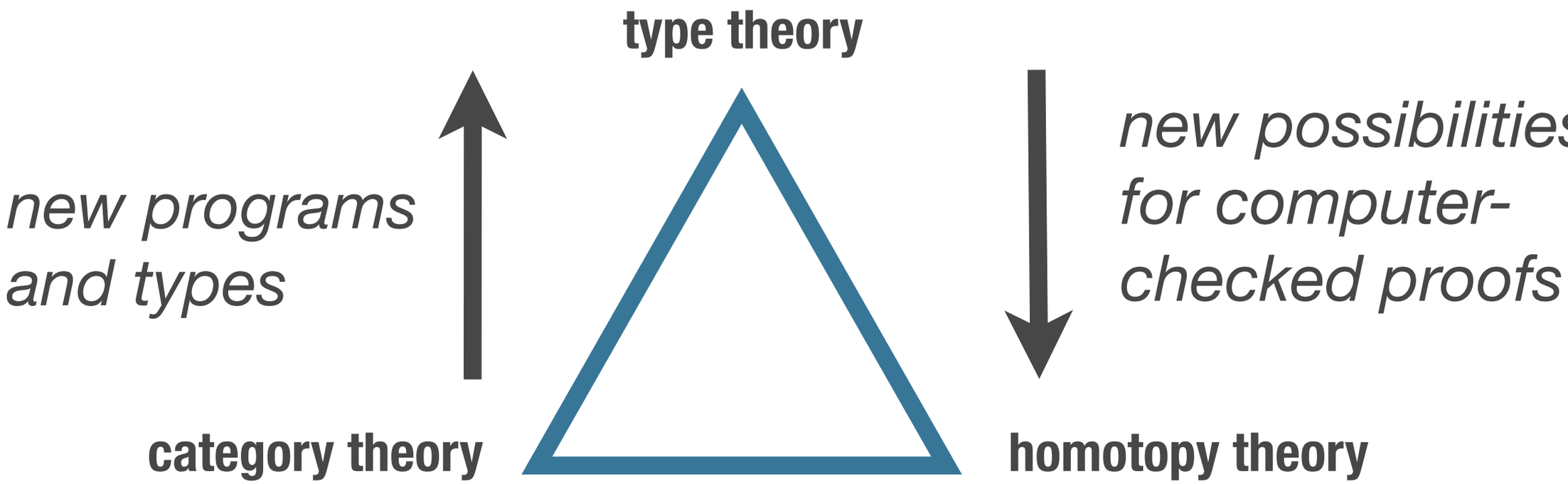
Types are ∞-groupoids

type theory	**set theory**
<program> : <type>	x ∈ S
<proof> : <prog$_1$> = <prog$_2$>	x = y
<2-proof> : <proof$_1$> = <proof$_2$>	
<3-proof> : <2-proof$_1$> = <2-proof$_2$>	
⋮	

Proofs, 2-proofs, 3-proofs, …
all influence how a program runs

∞-*groupoid*: each level has a group(oid) structure, and they interact…

Homotopy Type Theory



Outline

1. Computer-checked homotopy theory

2. Computer-checked software

Outline

1.Computer-checked homotopy theory

2.Computer-checked software

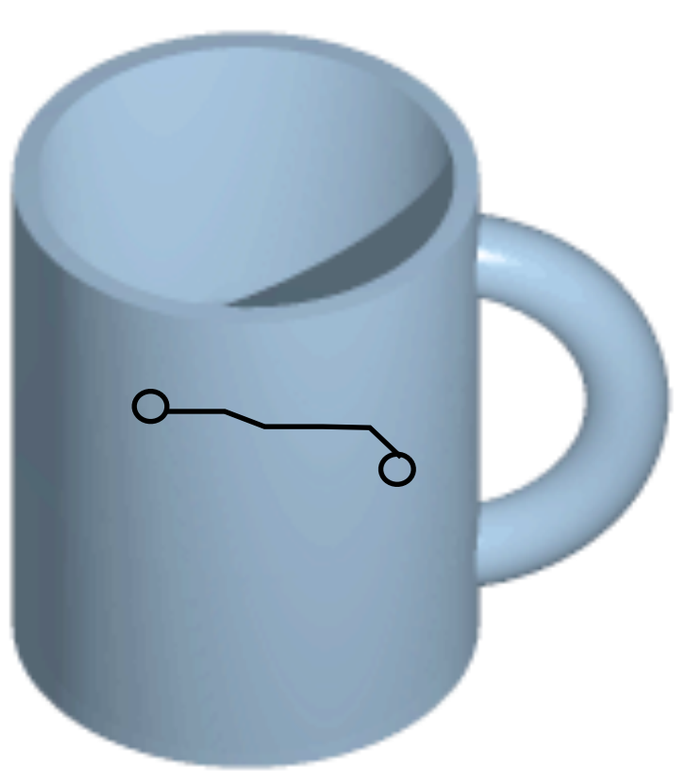
Homotopy Theory

A branch of topology,
the study of spaces and continuous deformations

[image from wikipedia]

Homotopy Theory

A branch of topology,
the study of spaces and continuous deformations



[image from wikipedia]

Homotopy

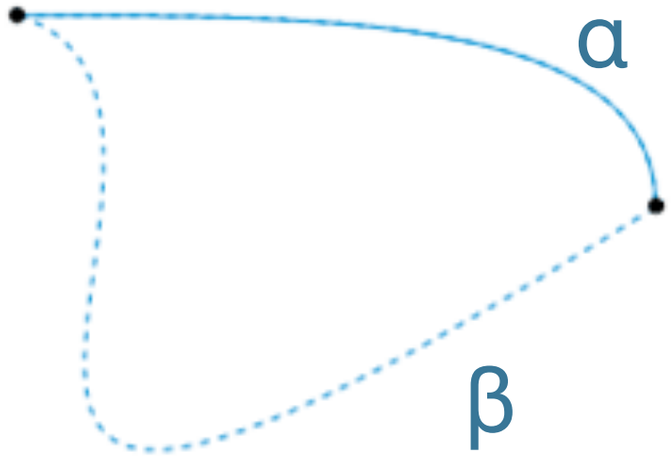
Deformation of one path into another

α

β

Homotopy

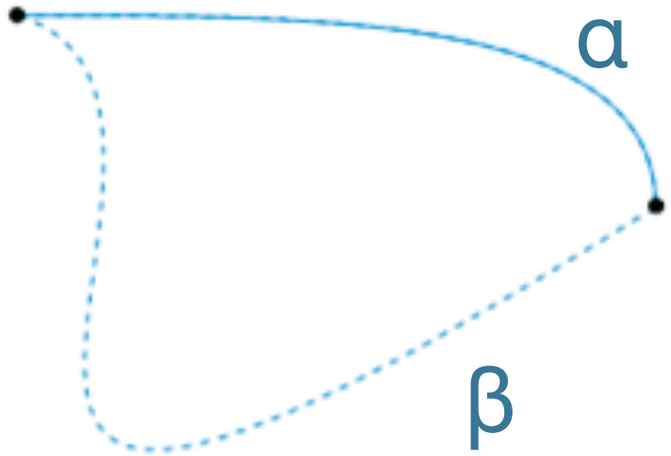
Deformation of one path into another



[image from wikipedia]

Homotopy

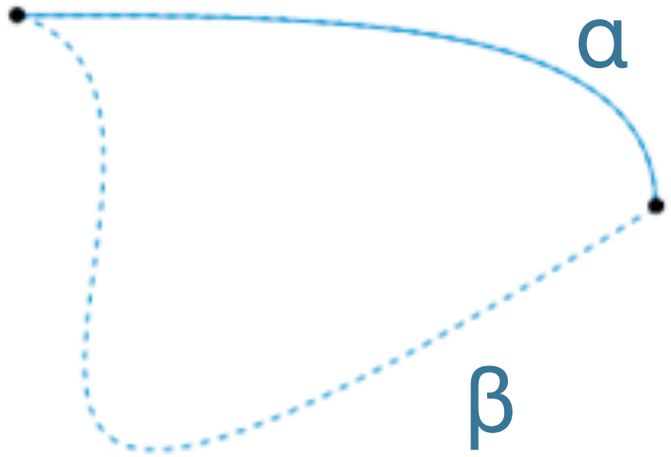
Deformation of one path into another



[image from wikipedia]

Homotopy

Deformation of one path into another

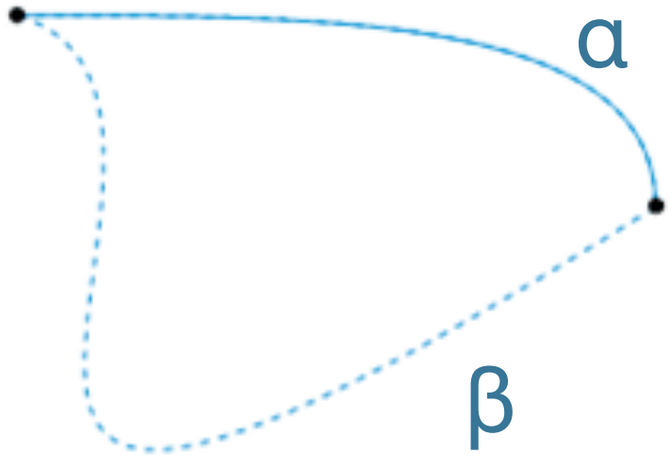


= 2-dimensional *path between paths*

[image from wikipedia]

Homotopy

Deformation of one path into another



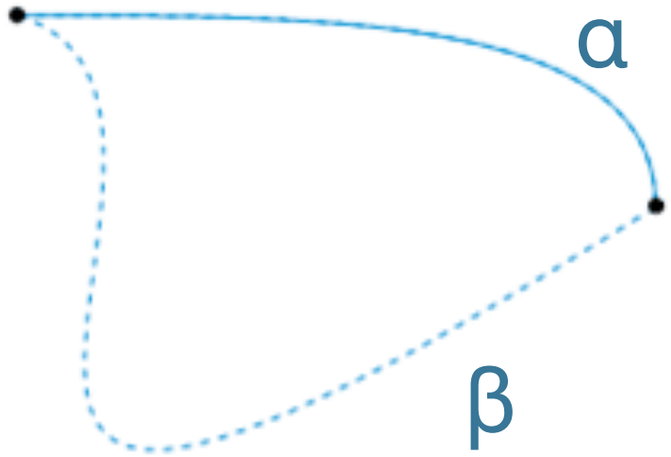
<2-proof> : α = β

= 2-dimensional *path between paths*

[image from wikipedia]

Homotopy

Deformation of one path into another



<2-proof> : α = β

= 2-dimensional *path between paths*

Homotopy theory is the study of spaces by way of their paths, homotopies, homotopies between homotopies, ….

[image from wikipedia]

Synthetic vs Analytic

Synthetic geometry (Euclid)

POSTULATES.

I.

Let it be granted that a straight line may be drawn from any one point to any other point.

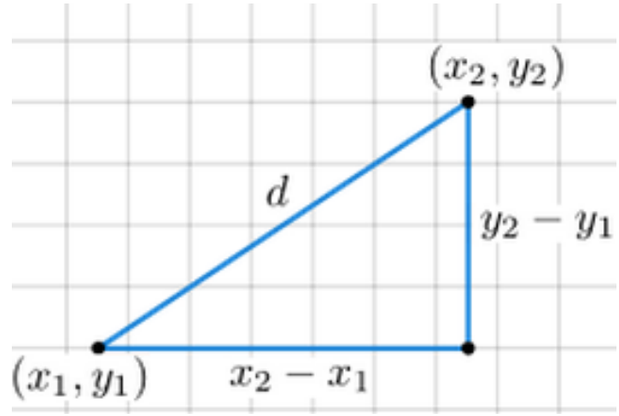
II.

That a terminated straight line may be produced to any length in a straight line.

III.

And that a circle may be described from any centre, at any distance from that centre.

Analytic geometry (Descartes)



[image from wikipedia]

Synthetic vs Analytic

Synthetic geometry (Euclid)

POSTULATES.

I.

Let it be granted that a straight line may be drawn from any one point to any other point.

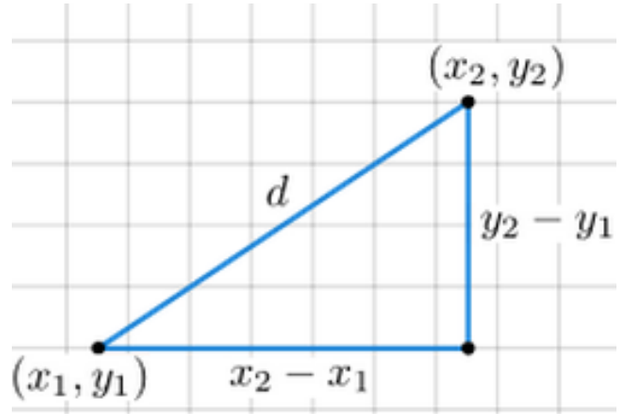
II.

That a terminated straight line may be produced to any length in a straight line.

III.

And that a circle may be described from any centre, at any distance from that centre.

Analytic geometry (Descartes)



Classical homotopy theory is analytic:

- a space is a set of points equipped with a topology
- a path is a set of points, given continuously

[image from wikipedia]

Synthetic homotopy theory

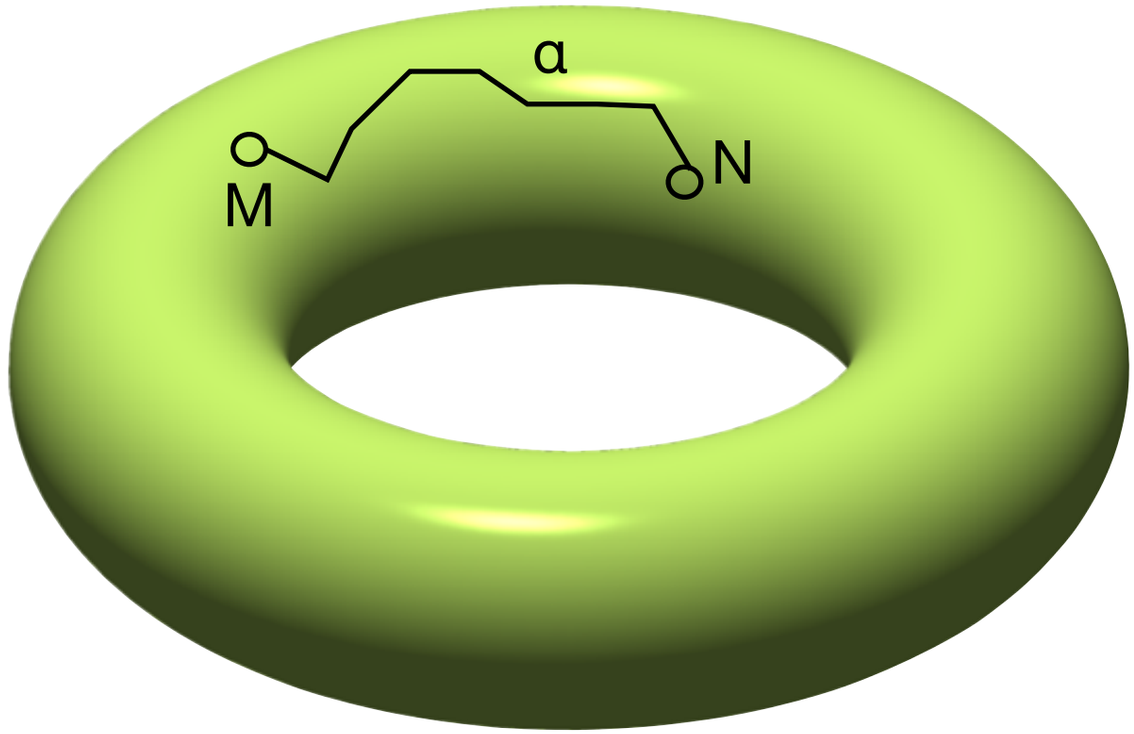
homotopy theory	**type theory**
space	`<type>`
points	`<program> : <type>`
paths	`<proof> : <prog`$_1$`> = <prog`$_2$`>`
homotopies	`<2-proof> : <proof`$_1$`> = <proof`$_2$`>`
⋮	⋮

Synthetic homotopy theory

homotopy theory	**type theory**
space	`<type>`
points	`<program> : <type>`
paths	`<proof> : <prog`$_1$`> = <prog`$_2$`>`
homotopies	`<2-proof> : <proof`$_1$`> = <proof`$_2$`>`
⋮	⋮

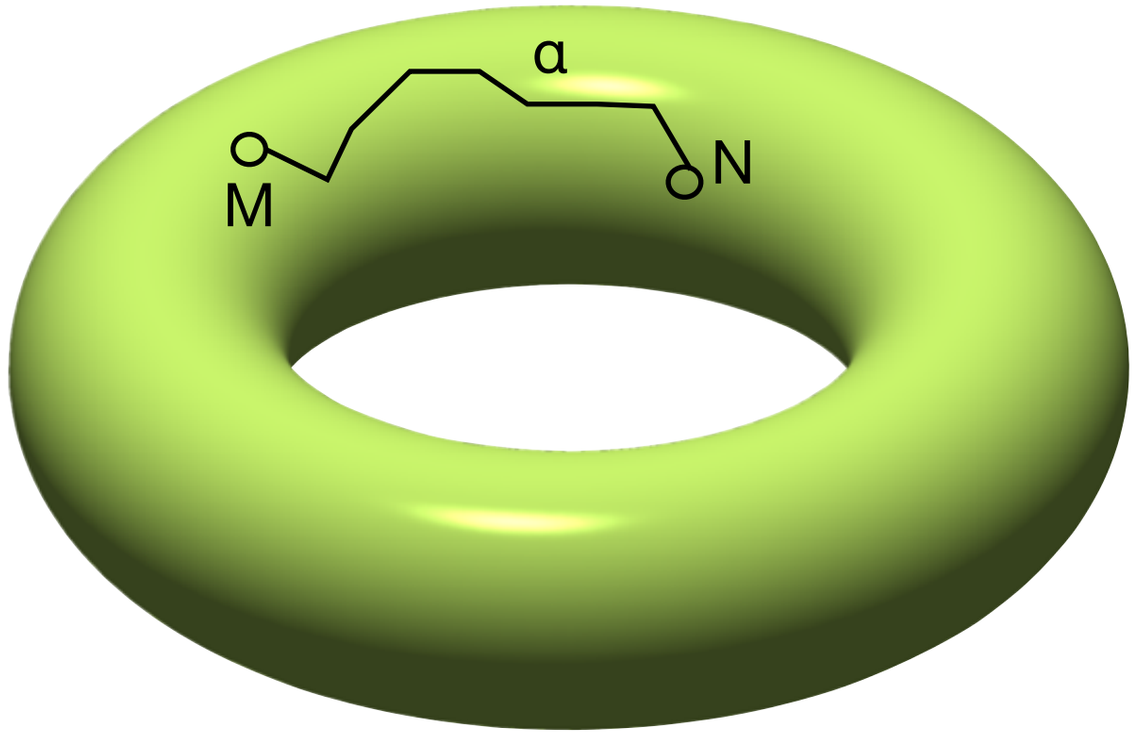
*A path is **not** a set of points; it is a primitive notion*

Spaces as types



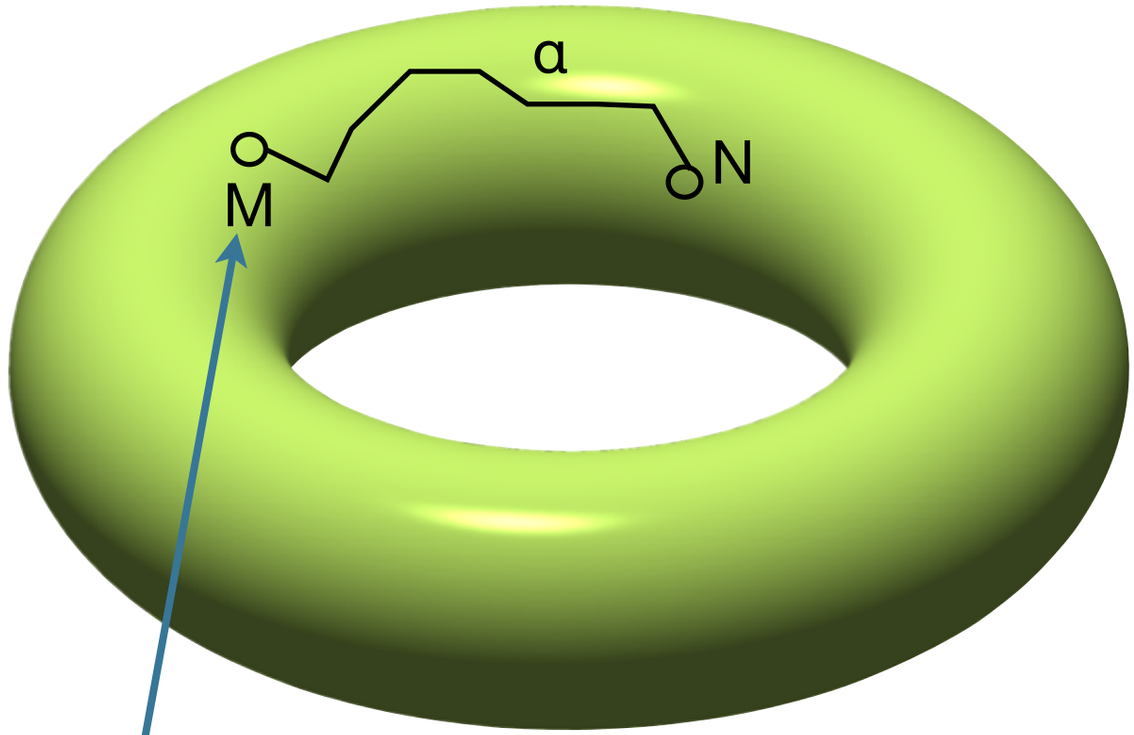
Spaces as types

a space is a type A



Spaces as types

a space is a type A

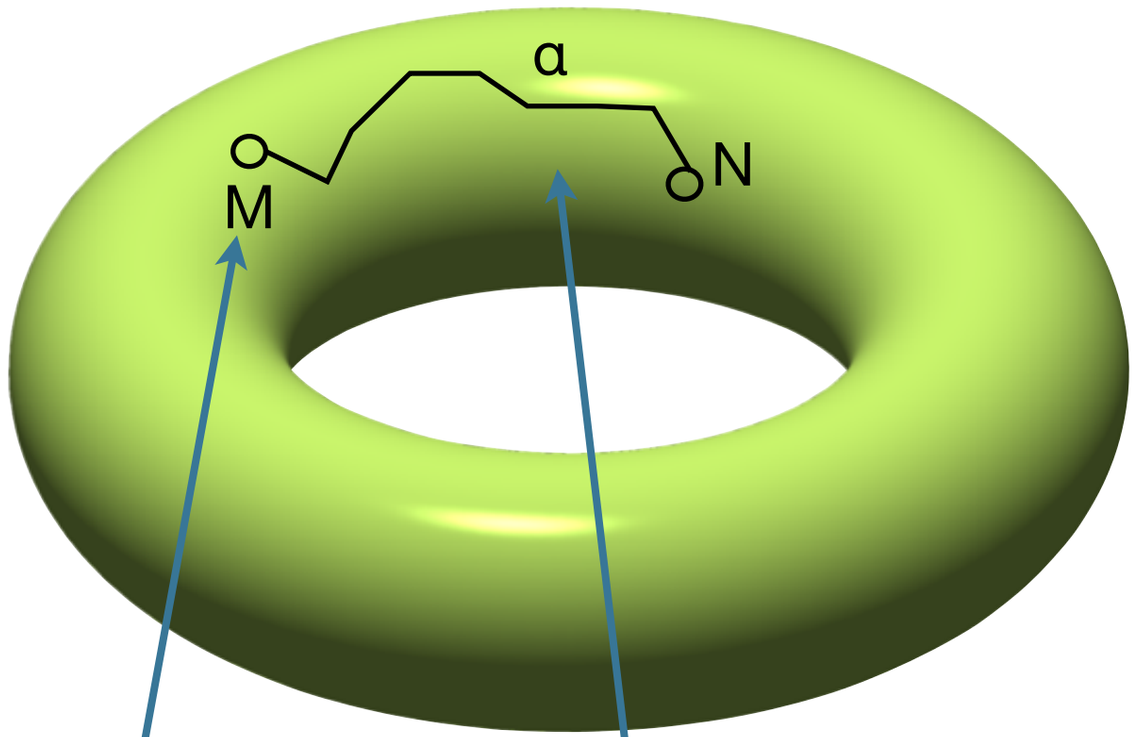


points are programs

M : A

Spaces as types

a space is a type A



points are programs

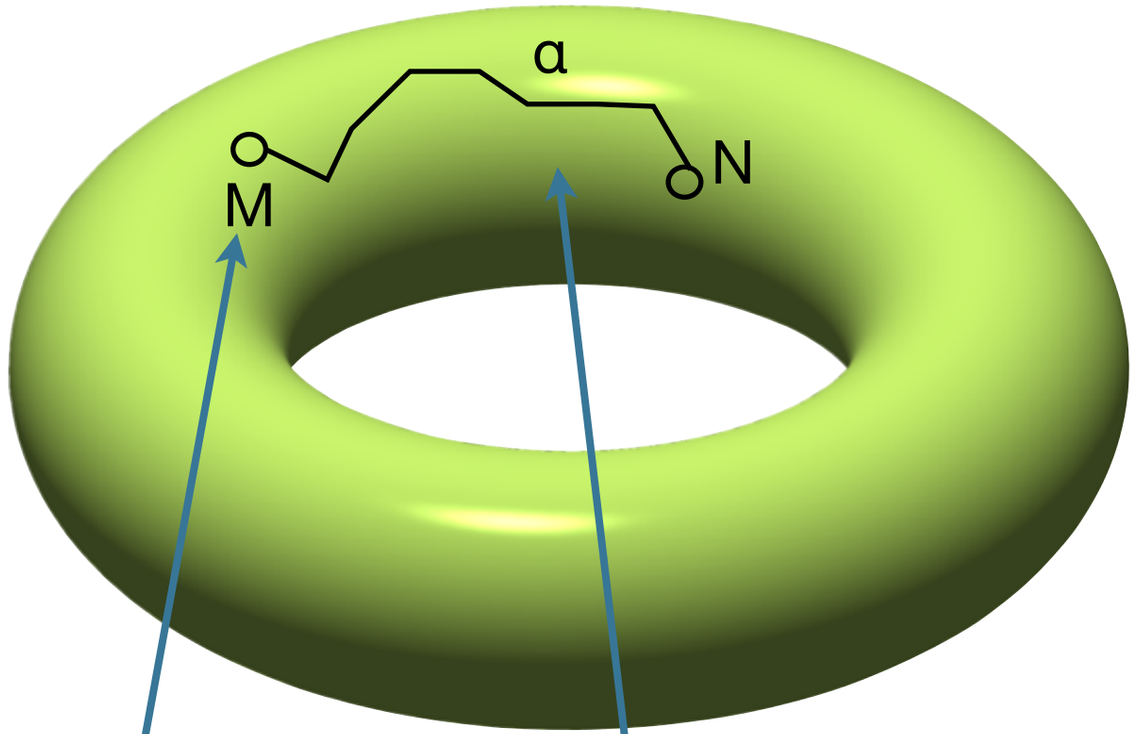
M : A

paths are *proofs of equality*

$\alpha : M =_A N$

Spaces as types

a space is a type A



points are programs

M : A

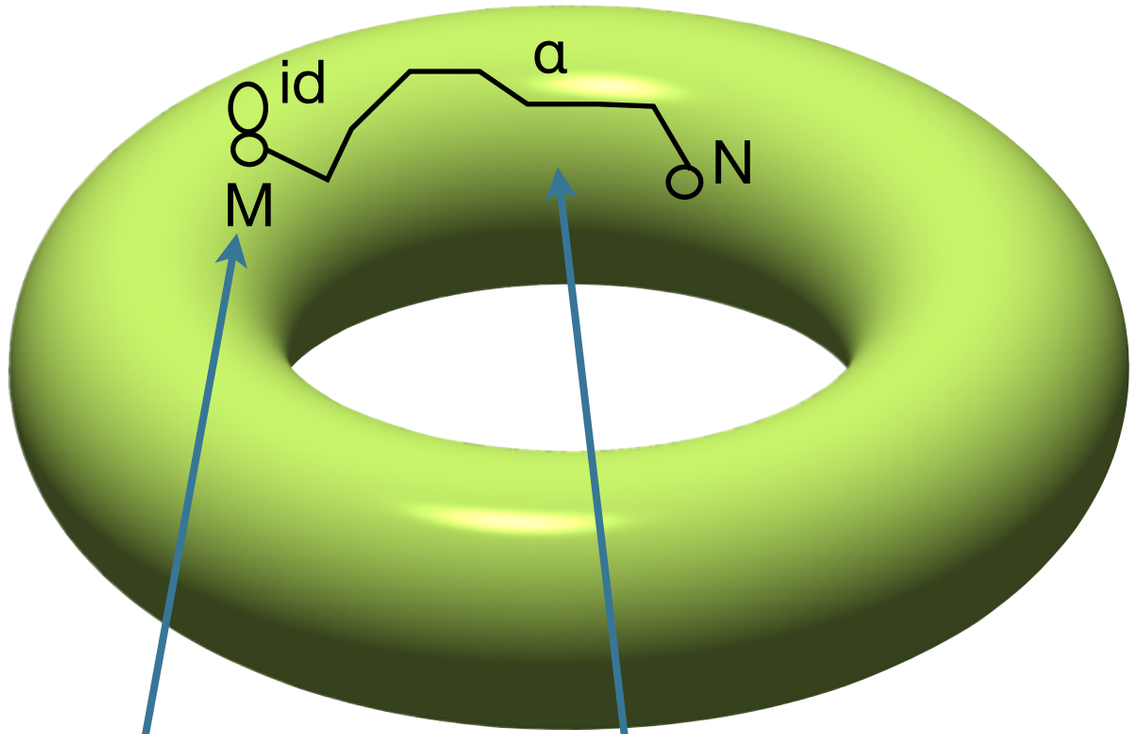
paths are *proofs of equality*

$\alpha : M =_A N$

path operations

Spaces as types

a space is a type A



points are programs

M : A

paths are *proofs of equality*

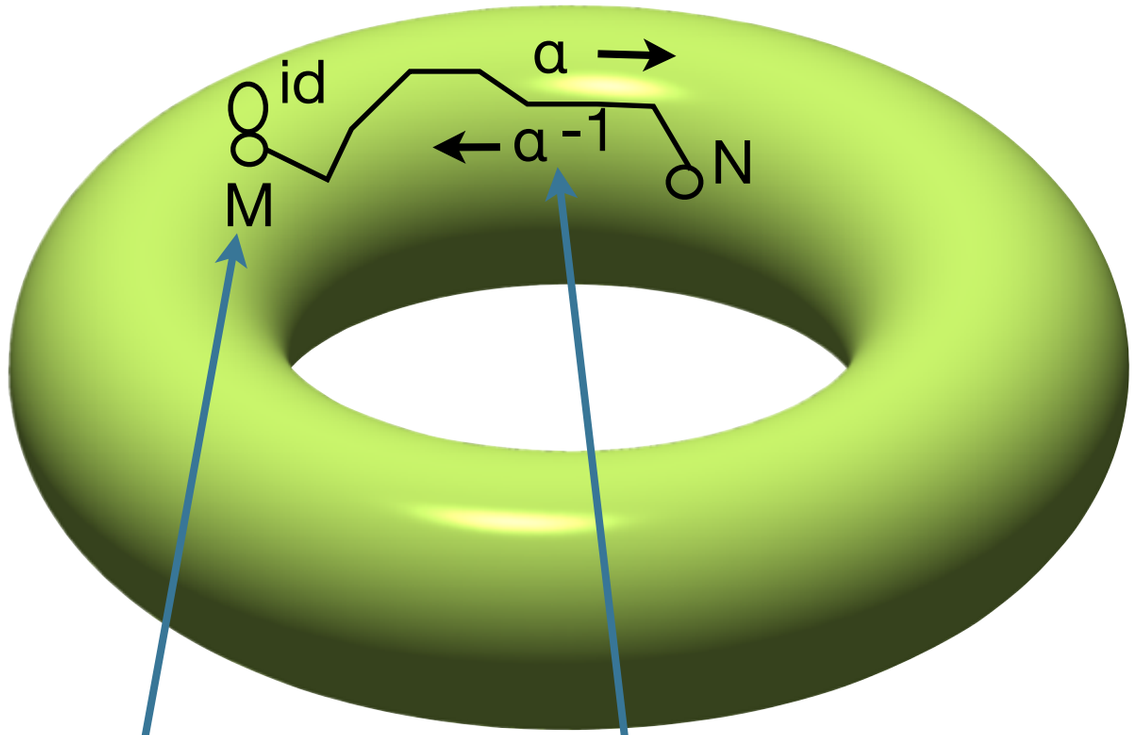
$\alpha : M =_A N$

path operations

```
id      : M = M (refl)
```

Spaces as types

a space is a type A



points are programs

M : A

paths are *proofs of equality*

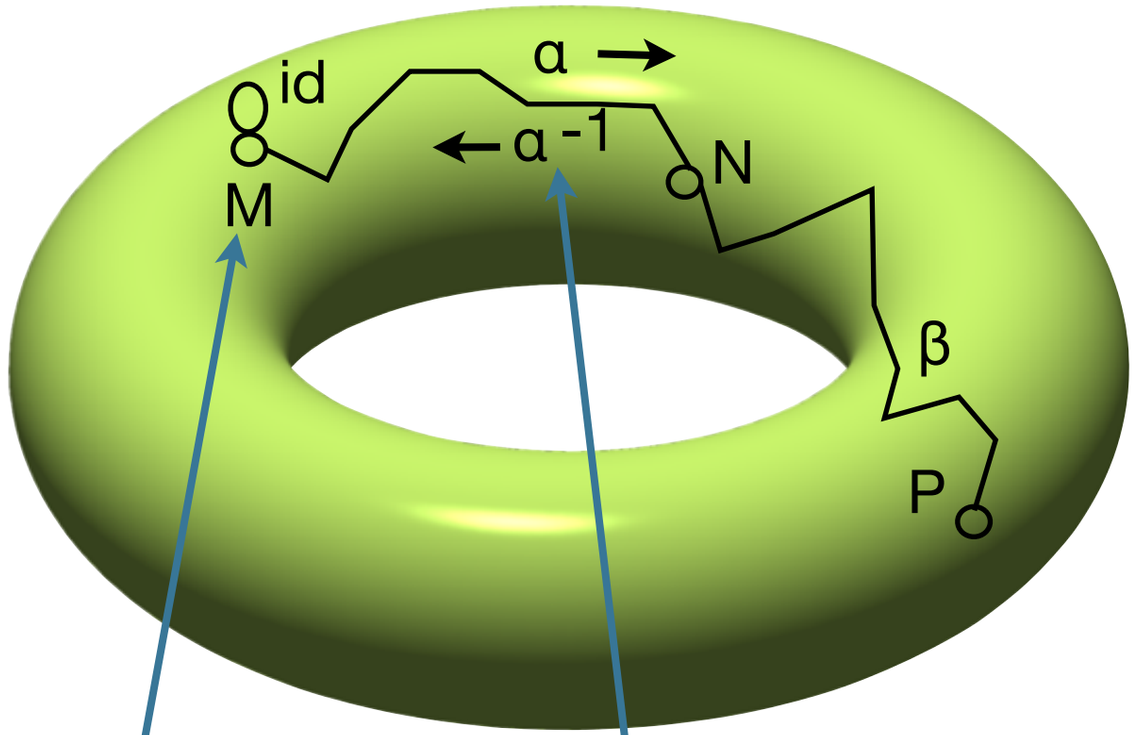
$\alpha : M =_A N$

path operations

```
id      : M = M (refl)
α⁻¹     : N = M (sym)
```

Spaces as types

a space is a type A



points are programs

M : A

paths are *proofs of equality*

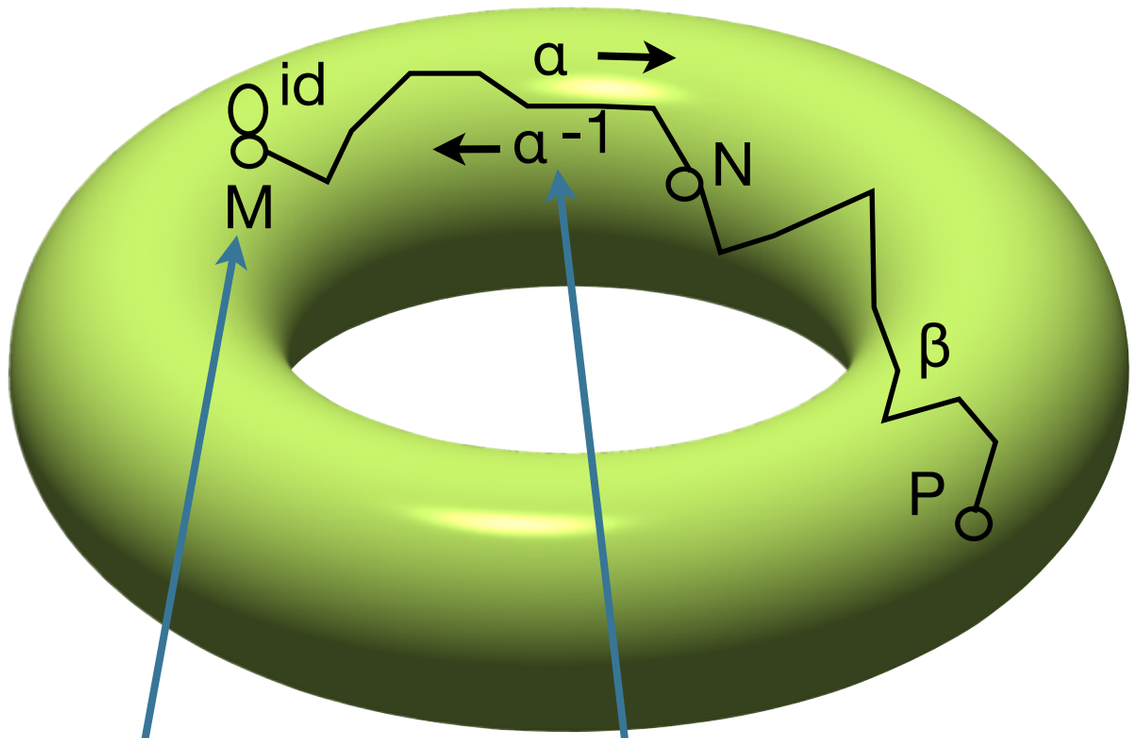
$\alpha : M =_A N$

path operations

```
id      : M = M (refl)
α⁻¹     : N = M (sym)
β o α   : M = P (trans)
```

Spaces as types

a space is a type A



points are programs

M : A

paths are *proofs of equality*

$\alpha : M =_A N$

path operations

```
id      : M = M (refl)
α⁻¹     : N = M (sym)
β o α   : M = P (trans)
```

homotopies

```
id o p = p
p⁻¹ o p = id
r o (q o p)
   = (r o q) o p
```

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

$\pi_4(S^3) = \mathbb{Z}_2$

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]

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

$\pi_4(S^3) = \mathbb{Z}_2$

Freudenthal

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

K(G,n)

Cohomology axioms

Blakers-Massey

Van Kampen

Covering spaces

Whitehead for n-types

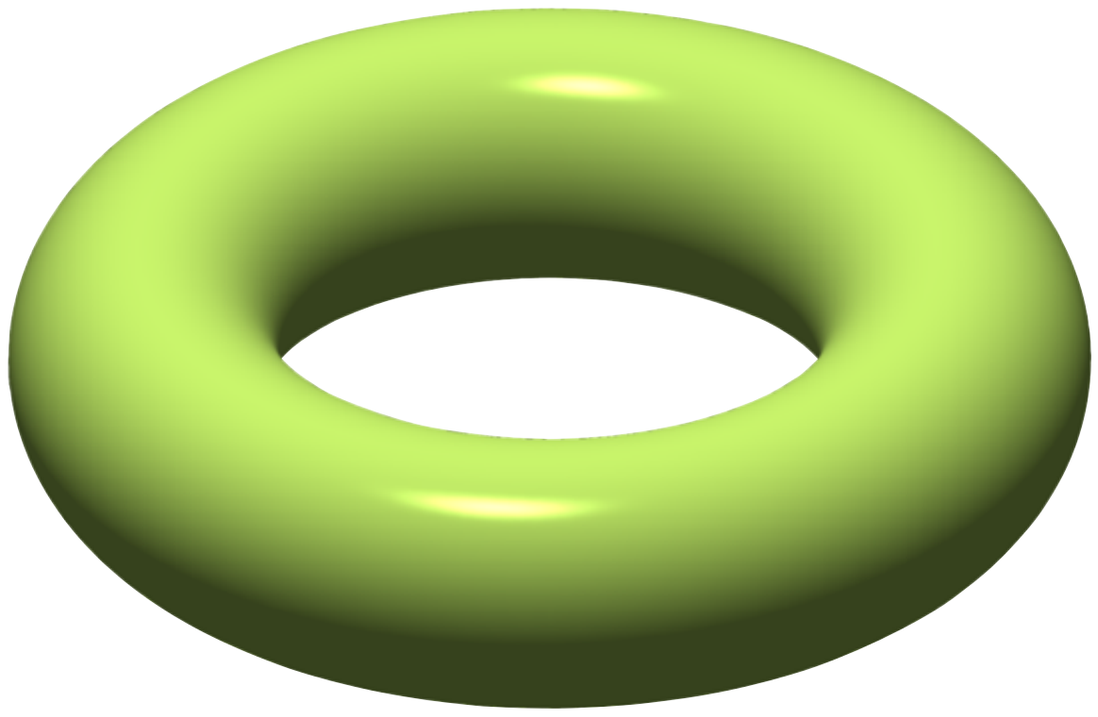
[Brunerie, Finster, Hou, Licata, Lumsdaine, Shulman]

Homotopy Groups

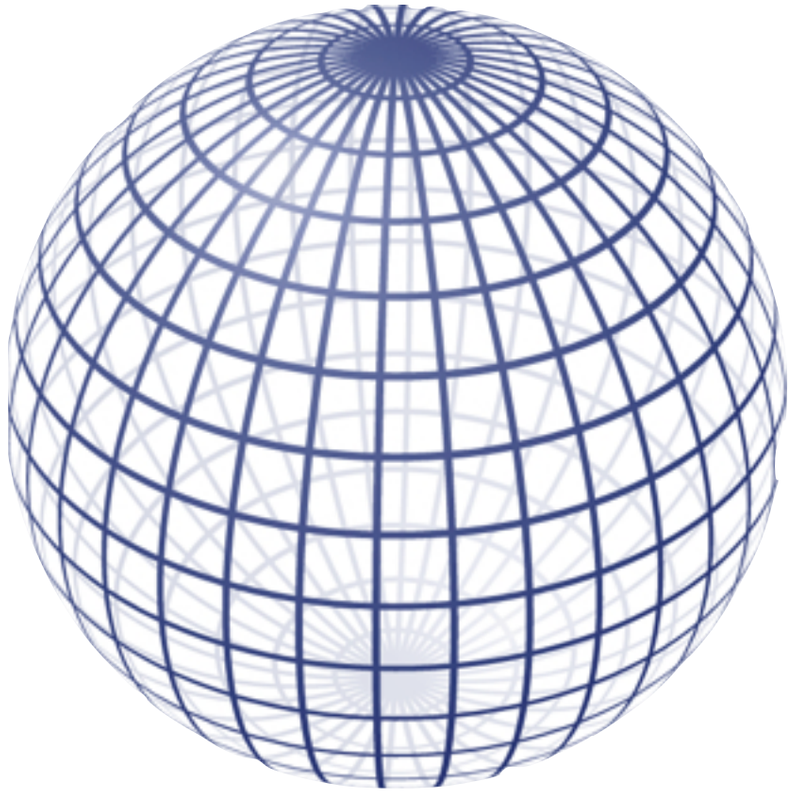
Homotopy groups of a space X:

- $\pi_1(X)$ is fundamental group (group of loops)
- $\pi_2(X)$ is group of *homotopies* (2-dimensional loops)
- $\pi_3(X)$ is group of 3-dimensional loops
- …

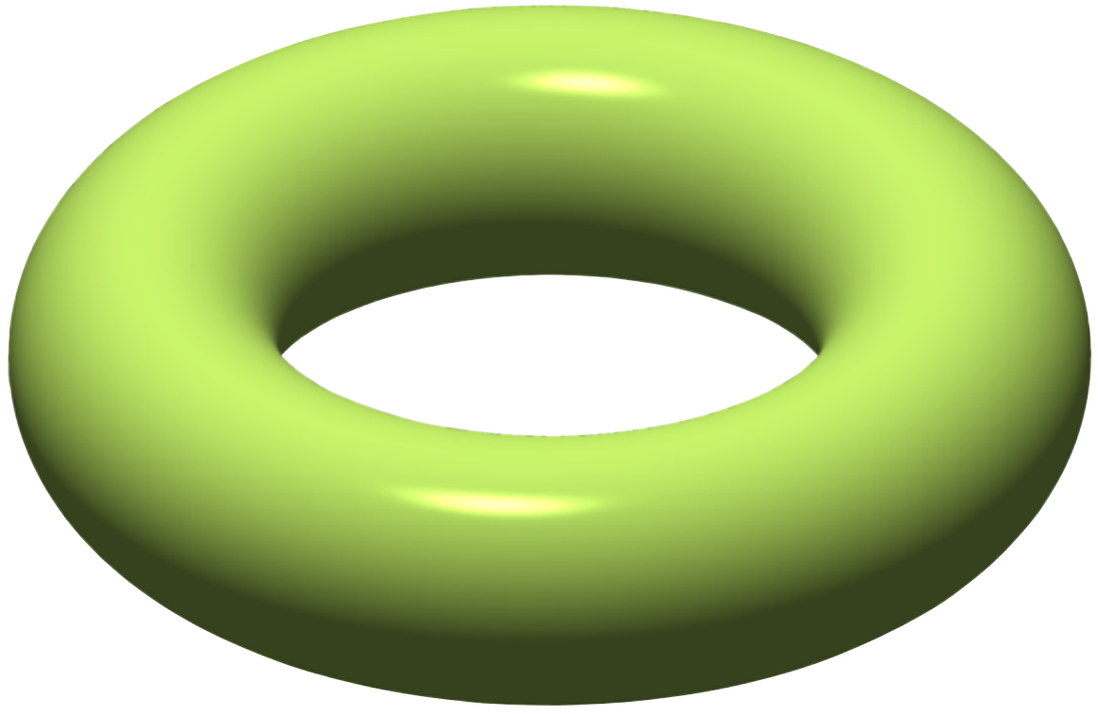
Telling spaces apart



$\neq$

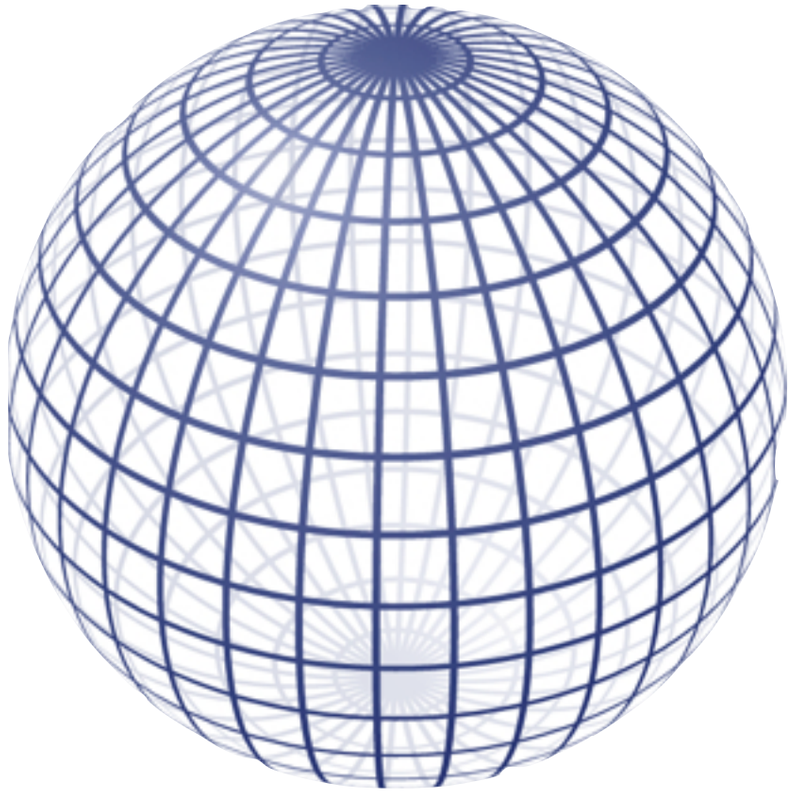


Telling spaces apart



fundamental group
is non-trivial ($\mathbb{Z} \times \mathbb{Z}$)

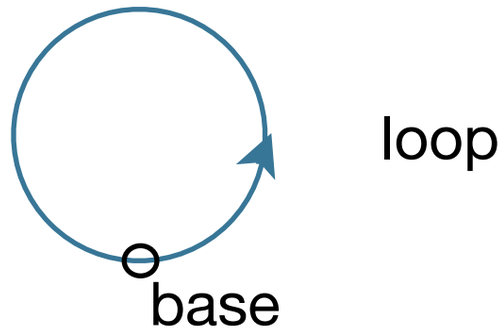
$\neq$



fundamental group
is trivial

The Circle

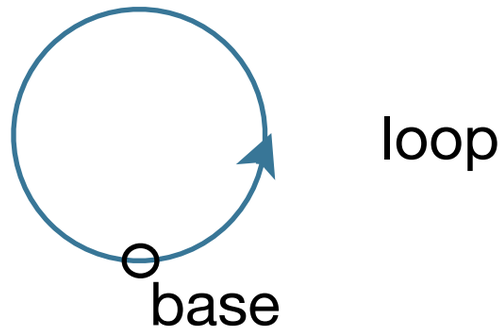
Circle S^1 is a **higher inductive type** generated by



The Circle

Circle S^1 is a **higher inductive type** generated by

$$\begin{aligned} \mathtt{base} &: S^1 \\ \mathtt{loop} &: \mathtt{base} = \mathtt{base} \end{aligned}$$

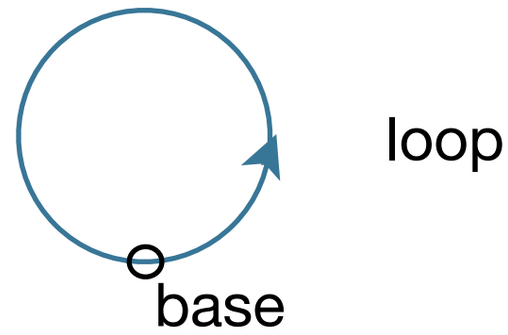


The Circle

Circle S^1 is a **higher inductive type** generated by

point $\quad base : S^1$

$\quad loop : base = base$

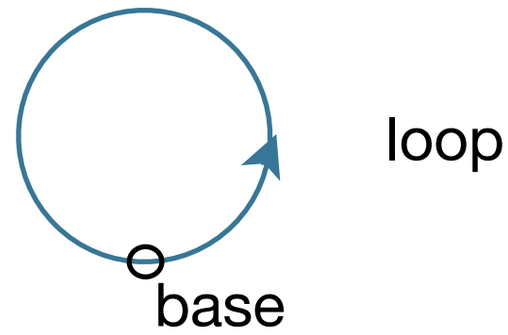


The Circle

Circle S^1 is a **higher inductive type** generated by

point `base : S¹`

path `loop : base = base`

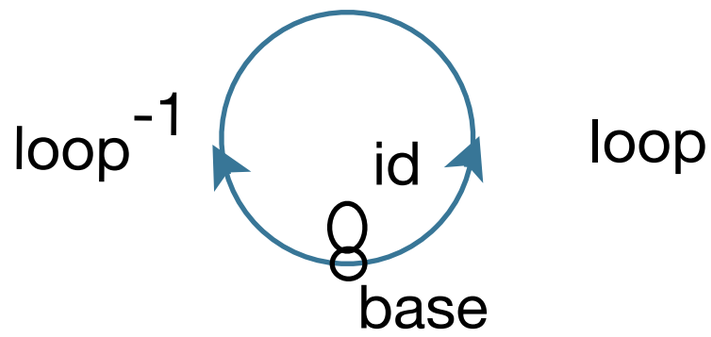


The Circle

Circle S^1 is a **higher inductive type** generated by

point `base : S¹`

path `loop : base = base`



Free type: equipped with structure

`id`

`loop⁻¹`

`loop o loop`

`inv : loop o loop⁻¹ = id`

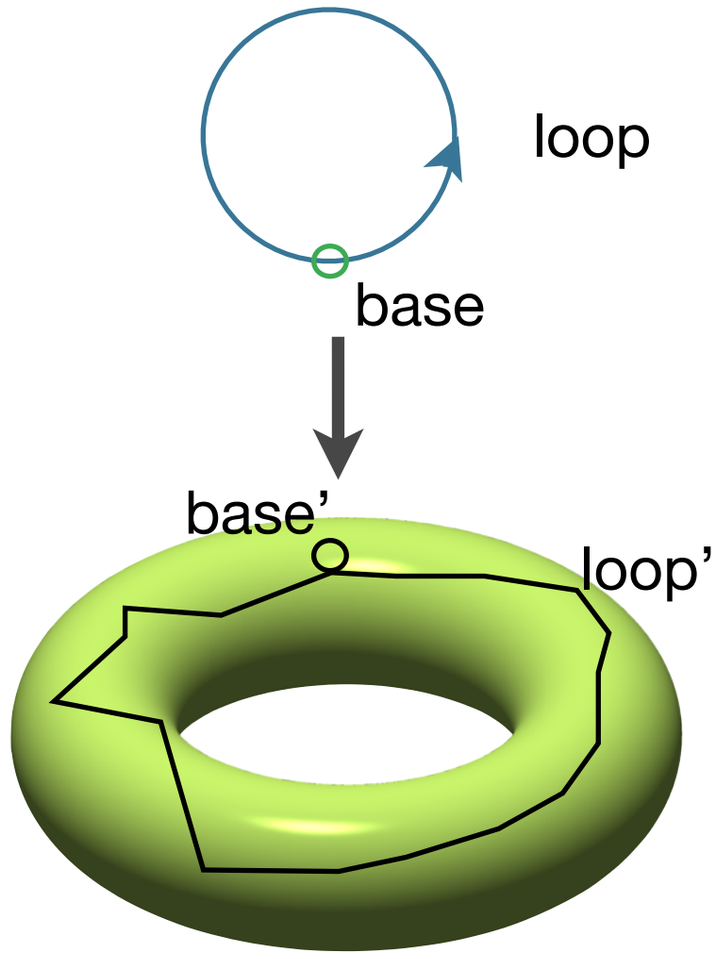
`...`

The Circle

Circle recursion:

function $S^1 \to X$ determined by

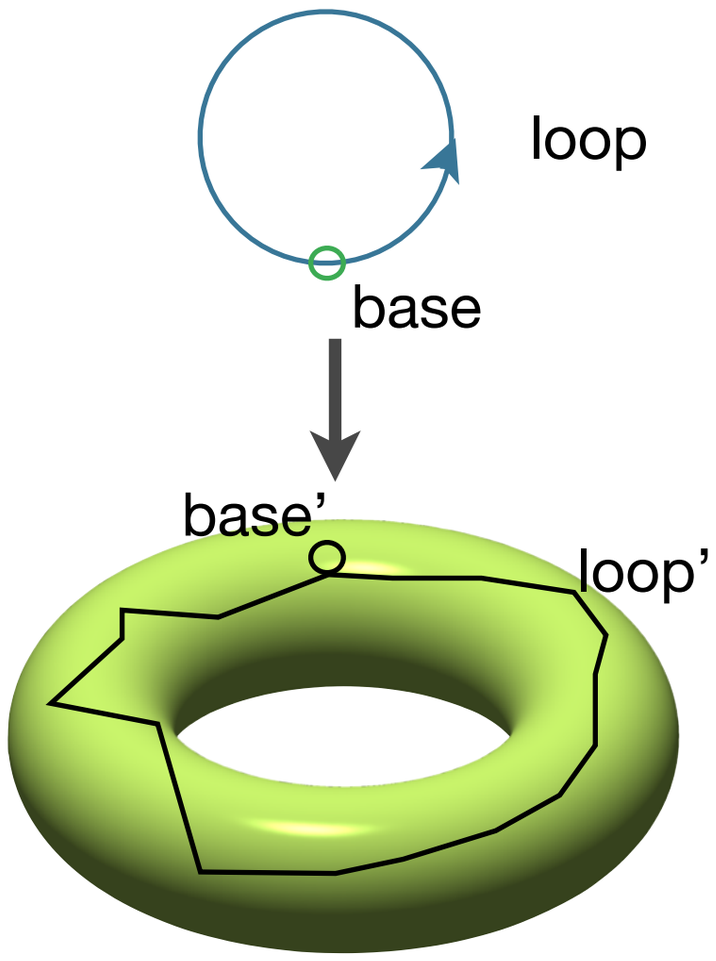
```
base' : X
loop' : base' = base'
```



The Circle

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

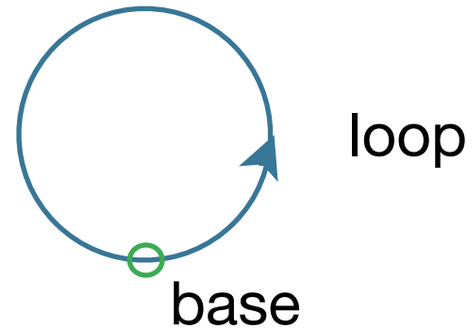
```
base' : X
loop' : base' = base'
```



Circle induction: To prove a predicate P for all points on the circle, suffices to prove P(base), continuously in the loop

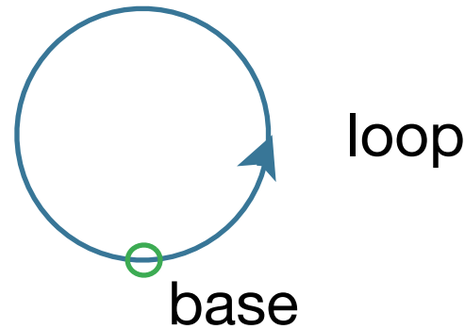
Fundamental group of circle

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



Fundamental group of circle

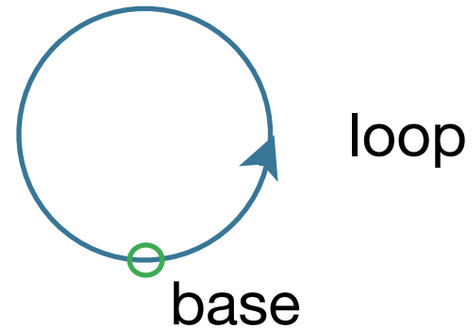
How many different loops are there on the circle, up to homotopy?




```
id
```

Fundamental group of circle

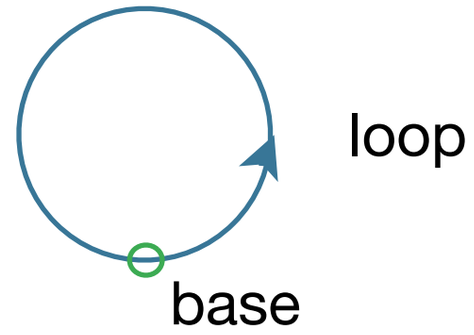
How many different loops are there on the circle, up to homotopy?




```
id
loop
```

Fundamental group of circle

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



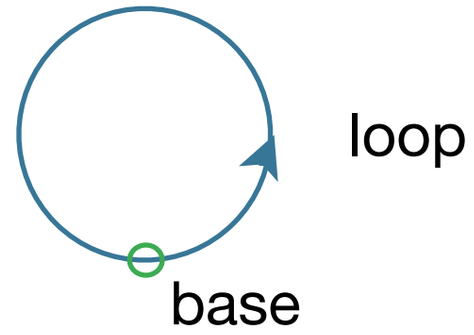
`id`

`loop`

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

Fundamental group of circle

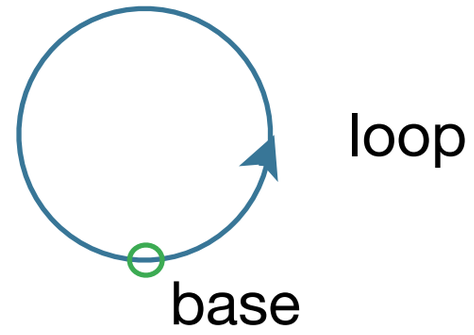
How many different loops are there on the circle, up to homotopy?




```
id
loop
loop⁻¹
loop o loop
```

Fundamental group of circle

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



`id`

`loop`

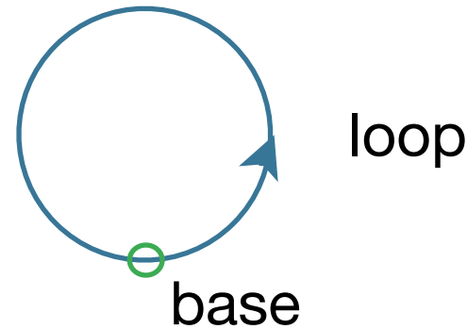
$\texttt{loop}^{-1}$

`loop o loop`

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

Fundamental group of circle

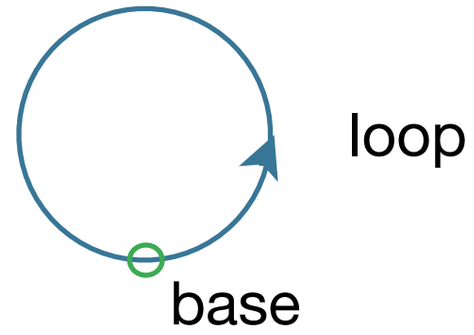
How many different loops are there on the circle, up to homotopy?




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

Fundamental group of circle

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



`id`

`loop`

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

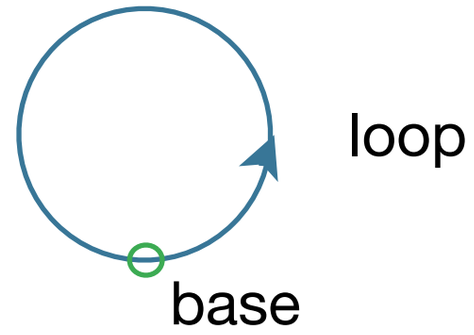
`loop o loop`

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

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

Fundamental group of circle

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



`id` `0`

`loop`

`loop⁻¹`

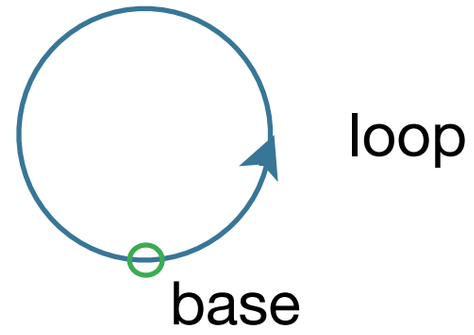
`loop o loop`

`loop⁻¹ o loop⁻¹`

`loop o loop⁻¹ = id`

Fundamental group of circle

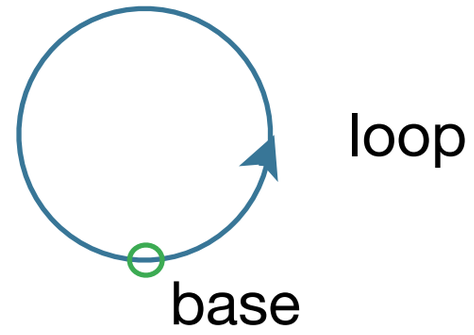
How many different loops are there on the circle, up to homotopy?



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

Fundamental group of circle

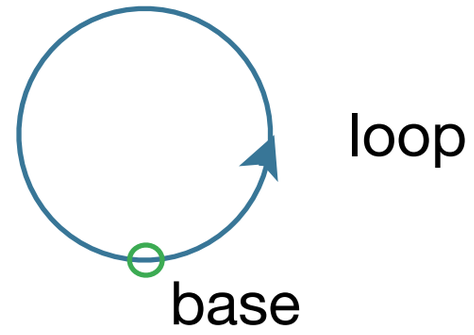
How many different loops are there on the circle, up to homotopy?



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

Fundamental group of circle

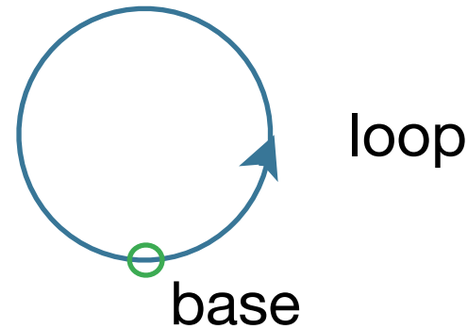
How many different loops are there on the circle, up to homotopy?



`id`	`0`
`loop`	`1`
`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?

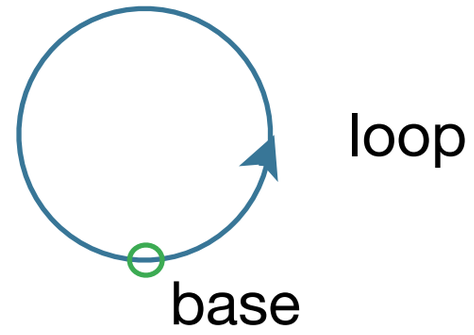


`id`	`0`
`loop`	`1`
`loop⁻¹`	`-1`
`loop o loop`	`2`
`loop⁻¹ o loop⁻¹`	`-2`

`loop o loop⁻¹ = id`

Fundamental group of circle

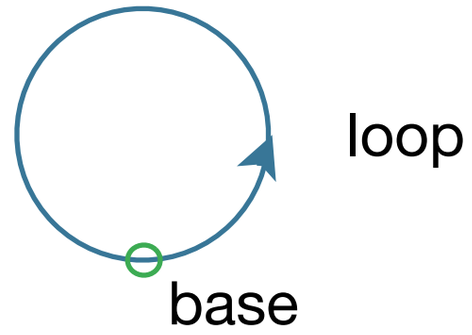
How many different loops are there on the circle, up to homotopy?



`id`	`0`
`loop`	`1`
`loop⁻¹`	`-1`
`loop o loop`	`2`
`loop⁻¹ o loop⁻¹`	`-2`
`loop o loop⁻¹ = id`	`0`

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`	`-2`
`loop o loop⁻¹ = id`	`0`

integers are “codes” for paths on the circle

Fundamental group of circle

Definition. $\Omega(S^1)$ is the **type** of loops at base
i.e. the type $(\mathtt{base} =_{S1} \mathtt{base})$

Fundamental group of circle

Definition. $\Omega(S^1)$ is the **type** of loops at base
i.e. the type $(\mathtt{base} =_{S1} \mathtt{base})$

Theorem. $\Omega(S^1)$ is equivalent to $\mathbb{Z}$,
by a map that sends o to +

Fundamental group of circle

Definition. $\Omega(S^1)$ is the **type** of loops at base
i.e. the type $(\mathtt{base} =_{S1} \mathtt{base})$

Theorem. $\Omega(S^1)$ is equivalent to $\mathbb{Z}$,
by a map that sends o to +

Corollary: Fundamental group
of the circle is isomorphic to $\mathbb{Z}$

Fundamental group of circle

Definition. $\Omega(S^1)$ is the **type** of loops at base
i.e. the type $(base =_{S1} base)$

Theorem. $\Omega(S^1)$ is equivalent to $\mathbb{Z}$,
by a map that sends o to +

Corollary: Fundamental group
of the circle is isomorphic to $\mathbb{Z}$

0-truncation (set of connected components) of $\Omega(S^1)$

Fundamental group of circle

Theorem. $\Omega(S^1)$ is equivalent to $\mathbb{Z}$

Proof (Shulman, L.): two mutually inverse functions

```
wind  : Ω(S¹) → ℤ

loop⁻ : ℤ → Ω(S¹)
```

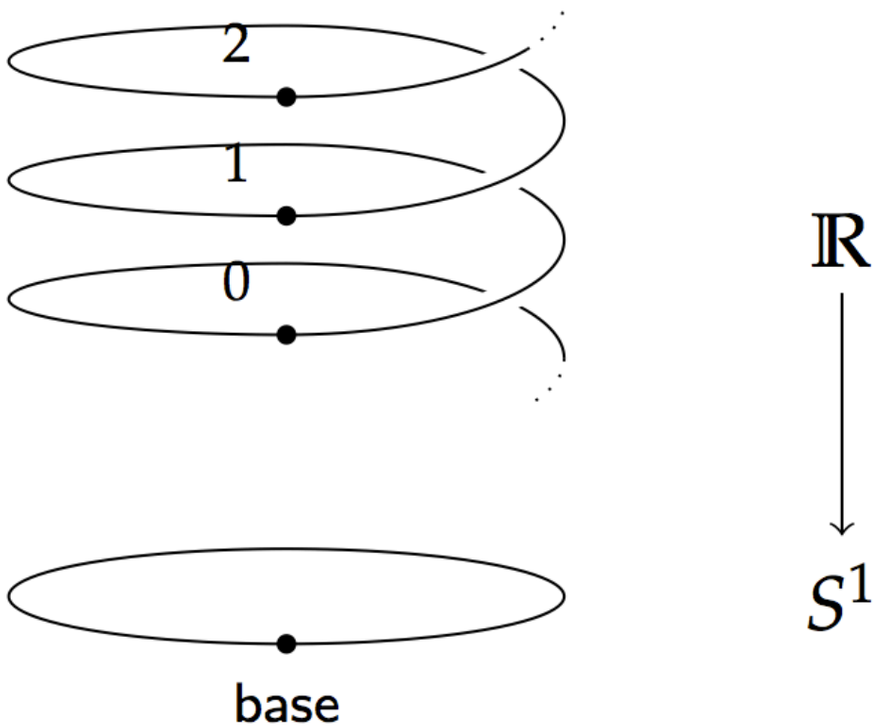
Fundamental group of circle

Theorem. $\Omega(S^1)$ is equivalent to $\mathbb{Z}$

Proof (Shulman, L.): two mutually inverse functions

```
wind   : Ω(S¹) → ℤ


loop⁻  : ℤ → Ω(S¹)
loop⁰   = id
loop⁺ⁿ  = loop o loop o … loop       (n times)
loop⁻ⁿ  = loop⁻¹ o loop⁻¹ o … loop⁻¹  (n times)
```

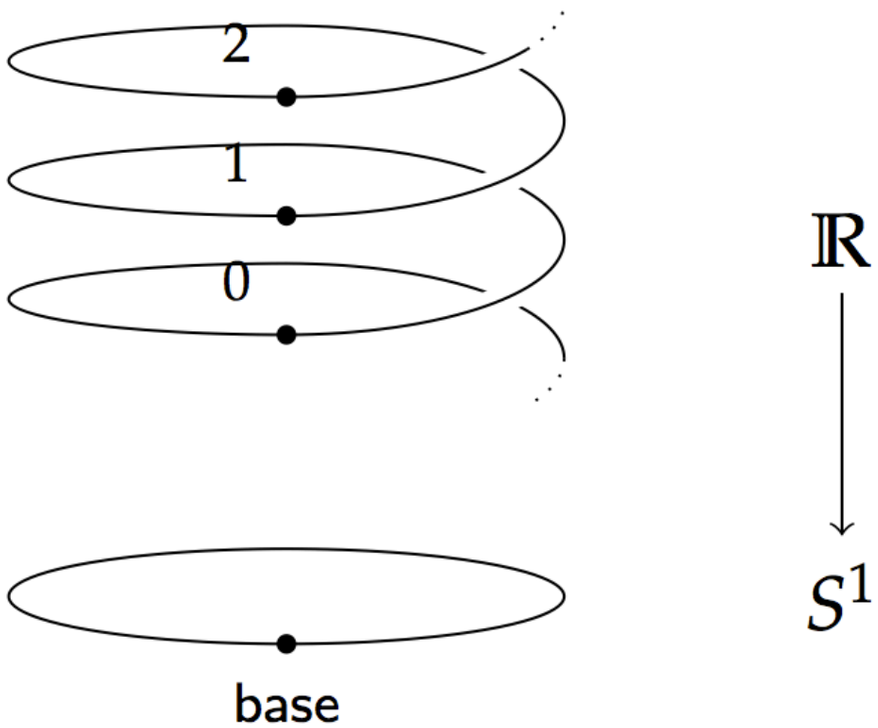
Universal Cover



$\texttt{wind} : \Omega(S^1) \to \mathbb{Z}$

defined by **lifting** a loop to the cover, and giving the other endpoint of 0

Universal Cover

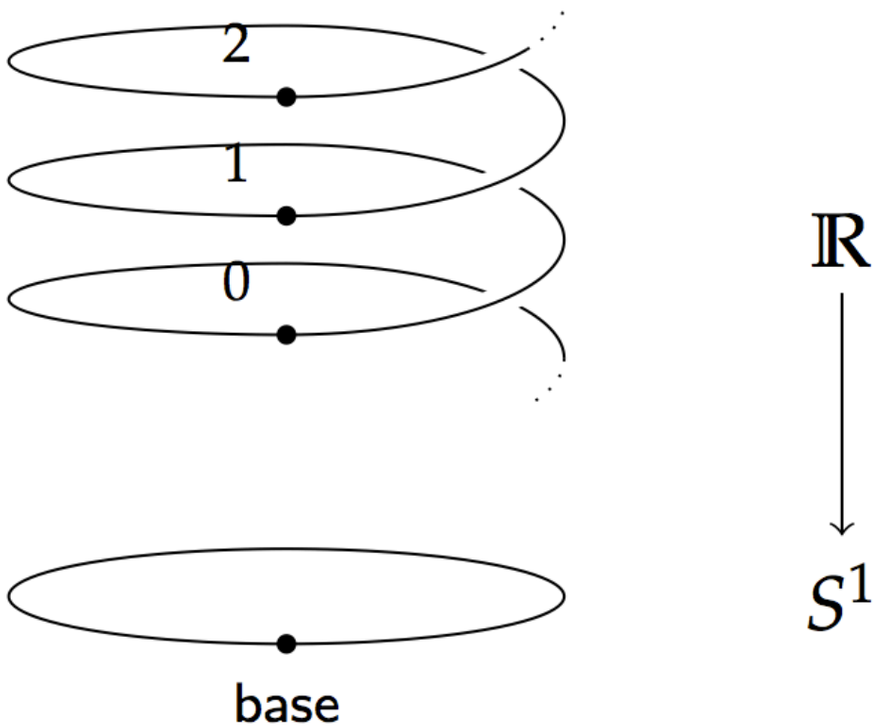


$\texttt{wind} : \Omega(S^1) \to \mathbb{Z}$

defined by **lifting** a loop to the cover, and giving the other endpoint of 0

lifting is functorial

Universal Cover



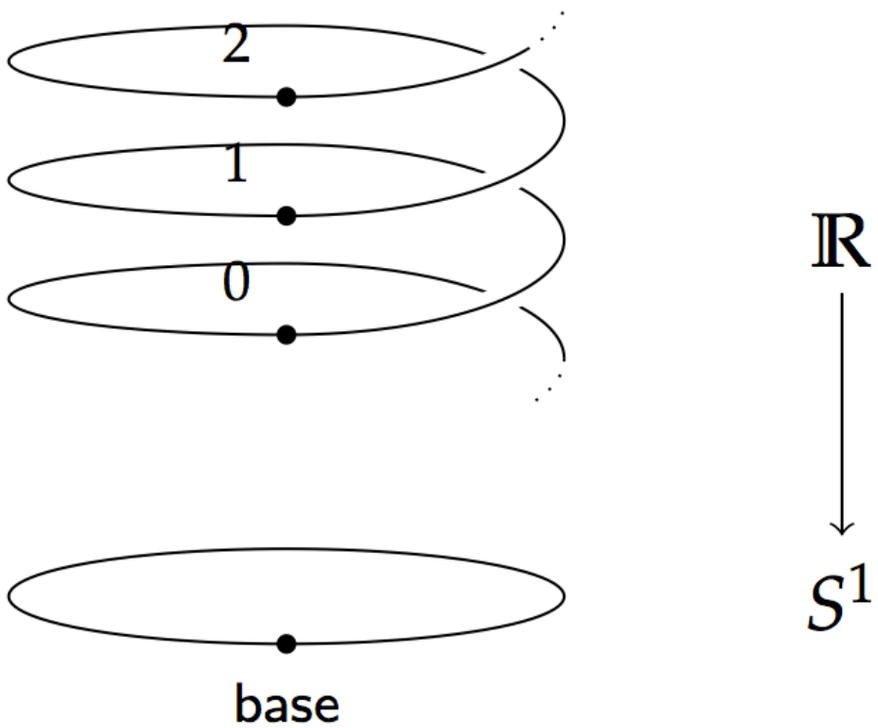
`wind` : $\Omega(S^1) \to \mathbb{Z}$

defined by **lifting** a loop to the cover, and giving the other endpoint of 0

lifting is functorial

lifting `loop` adds 1

Universal Cover



`wind` : $\Omega(S^1) \to \mathbb{Z}$

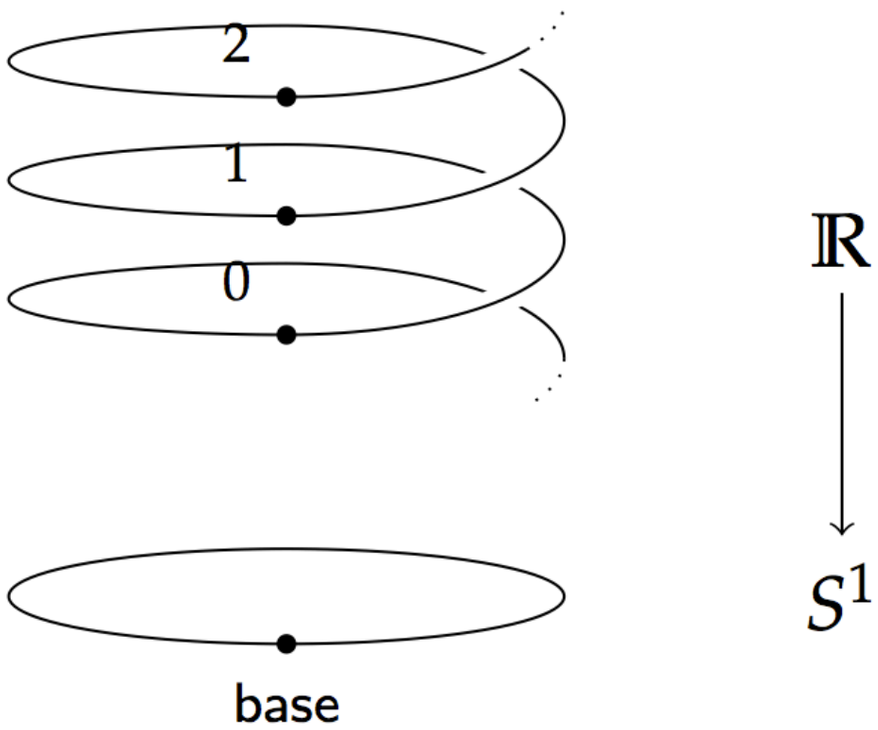
defined by **lifting** a loop to the cover, and giving the other endpoint of 0

lifting is functorial

lifting `loop` adds 1

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

Universal Cover



$\texttt{wind} : \Omega(S^1) \to \mathbb{Z}$

defined by **lifting** a loop to the cover, and giving the other endpoint of 0

lifting is functorial

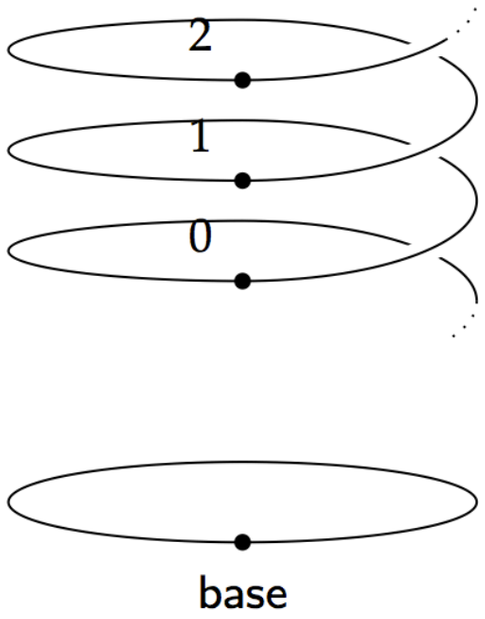
lifting `loop` adds 1

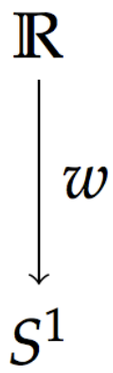
lifting $\texttt{loop}^{-1}$ subtracts 1

Example:

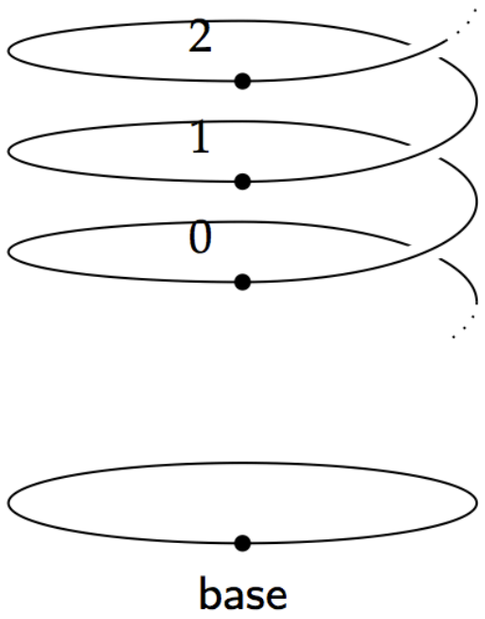
$$\begin{aligned} &\texttt{wind}(\texttt{loop} \circ \texttt{loop}^{-1}) \\ &= 0 + 1 - 1 \\ &= 0 \end{aligned}$$

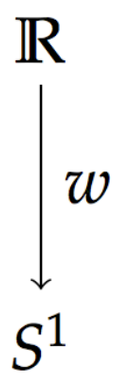
Universal Cover





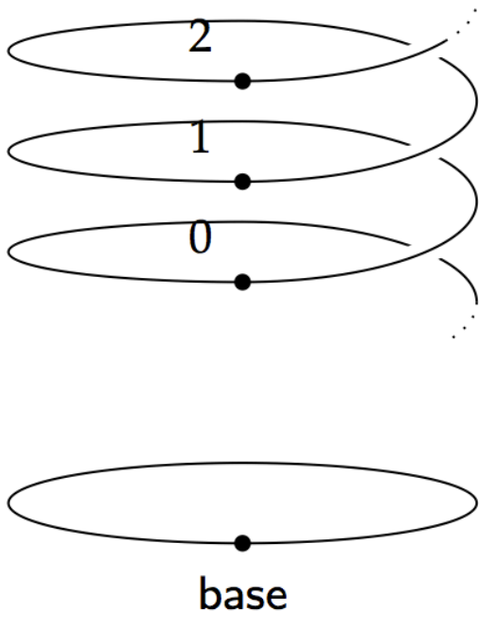
Universal Cover

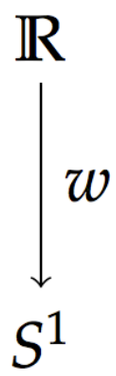





```
Cover : S¹ → Type
Cover(base) := ℤ
Cover₁(loop) :=
ua(successor) : ℤ = ℤ
```

Universal Cover

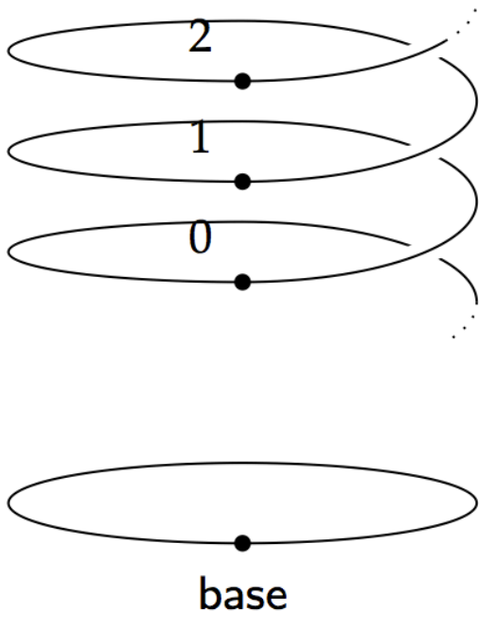




defined by circle recursion

```
Cover : S¹ → Type
Cover(base) := ℤ
Cover₁(loop) :=
ua(successor) : ℤ = ℤ
```

Universal Cover



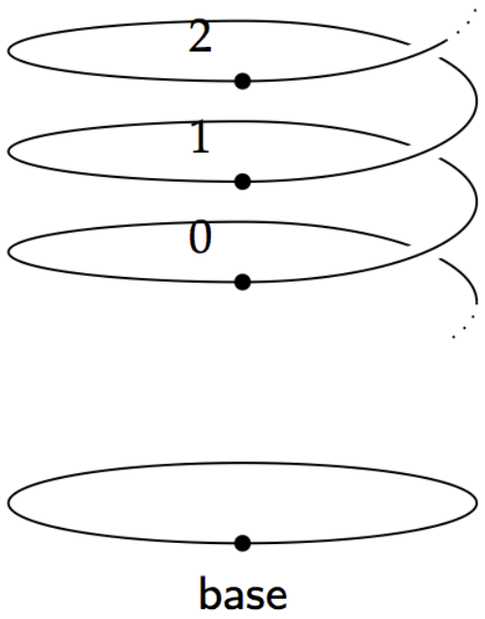
$$\begin{array}{c} \mathbb{R} \\ \downarrow w \\ S^1 \end{array}$$

defined by circle recursion

```
Cover : S¹ → Type
Cover(base) := ℤ
Cover₁(loop) :=
ua(successor) : ℤ = ℤ
```

interpret loop as “add 1” bijection

Universal Cover



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

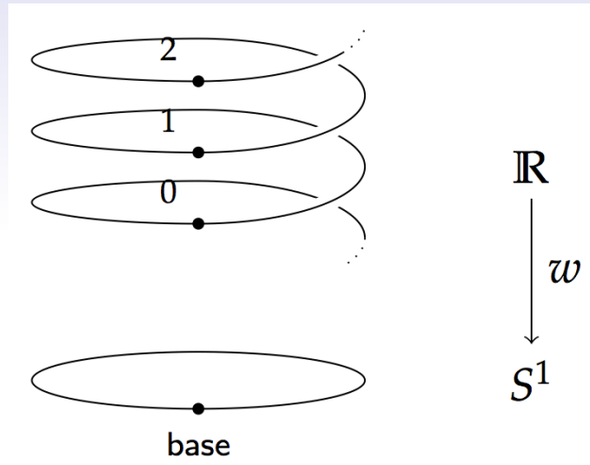
defined by circle recursion

```
Cover : S¹ → Type
Cover(base) := ℤ
Cover₁(loop) :=
  ua(successor) : ℤ = ℤ
```

Voevodsky's univalence axiom

interpret loop as "add 1" bijection

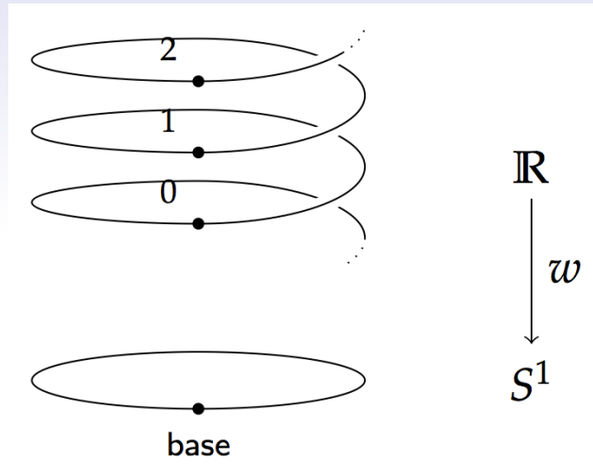
Winding number




```
wind : Ω(S¹) → ℤ
wind(p) = transport_Cover(p,0)
```

lift p to cover, starting at 0

Winding number



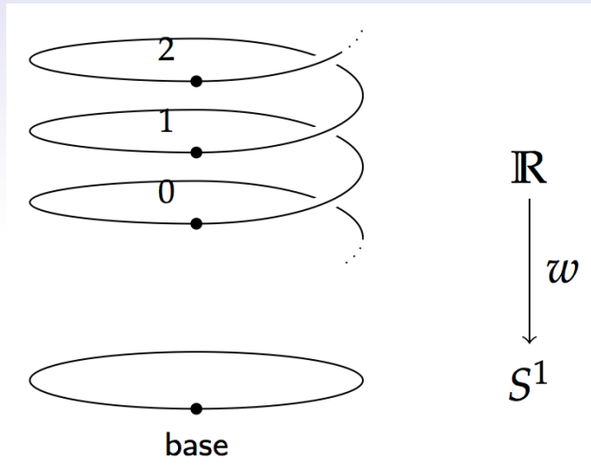
$$\mathtt{wind} : \Omega(S^1) \to \mathbb{Z}$$

$$\mathtt{wind}(p) = \mathtt{transport}_{\mathtt{Cover}}(p, 0)$$

lift p to cover, starting at 0

$$\mathtt{wind}(\mathtt{loop}^{-1} \circ \mathtt{loop})$$

Winding number

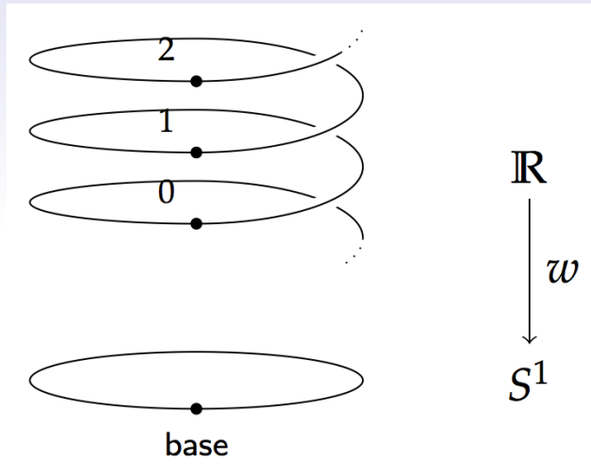


$$\mathtt{wind} : \Omega(S^1) \to \mathbb{Z}$$
$$\mathtt{wind}(p) = \mathtt{transport}_{\mathtt{Cover}}(p, 0)$$

lift p to cover, starting at 0

$$\mathtt{wind}(\mathtt{loop}^{-1} \circ \mathtt{loop})$$
$$= \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop}^{-1} \circ \mathtt{loop}, 0)$$

Winding number

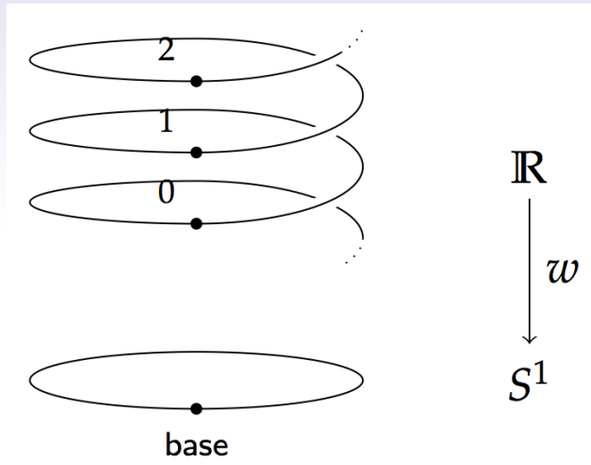


$$\mathtt{wind} : \Omega(S^1) \to \mathbb{Z}$$
$$\mathtt{wind}(p) = \mathtt{transport}_{\mathtt{Cover}}(p, 0)$$

lift p to cover, starting at 0

$$\mathtt{wind}(\mathtt{loop}^{-1} \circ \mathtt{loop})$$
$$= \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop}^{-1} \circ \mathtt{loop}, 0)$$
$$= \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop}^{-1}, \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop}, 0))$$

Winding number

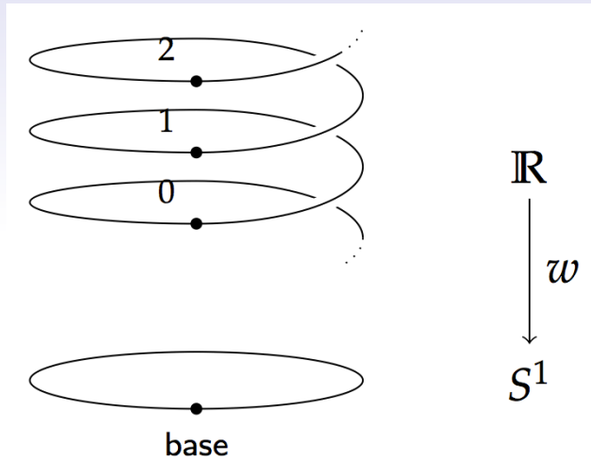



```
wind : Ω(S¹) → ℤ
wind(p) = transport_Cover(p,0)
```

lift p to cover, starting at 0

```
  wind(loop⁻¹ o loop)
= transport_Cover(loop⁻¹ o loop, 0)
= transport_Cover(loop⁻¹, transport_Cover(loop,0))
= transport_Cover(loop⁻¹, 1)
```

Winding number



$$\mathtt{wind} : \Omega(S^1) \to \mathbb{Z}$$
$$\mathtt{wind}(p) = \mathtt{transport}_{\mathtt{Cover}}(p,0)$$

lift p to cover, starting at 0

$$\mathtt{wind}(\mathtt{loop}^{-1} \circ \mathtt{loop})$$
$$= \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop}^{-1} \circ \mathtt{loop}, 0)$$
$$= \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop}^{-1}, \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop},0))$$
$$= \mathtt{transport}_{\mathtt{Cover}}(\mathtt{loop}^{-1}, 1)$$
$$= 0$$

Fundamental group of the circle

The HoTT book

7.2 Some basic homotopy groups 173

7.2.1.1 Encode/decode proof

By definition, $\Omega(S^1)$ is base $=_{S^1}$ base. If we attempt to prove that $\Omega(S^1) = \mathbb{Z}$ by directly constructing an equivalence, we will get stuck, because type theory gives you little leverage for working with loops. Instead, we generalize the theorem statement to the path fibration, and analyze the whole fibration:

$$P(x : S^1) :\equiv (\mathrm{base} =_{S^1} x)$$

with one end-point free.

We show that $P(x)$ is equal to another fibration, which gives a more explicit description of the paths—we call this other fibration "codes", because its elements are data that act as codes for paths on the circle. In this case, the codes fibration is the universal cover of the circle.

Definition 7.2.1 (Universal Cover of S^1). Define $\mathrm{code}(x : S^1) : \mathcal{U}$ by circle-recursion, with

$$\mathrm{code}(\mathrm{base}) :\equiv \mathbb{Z}$$
$$\mathrm{code}(\mathrm{loop}) :\equiv \mathrm{ua}(\mathrm{succ})$$

where succ is the equivalence $\mathbb{Z} \simeq \mathbb{Z}$ given by adding one, which by univalence determines a path from $\mathbb{Z}$ to $\mathbb{Z}$ in $\mathcal{U}$.

To define a function by circle recursion, we need to find a point and a loop in the target. In this case, the target is $\mathcal{U}$, and the point we choose is $\mathbb{Z}$, corresponding to our expectation that the fiber of the universal cover should be the integers. The loop we choose is the successor/predecessor isomorphism on $\mathbb{Z}$, which corresponds to the fact that going around the loop in the base goes up one level on the helix. Univalence is necessary for this part of the proof, because we need a non-trivial equivalence on $\mathbb{Z}$.

From this definition, it is simple to calculate that transporting with code takes loop to the successor function, and loop^{-1} to the predecessor function:

Lemma 7.2.2. $\mathrm{transport}^{\mathrm{code}}(\mathrm{loop}, z) = z + 1$ *and* $\mathrm{transport}^{\mathrm{code}}(\mathrm{loop}^{-1}, z) = z - 1$

Proof. For the first, we calculate as follows:

$$\begin{aligned}
&\mathrm{transport}^{\mathrm{code}}(\mathrm{loop}, z)\\
=\ &\mathrm{transport}^{A\to A}(\langle\mathrm{code}(\mathrm{loop})\rangle_1, z) && \text{associativity}\\
=\ &\mathrm{transport}^{A\to A}(\mathrm{ua}(\mathrm{succ}), z) && \text{reduction for circle-recursion}\\
=\ &z + 1 && \text{reduction for ua}
\end{aligned}$$

The second follows from the first, because $\mathrm{transport}^P p$ and and $\mathrm{transport}^P p^{-1}$ are always inverses, so $\mathrm{transport}^{\mathrm{code}}\mathrm{loop}^{-1}$ must be the inverse of the $- + 1$. □

In the remainder of the proof, we will show that P and code are equivalent.

7.2.1.1.1 Encoding. Next, we define a function encode that maps paths to codes:

Definition 7.2.3. Define $\mathrm{encode} : \prod_{(x:S^1)} P(x) \to \mathrm{code}(x)$ by

$$\mathrm{encode}\ p :\equiv \mathrm{transport}^{\mathrm{code}}(p, 0)$$

(we leave the argument x implicit).

Encode is defined by lifting a path into the universal cover, which determines an equivalence, and then applying the resulting equivalence to 0. The interesting thing about this function is that it computes a concrete number from a loop on the circle, when this loop is represented using the abstract groupoidal framework of HoTT. To gain an intuition for how it does this, observe that by the above lemmas, $\mathrm{transport}^{\mathrm{code}}(\mathrm{loop}, x)$ is $- + 1$ and $\mathrm{transport}^{\mathrm{code}}\mathrm{loop}^{-1}x$ is $x - 1$. Further, transport is functorial (chapter 2), so $\mathrm{transport}^{\mathrm{code}}\mathrm{loop} \cdot \mathrm{loop}$ is $(\mathrm{transport}^{\mathrm{code}}\mathrm{loop}) \circ (\mathrm{transport}^{\mathrm{code}}(\mathrm{loop},))$, etc. Thus, when p is a composition like

$$\mathrm{loop} \cdot \mathrm{loop}^{-1} \cdot \mathrm{loop} \cdot \ldots$$

$\mathrm{transport}^{\mathrm{code}}p$ will compute a composition of functions like

$$\langle - + 1\rangle \circ \langle - - 1\rangle \circ \langle - + 1\rangle \circ \ldots$$

Applying this composition of functions to 0 will compute the *winding number* of the path—how many times it goes around the circle, with orientation marked by whether it is positive or negative, after inverses have been canceled. Thus, the computational behavior of encode follows from the reduction rules for higher-inductive types and univalence, and the action of transport on compositions and inverses.

Note that the instance $\mathrm{encode}' :\equiv \mathrm{encode}_{\mathrm{base}}$ has type base = base $\to \mathbb{Z}$, which will be one half of the equivalence between base = base and $\mathbb{Z}$.

7.2.1.1.2 Decoding. Decoding an integer as a path is defined by recursion:

Definition 7.2.4. Define $\mathrm{loop}^z : \mathbb{Z} \to$ base = base by

$$\mathrm{loop}^n = \begin{cases} \mathrm{loop} \cdot \mathrm{loop} \cdot \ldots \cdot \mathrm{loop}\ (n \text{ times}) & \text{for positive } n\\ \mathrm{loop}^{-1} \cdot \mathrm{loop}^{-1} \cdot \ldots \cdot \mathrm{loop}^{-1}\ (n \text{ times}) & \text{for negative } n\\ \mathrm{refl} & \text{for } 0 \end{cases}$$

Since what we want overall is an equivalence between base = base and $\mathbb{Z}$, we might expect to be able to prove that encode' and loop^- give an equivalence. The problem comes in trying to prove the "decode after encode" direction, where we would need to show that $\mathrm{loop}^{\mathrm{encode}\,p} = p$ for all p. We would like to apply path induction, but path induction

does not apply to loops like a with both endpoints fixed! The way to solve this problem is to generalize the theorem to show that $\mathrm{loop}^{\mathrm{encode}_x p} = p$ for all $x : S^1$ and $p : \mathrm{base} = x$. However, this does not make sense as is, because loop^- is defined only for base = base, whereas here it is applied to a base = x. Thus, we generalize loop as follows:

Definition 7.2.5. Define $\mathrm{decode} : \prod_{(x:S^1)} \prod_{(\mathrm{code}(x) \to P(x))}$, by circle induction on x. It suffices to give a function $\mathrm{code}(\mathrm{base}) \to P(\mathrm{base})$, for which we use loop^-, and to show that loop^- respects the loop.

Proof. To show that loop^- respects the loop, it suffices to give a path from loop^- to itself that lies over loop. Formally, this means a path from $\mathrm{transport}^{(x'\to \mathrm{code}(x')\to P(x'))}(\mathrm{loop}, \mathrm{loop}^-)$ to loop^-. We define such a path as follows:

$$\begin{aligned}
&\mathrm{transport}^{(x'\to \mathrm{code}(x')\to P(x'))}(\mathrm{loop}, \mathrm{loop}^-)\\
=\ &\mathrm{transport}^{\mathrm{code}\to P}\mathrm{loop} \circ \mathrm{loop}^- \circ \mathrm{transport}^{\mathrm{code}}\mathrm{loop}^{-1}\\
=\ &(- \cdot \mathrm{loop}) \circ (\mathrm{loop}^-) \circ \mathrm{transport}^{\mathrm{code}}\mathrm{loop}^{-1}\\
=\ &(- \cdot \mathrm{loop}) \circ (\mathrm{loop}^-) \circ (- - 1)\\
=\ &(n \mapsto \mathrm{loop}^{n-1} \cdot \mathrm{loop})
\end{aligned}$$

From line 1 to line 2, we apply the definition of transport when the outer connective of the fibration is $\to$, which reduces the transport to pre- and post-composition with transport at the domain and range types. From line 2 to line 3, we apply the definition of transport when the type family is base = x, which is post-composition of paths. From line 3 to line 4, we use the action of code on loop^{-1} defined in Lemma 7.2.2. From line 4 to line 5, we simply reduce the function composition. Thus, it suffices to show that for all n, $\mathrm{loop}^{n-1} \cdot \mathrm{loop} = \mathrm{loop}^n$, which is an easy induction, using the groupoid laws. □

7.2.1.1.3 Decoding after encoding

Lemma 7.2.6. *For all for all* $x : S^1$ *and* $p : \mathrm{base} = x$, $\mathrm{decode}_x(\mathrm{encode}_x(p)) = p$.

Proof. By path induction, it suffices to show that $\mathrm{decode}_{\mathrm{base}}(\mathrm{encode}_{\mathrm{base}}(\mathrm{refl}_{\mathrm{base}})) = \mathrm{refl}_{\mathrm{base}}$. But $\mathrm{encode}_{\mathrm{base}}(\mathrm{refl}_{\mathrm{base}}) \equiv \mathrm{transport}^{\mathrm{code}}(\mathrm{refl}_{\mathrm{base}}, 0) \equiv 0$, and $\mathrm{decode}_{\mathrm{base}}(0) \equiv \mathrm{loop}^0 \equiv \mathrm{refl}_{\mathrm{base}}$. □

7.2.1.1.4 Encoding after decoding

Lemma 7.2.7. *For all for all* $x : S^1$ *and* $c : \mathrm{code}(x)$, $\mathrm{encode}_x(\mathrm{decode}_x(c)) = c$.

Proof. The proof is by circle induction. It suffices to show the case for base, because the case for loop is a path between paths in $\mathbb{Z}$, which can be given by appealing to the fact that $\mathbb{Z}$ is a set.

Thus, it suffices to show, for all $n : \mathbb{Z}$, that

$$\mathrm{encode}'(\mathrm{loop}^n) = n$$

The proof is by induction, with cases for $0, 1, -1, n+1$, and $n-1$.

- In the case for 0, the result is true by definition.
- In the case for 1, $\mathrm{encode}'(\mathrm{loop}^1)$ reduces to $\mathrm{transport}^{\mathrm{code}}(\mathrm{loop}, 0)$, which by Lemma 7.2.2 is $0 + 1 = 1$.
- In the case for $n + 1$,

$$\begin{aligned}
&\mathrm{encode}'(\mathrm{loop}^{n+1})\\
=\ &\mathrm{encode}'(\mathrm{loop}^n \cdot \mathrm{loop})\\
=\ &\mathrm{transport}^{\mathrm{code}}(\langle\mathrm{loop}^n \cdot \mathrm{loop}\rangle, 0)\\
=\ &\mathrm{transport}^{\mathrm{code}}(\mathrm{loop}, \mathrm{transport}^{\mathrm{code}}(\langle\mathrm{loop}^n\rangle, 0))) && \text{by functoriality}\\
=\ &(\mathrm{transport}^{\mathrm{code}}(\langle\mathrm{loop}^n\rangle, 0)] + 1 && \text{by Lemma 7.2.2}\\
=\ &n + 1 && \text{by the IH}
\end{aligned}$$

- The cases for negatives are analogous.

□

7.2.1.1.5 Tying it all together

Theorem 7.2.8. *There is a family of equivalences* $\prod_{(x:S^1)} \prod_{(P(x) \simeq \mathrm{code}(x))}$.

Proof. The maps encode and decode are mutually inverse by Lemmas 7.2.6 and 7.2.6, and this can be improved to an equivalence. □

Instantiating at base gives

Corollary 7.2.9. $(\mathrm{base} = \mathrm{base}) \simeq \mathbb{Z}$

A simple induction shows that this equivalence takes addition to composition, so $\Omega(S^1) = \mathbb{Z}$ as groups.

Corollary 7.2.10. $\pi_k(S^1) = \mathbb{Z}$ *if* $k = 1$ *and* 1 *otherwise.*

Proof. For $k = 1$, we sketched the proof from Corollary 7.2.9 above. For $k > 1$, $\|\Omega^{k+1}(S^1)\|_0 = \|\Omega^k(\Omega S^1)\|_0 = \|\Omega^k(Z)\|_0$, which is 1 because $\mathbb{Z}$ is a set and π_n of a set is trivial (FIXME lemmas to cite?). □

Computer-checked

```
Cover : S¹ → Type
Cover x = S¹-rec Int (ua succEquiv) x

transport-Cover-loop : Path (transport Cover loop) succ
transport-Cover-loop =
  transport Cover loop
    =⟨ transport-ap-assoc Cover loop ⟩
  transport (λ x → x) (ap Cover loop)
    =⟨ ap   (transport (λ x → x))
            (βloop/rec Int (ua succEquiv)) ⟩
  transport (λ x → x) (ua succEquiv)
    =⟨ type-β _ ⟩
  succ ∎

transport-Cover-!loop : Path (transport Cover (! loop)) pred
transport-Cover-!loop =
  transport Cover (! loop)
    =⟨ transport-ap-assoc Cover (! loop) ⟩
  transport (λ x → x) (ap Cover (! loop))
    =⟨ ap (transport (λ x → x)) (ap-! Cover loop)⟩
  transport (λ x → x) (! (ap Cover loop))
    =⟨ ap  (λ y → transport (λ x → x) (! y))
           (βloop/rec Int (ua succEquiv)) ⟩
  transport (λ x → x) (! (ua succEquiv))
    =⟨ ap (transport (λ x → x)) (!-ua succEquiv) ⟩
  transport (λ x → x) (ua (!equiv succEquiv))
    =⟨ type-β _ ⟩
  pred ∎

encode : {x : S¹} →  Path base x  →  Cover x
encode a = transport Cover a Zero

encode' : Path base base → Int
encode' a = encode {base} a

loop^ : Int → Path base base
loop^ Zero     = id
loop^ (Pos One)   = loop
loop^ (Pos (S n)) = loop · loop^ (Pos n)
loop^ (Neg One)   = ! loop
loop^ (Neg (S n)) = ! loop · loop^ (Neg n)

loop^-preserves-pred
  : (n : Int) → Path (loop^ (pred n)) (! loop · loop^ n)
loop^-preserves-pred (Pos One) = ! (!-inv-l loop)
loop^-preserves-pred (Pos (S y)) =
    (--assoc (! loop) loop (loop^ (Pos y)))
  ∙ (ap (λ x → x · loop^ (Pos y)) (!-inv-l loop))
  ∙ (-unit-l (loop^ (Pos y)))
loop^-preserves-pred Zero = id
loop^-preserves-pred (Neg One) = id
loop^-preserves-pred (Neg (S y)) = id

decode : (x : S¹) → Cover x → Path base x
decode {x} =
 S¹-induction
  (λ x' → Cover x' → Path base x')
  loop^
  loop^-respects-loop
  x where
   abstract -- prevent Agda from normalizing
    loop^-respects-loop : transport (λ x' → Cover x' → Path base x') loop loop^ ≡ (λ n → loop^ n)
    loop^-respects-loop =
      (transport (λ x' → Cover x' → Path base x') loop loop^
        =⟨ transport⁼-Cover (Path base) loop loop^ ⟩
       transport (λ x' → Path base x') loop
        ∘ loop^
        ∘ transport Cover (! loop)
        =⟨ !ₕ⁼ (λ y → transport-Path-right loop (loop^ (transport Cover (! loop) y))) ⟩
       (λ p → loop · p)
        ∘ loop^
        ∘ transport Cover (! loop)
        =⟨ !ₕ⁼ (λ y → ap (λ x' → loop · loop^ x') (ap= transport-Cover-!loop)) ⟩
       (λ p → loop · p)
        ∘ loop^
        ∘ pred
        =⟨ id ⟩
       (λ n → loop · (loop^ (pred n)))
        =⟨ !ₕ⁼ (λ y → move-left-! _ loop (loop^ y) (loop^-preserves-pred y)) ⟩
       (λ n → loop^ n)
       ∎)
```

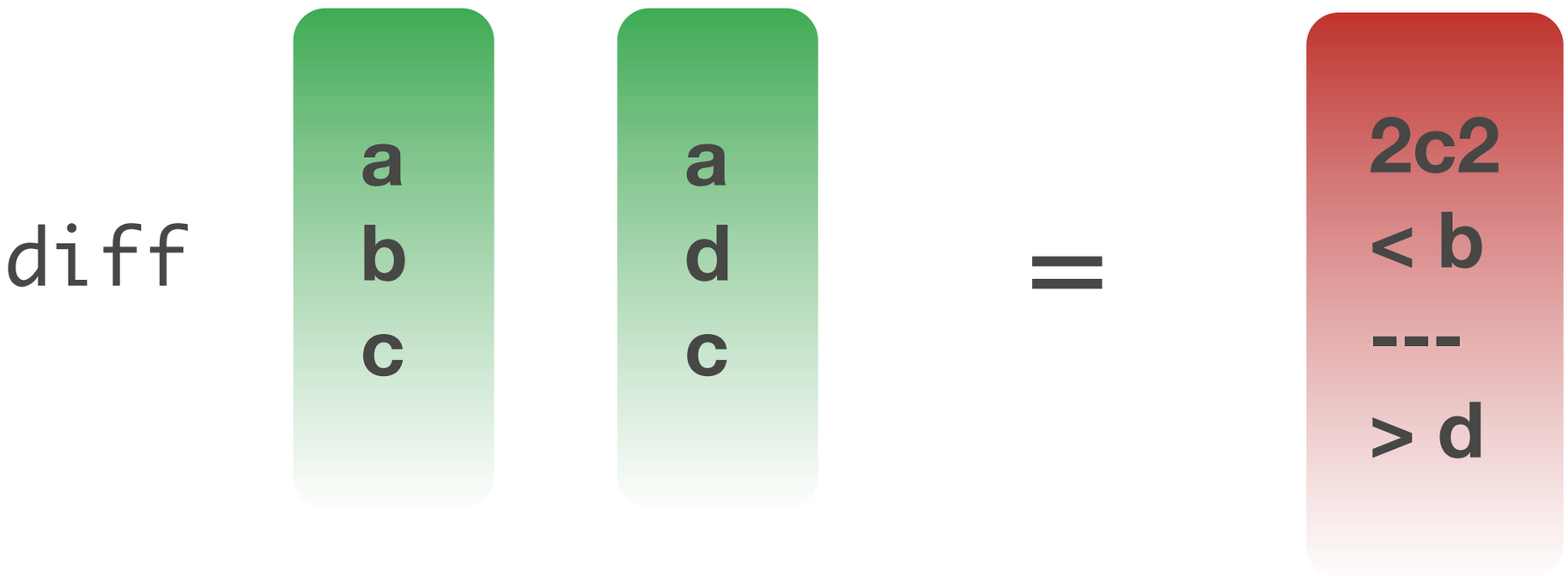
```
abstract -- prevent Agda from normalizing
  encode-loop^ : (n : Int) → Path (encode (loop^ n)) n
  encode-loop^ Zero = id
  encode-loop^ (Pos One) = ap= transport-Cover-loop
  encode-loop^ (Pos (S n)) =
    encode (loop^ (Pos (S n)))
      =⟨ id ⟩
    transport Cover (loop · loop^ (Pos n)) Zero
      =⟨ ap= (transport⁼-Cover (loop^ (Pos n))) ⟩
    transport  Cover loop
               (transport Cover (loop^ (Pos n)) Zero)
      =⟨ ap= transport-Cover-loop ⟩
    succ (transport Cover (loop^ (Pos n)) Zero)
      =⟨ id ⟩
    succ (encode (loop^ (Pos n)))
      =⟨ ap succ (encode-loop^ (Pos n)) ⟩
    succ (Pos n) ∎
  encode-loop^ (Neg One) = ap= transport-Cover-!loop
  encode-loop^ (Neg (S n)) =
    transport Cover (! loop · loop^ (Neg n)) Zero
      =⟨ ap= (transport⁼-Cover (! loop) (loop^ (Neg n))) ⟩
    transport Cover (! loop) (transport Cover (loop^ (Neg n)) Zero)
      =⟨ ap= transport-Cover-!loop ⟩
    pred (transport Cover (loop^ (Neg n)) Zero)
      =⟨ ap pred (encode-loop^ (Neg n)) ⟩
    pred (Neg n) ∎

encode-decode  : {x : S¹} → (c : Cover x)
               → Path (encode (decode{x} c)) c
encode-decode {x} = S¹-induction
   (λ (x : S¹) →  (c : Cover x)
               → Path (encode{x} (decode{x} c)) c)
   encode-loop^ (λ= (λ x' → fst (use-level (use-level (use-level HSet-Int _ _) _ _)) x))

decode-encode  : {x : S¹} (a : Path base x)
               → Path (decode (encode a)) a
decode-encode {x} a =
 path-induction
  (λ  (x' : S¹) (a' : Path base x')
      → Path (decode (encode a')) a')
  id a

Ω[S¹]-Equiv-Int : Equiv (Path base base) Int
Ω[S¹]-Equiv-Int =
  improve (hequiv encode decode decode-encode encode-loop^)

Ω[S¹]-is-Int : (Path base base) = Int
Ω[S¹]-is-Int = ua Ω[S¹]-Equiv-Int

π₁[S¹]-is-Int : π 1 One S¹ base = Int
π₁[S¹]-is-Int = UnTrunc.path _ _ HSet-Int · ap (Trunc (tl 0)) Ω[S¹]-is-Int
```

Outline

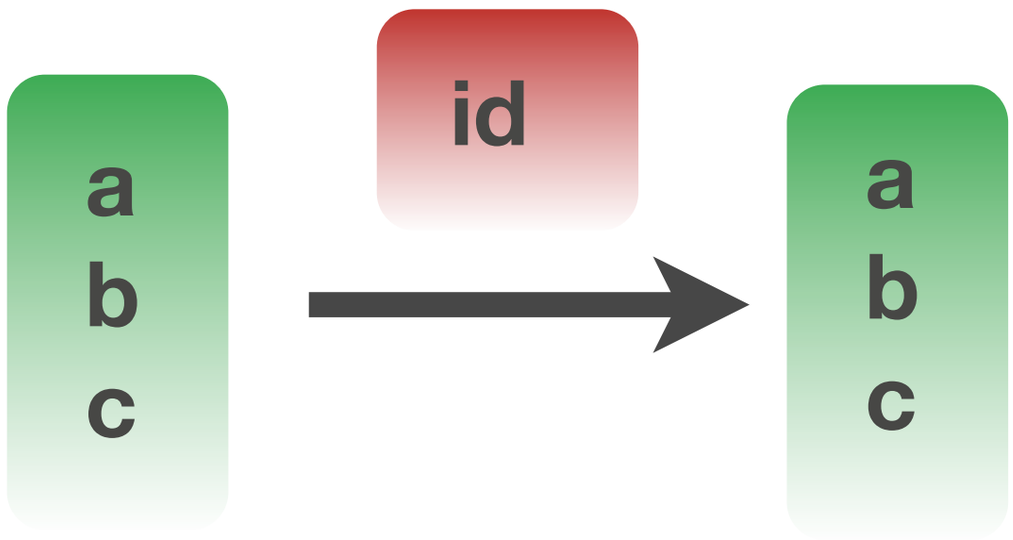
1. Computer-checked homotopy theory

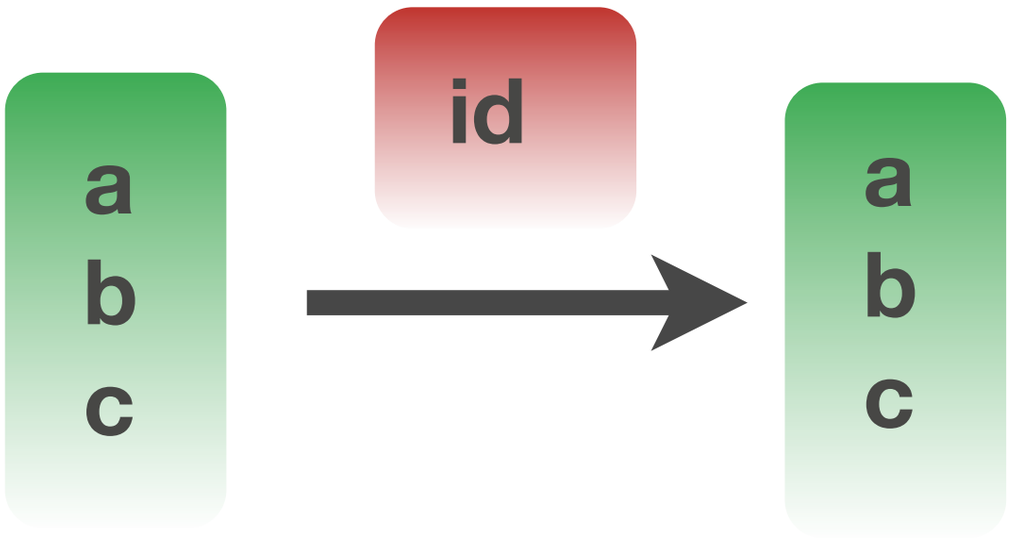
2. Computer-checked software

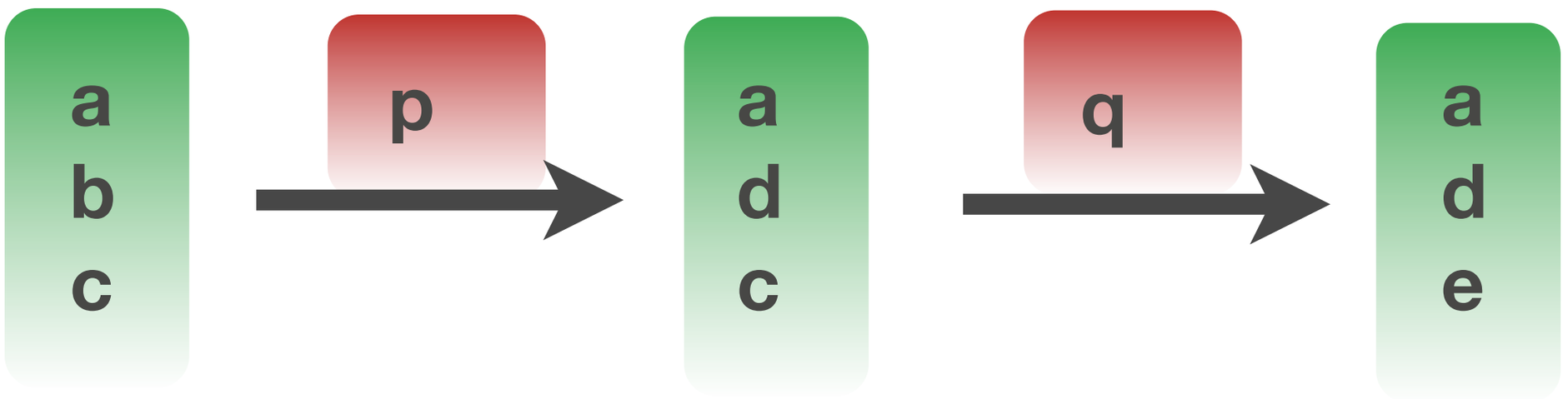
Patches

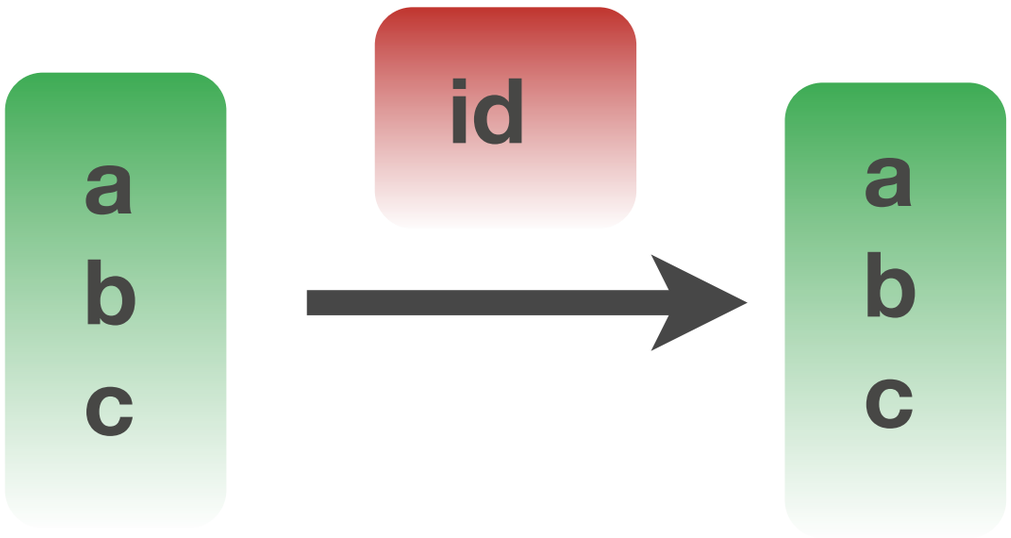


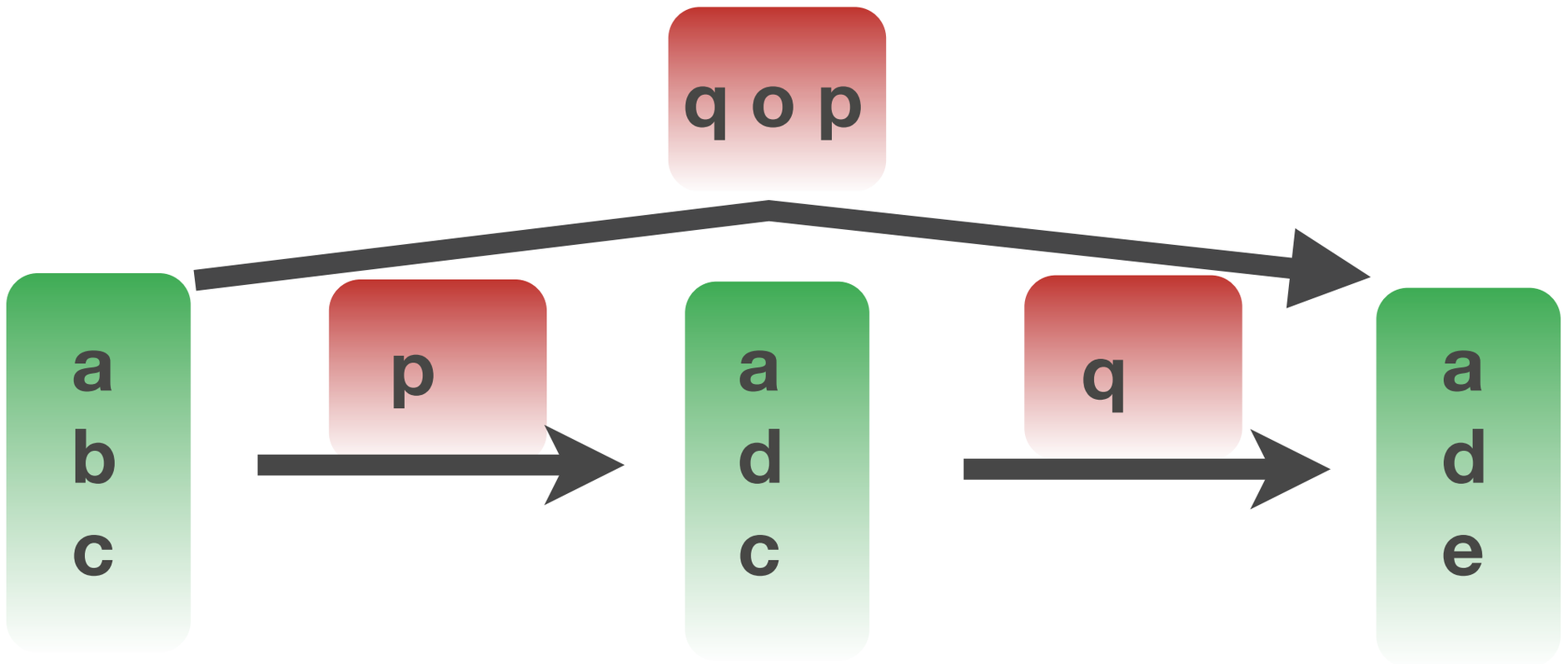
- Version control
- Collaborative editing

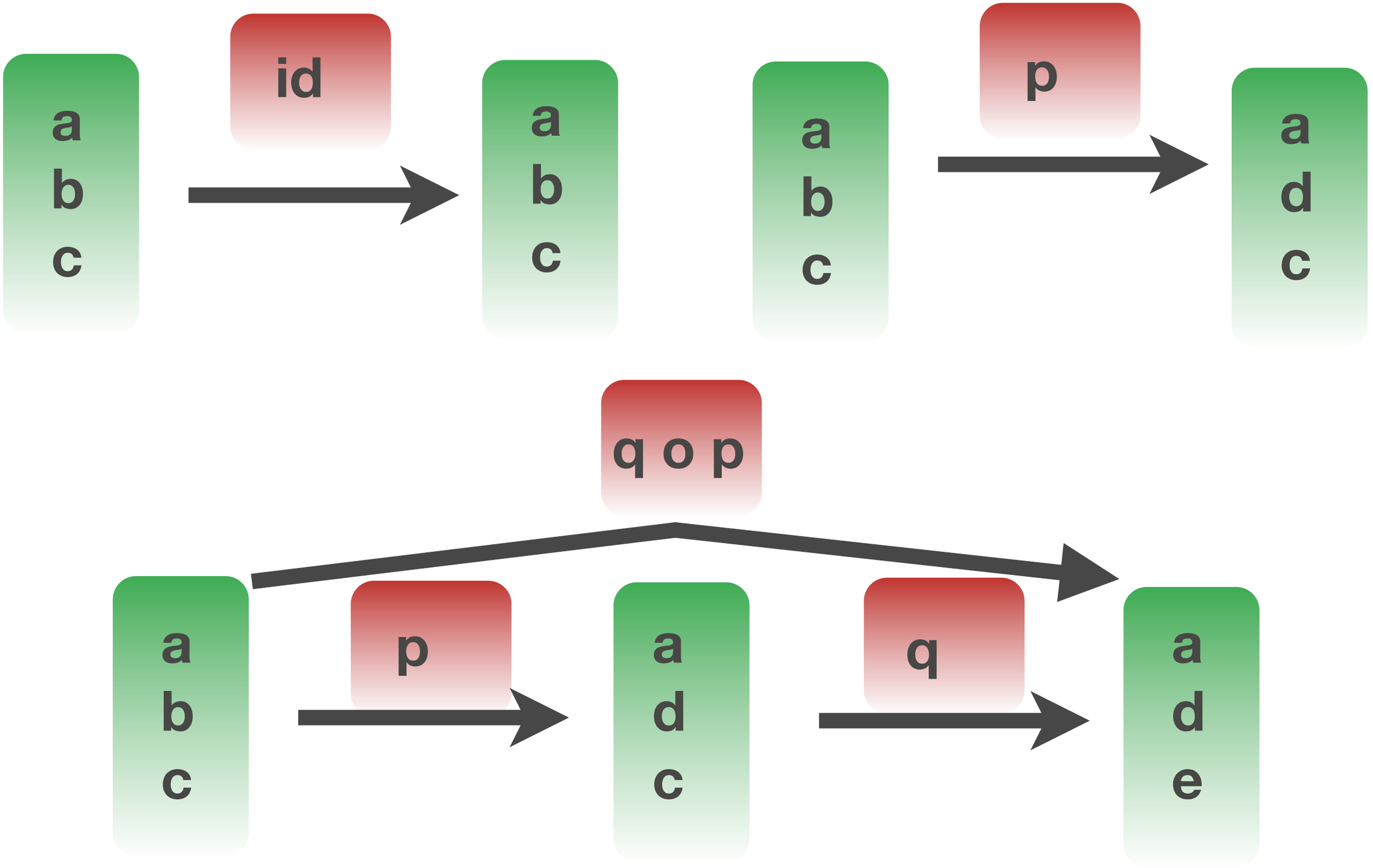

a
b
c
id
a
b
c

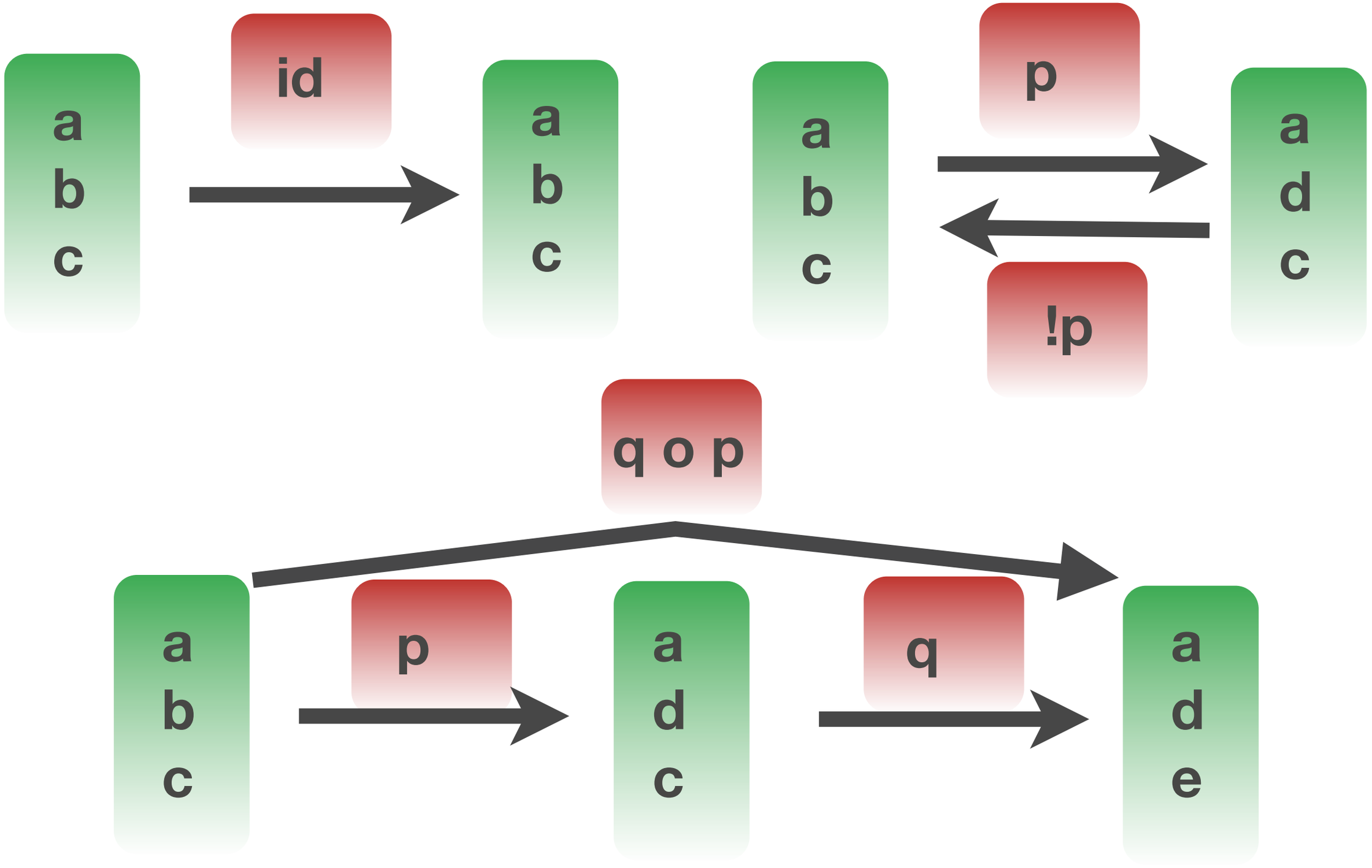

a
b
c
id
a
b
c

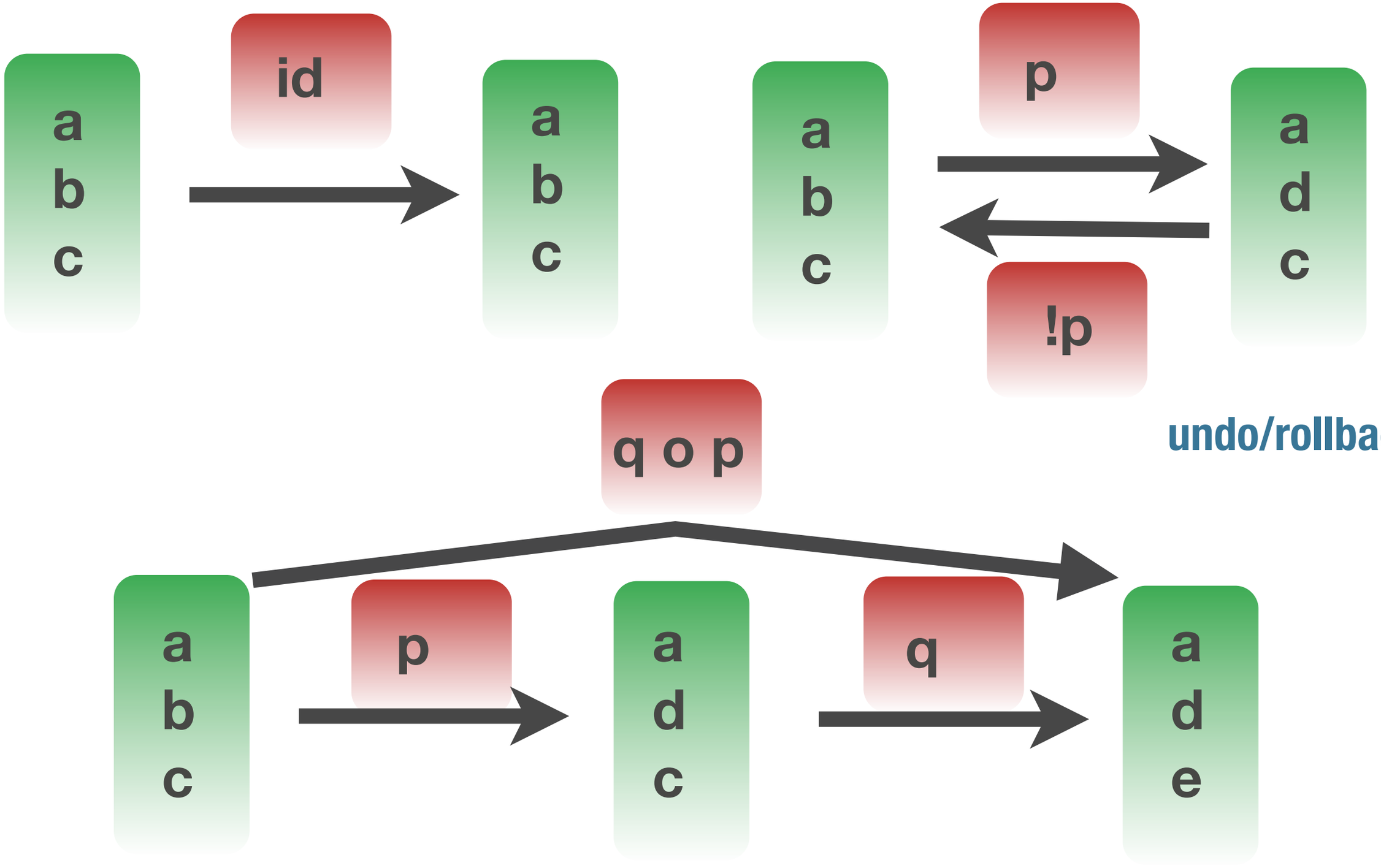

a
b
c
p
a
d
c
q
a
d
e


a
b
c
id
a
b
c

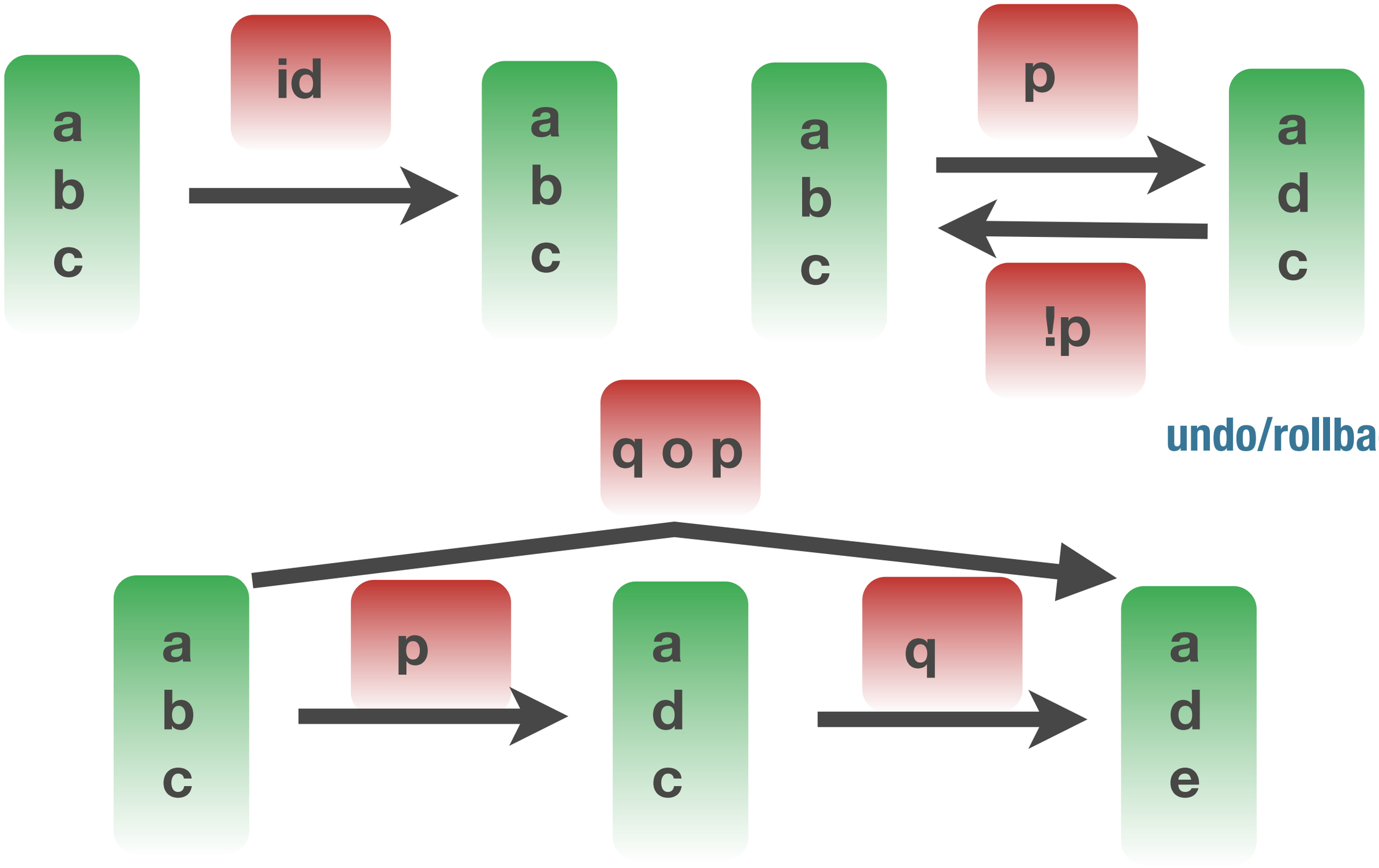

q o p
a
b
c
p
a
d
c
q
a
d
e


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

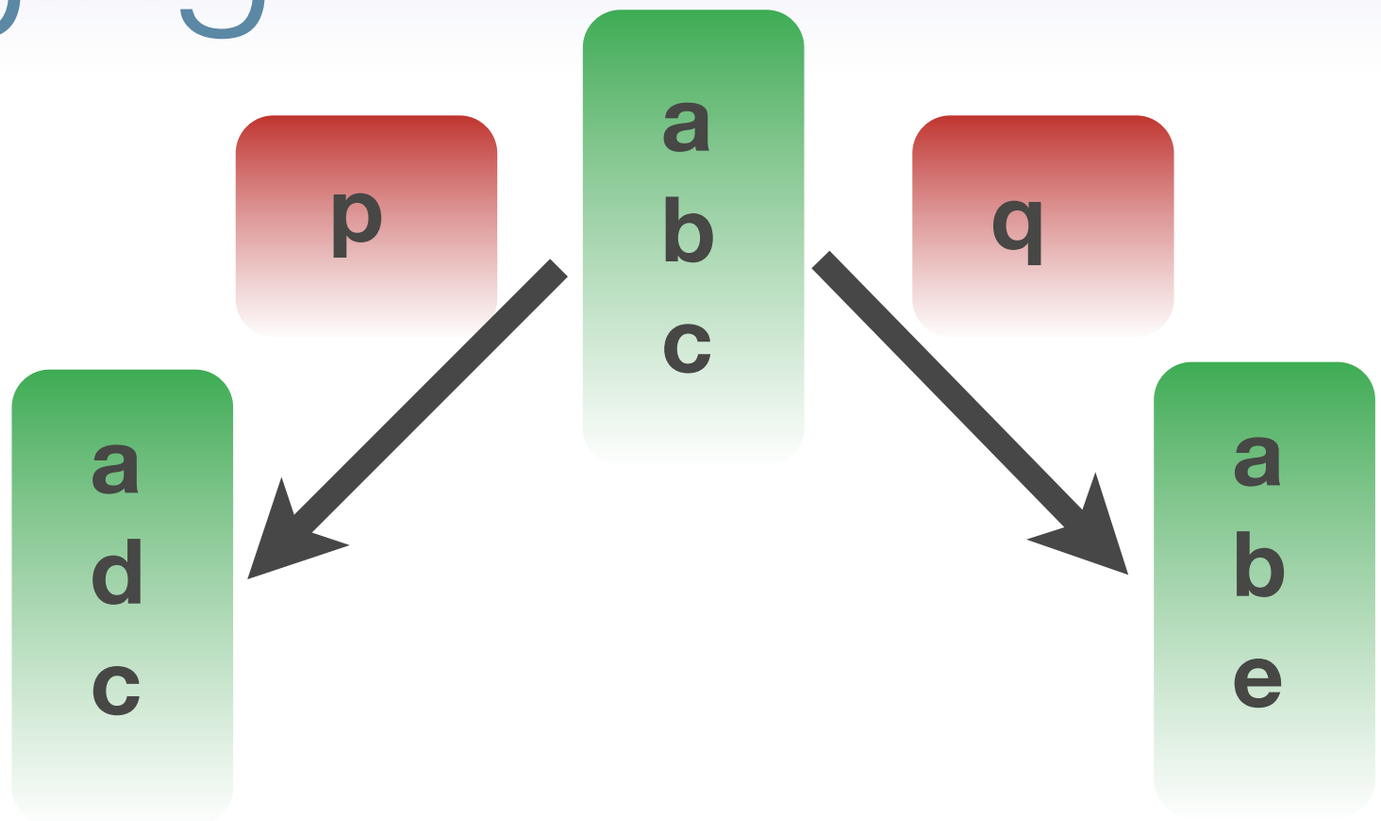

a
b
c
id
a
b
c
a
b
c
p
!p
a
d
c
q o p
a
b
c
p
a
d
c
q
a
d
e


a
b
c
id
a
b
c
a
b
c
p
!p
a
d
c
undo/rollback
q o p
a
b
c
p
a
d
c
q
a
d
e

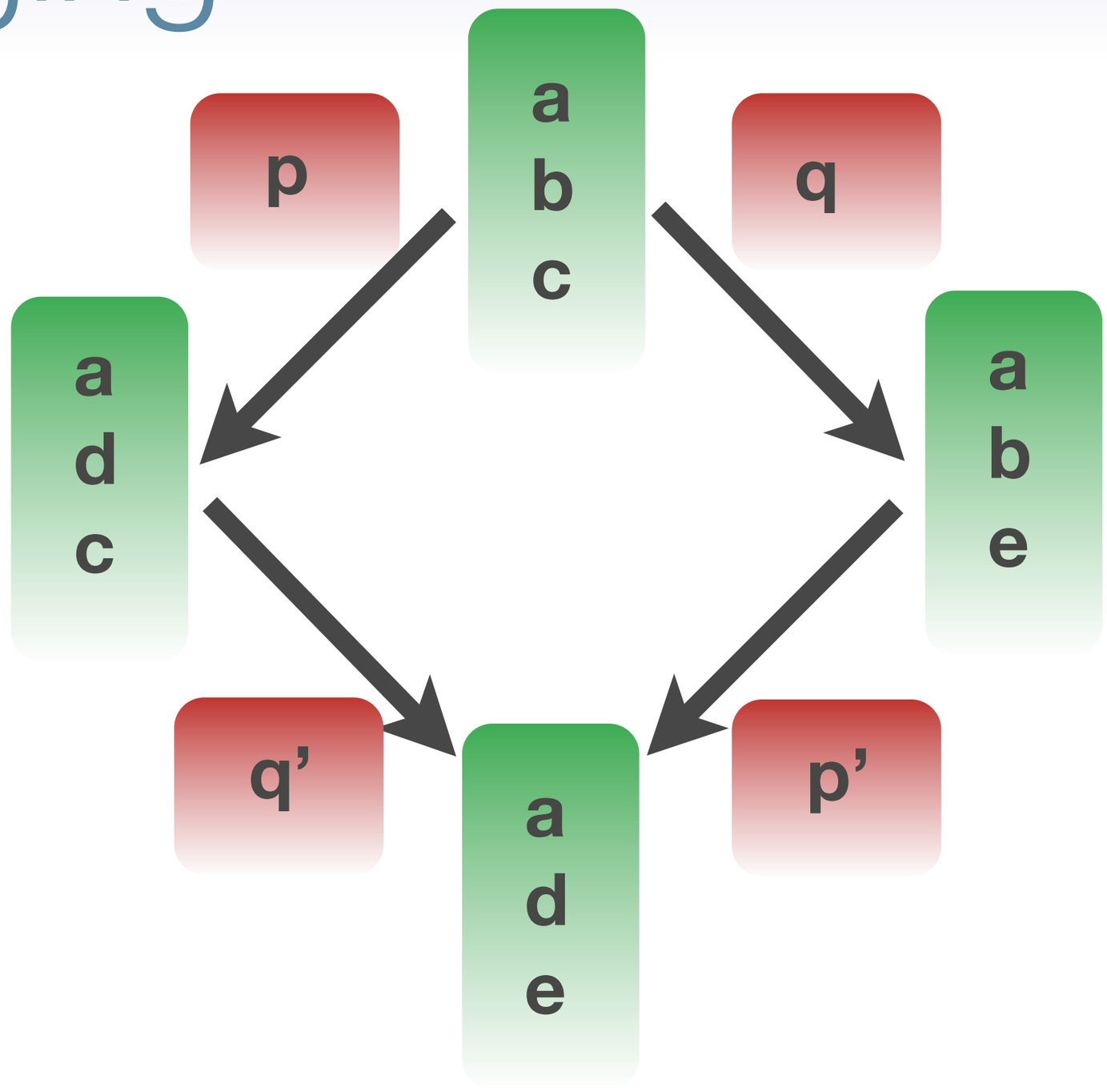
Patches are paths



Merging



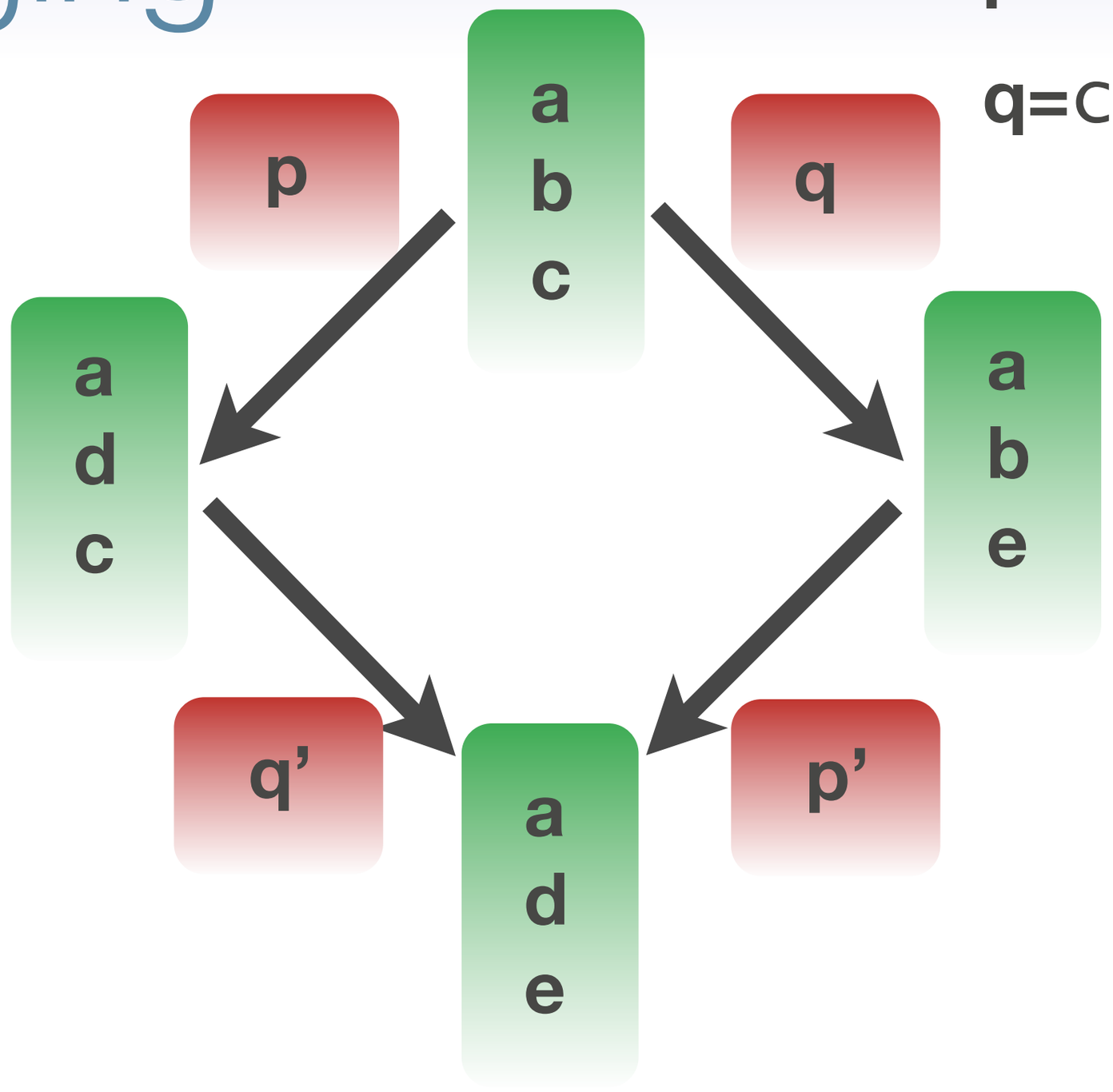
Merging



Merging

p=b↔d at 1

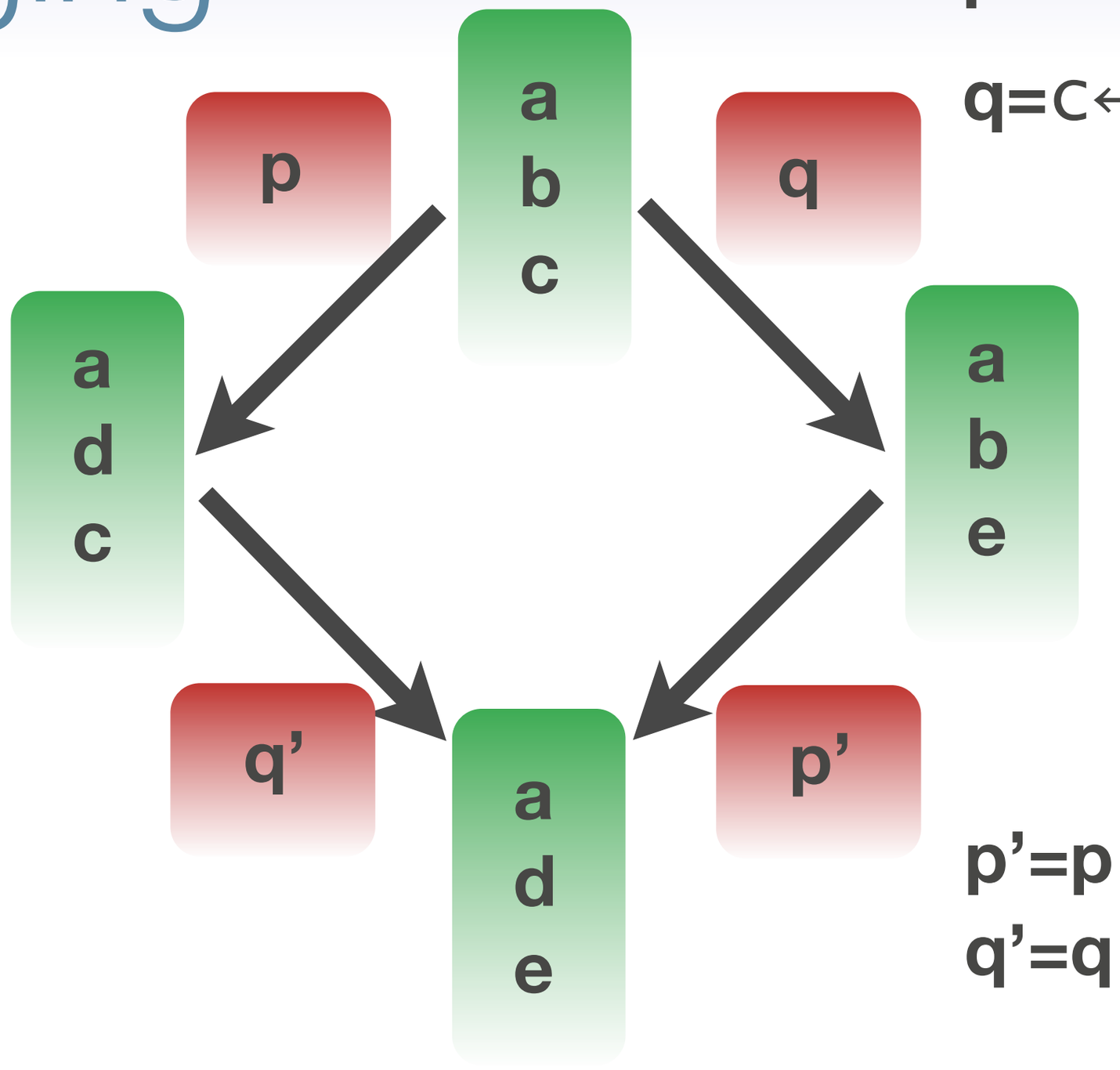
q=c↔e at 2



Merging

p=b↔d at 1

q=c↔e at 2



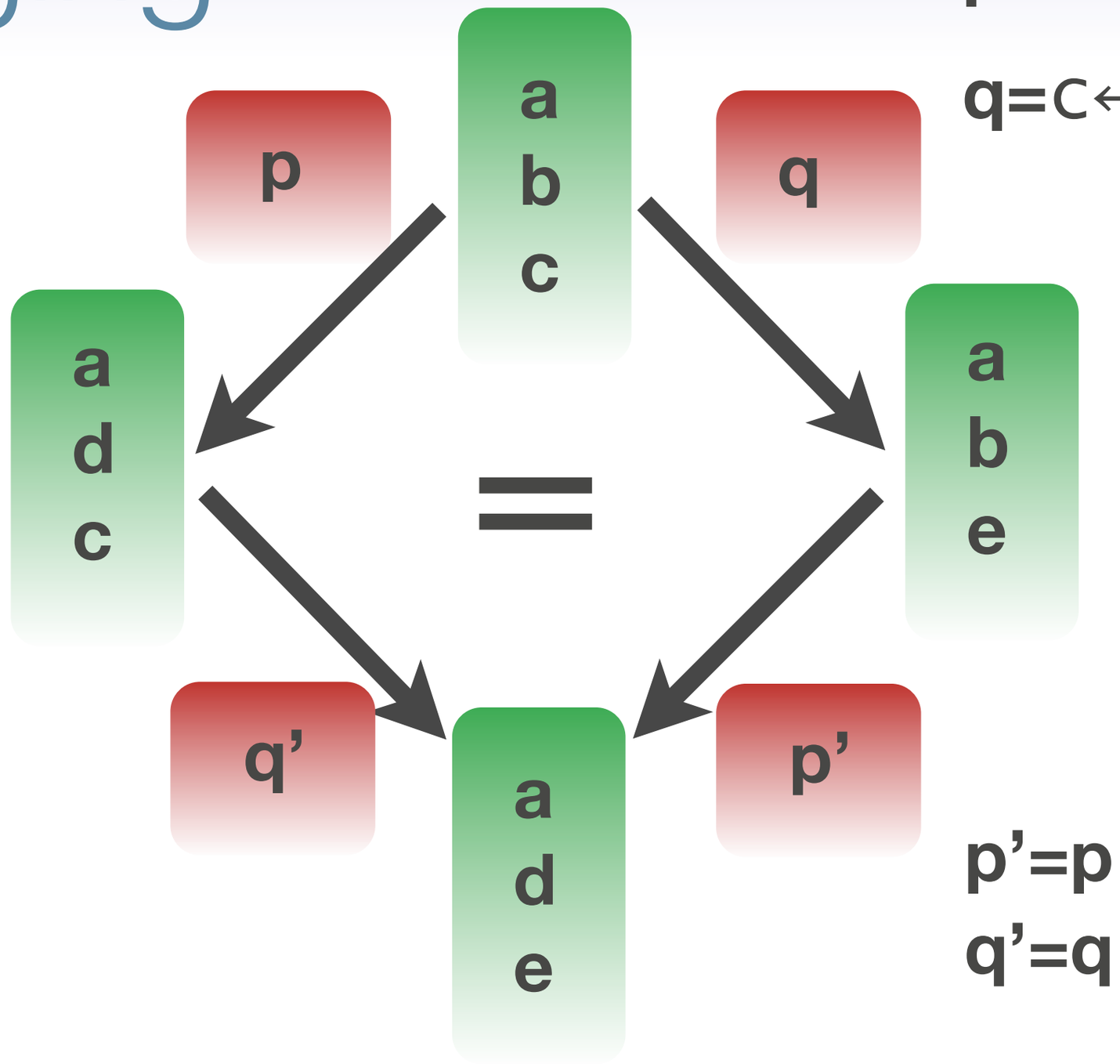
p’=p

q’=q

Merging

p=b↔d at 1

q=c↔e at 2

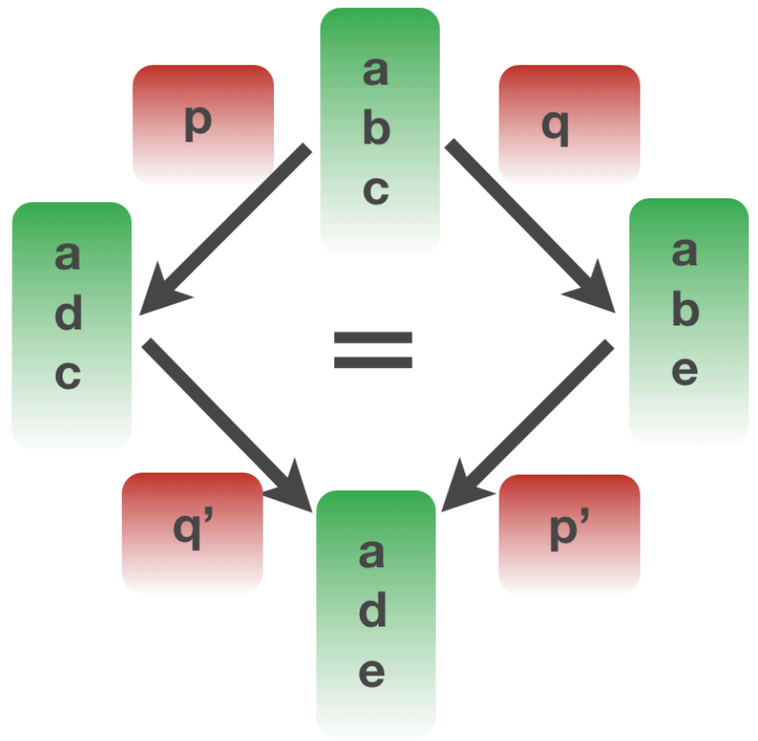


p’=p

q’=q

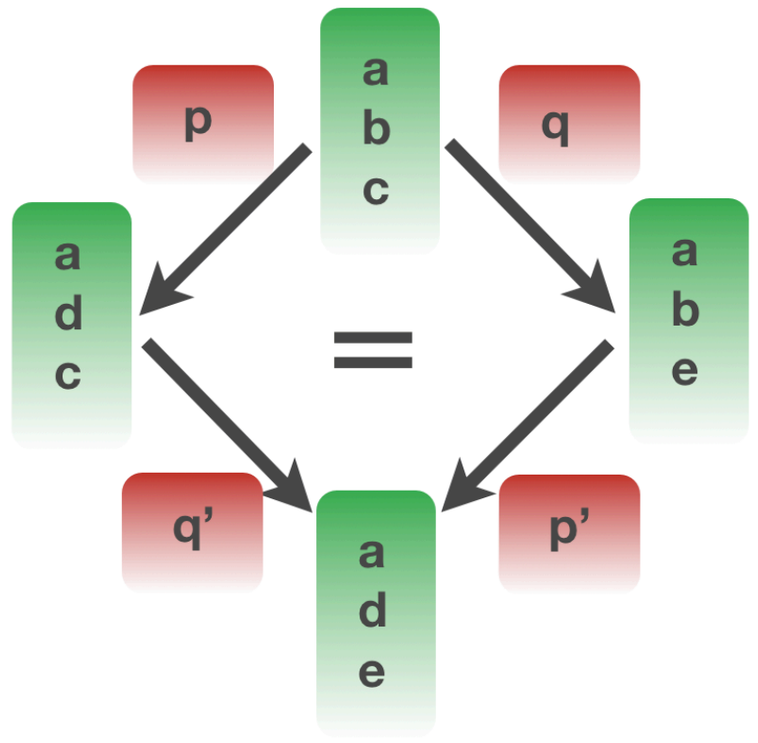
Merging

```
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)
```



Equational theory of patches
= paths between paths

Basic Patches

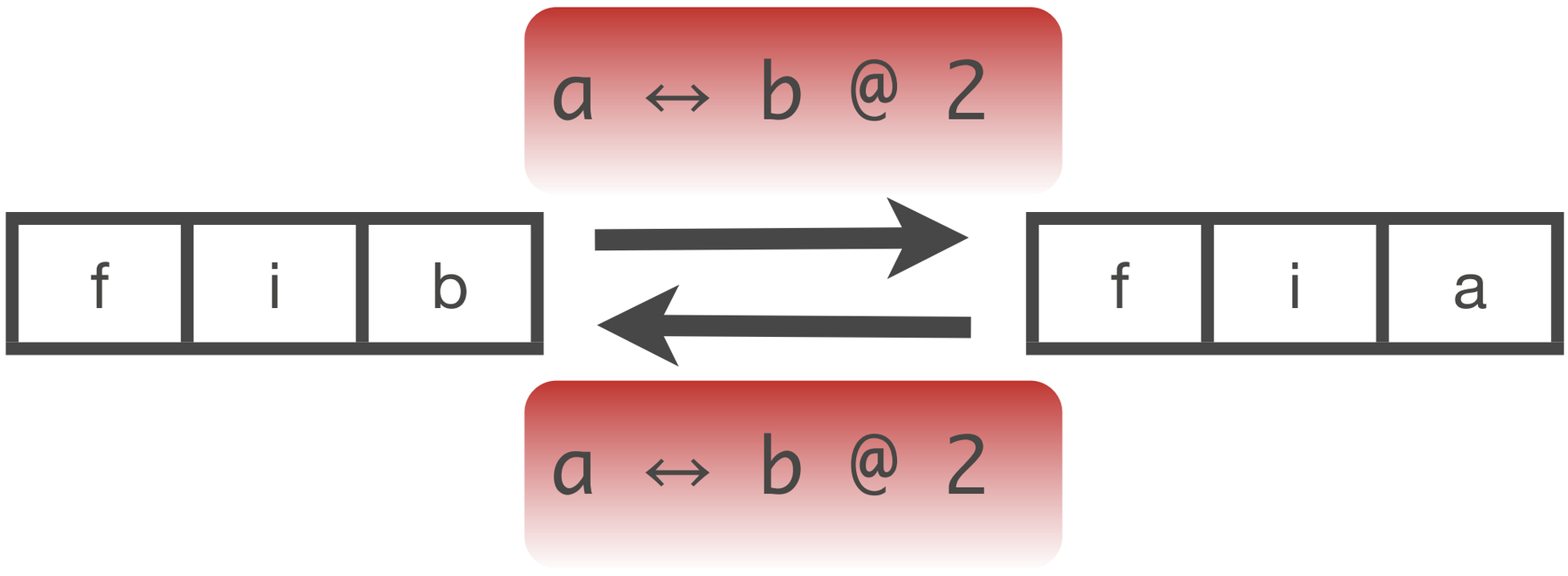
$$a \leftrightarrow b \ @ \ 2$$

$$a \leftrightarrow b \ @ \ 2$$

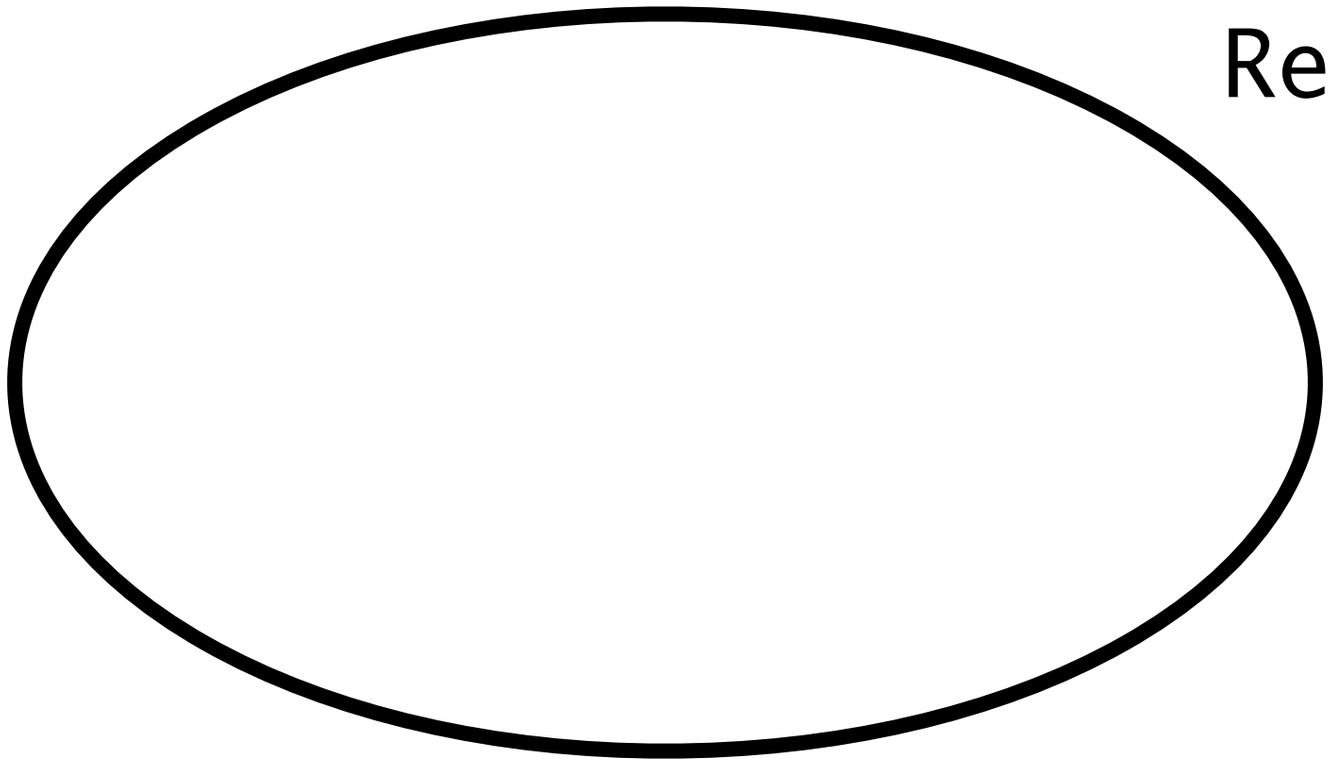
Basic Patches

- "Repository" is a char vector of length n

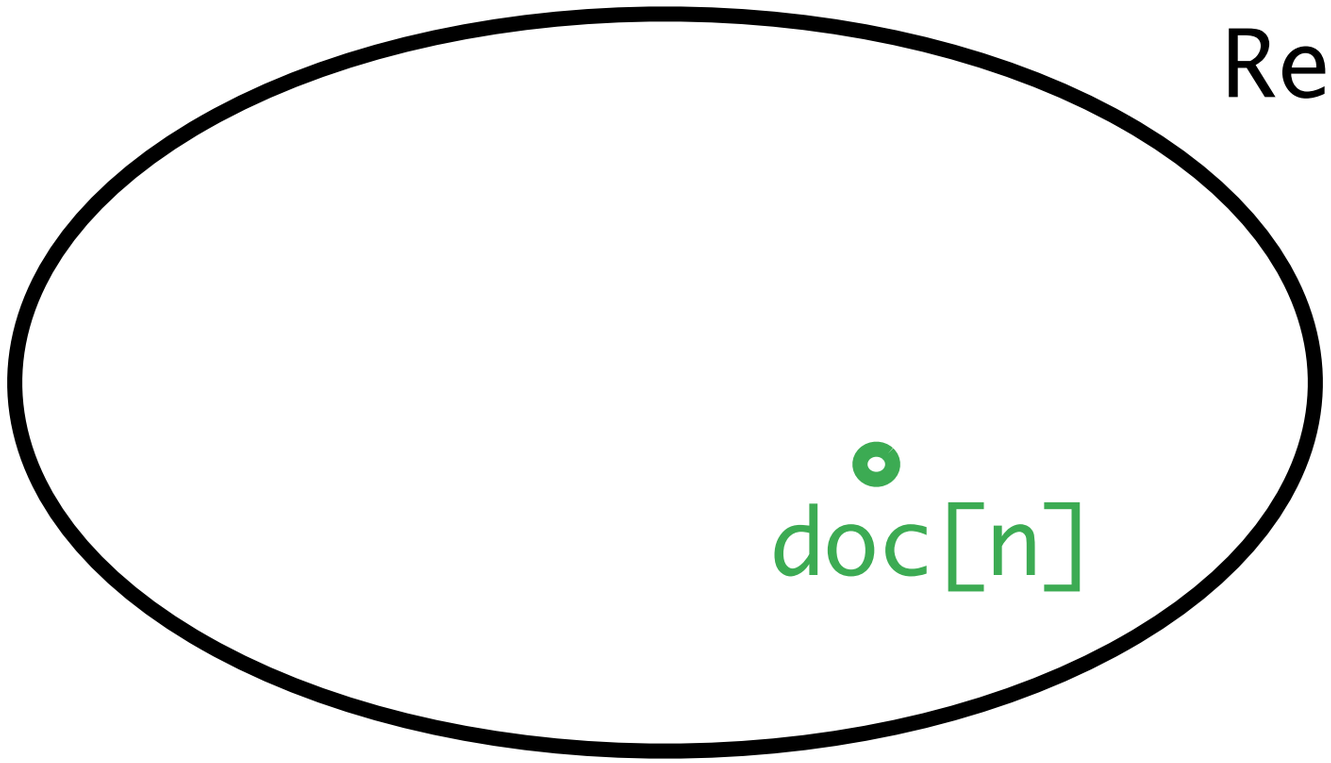
- Basic patch is `a ↔ b @ i` where `i<n`



Patches as a HIT

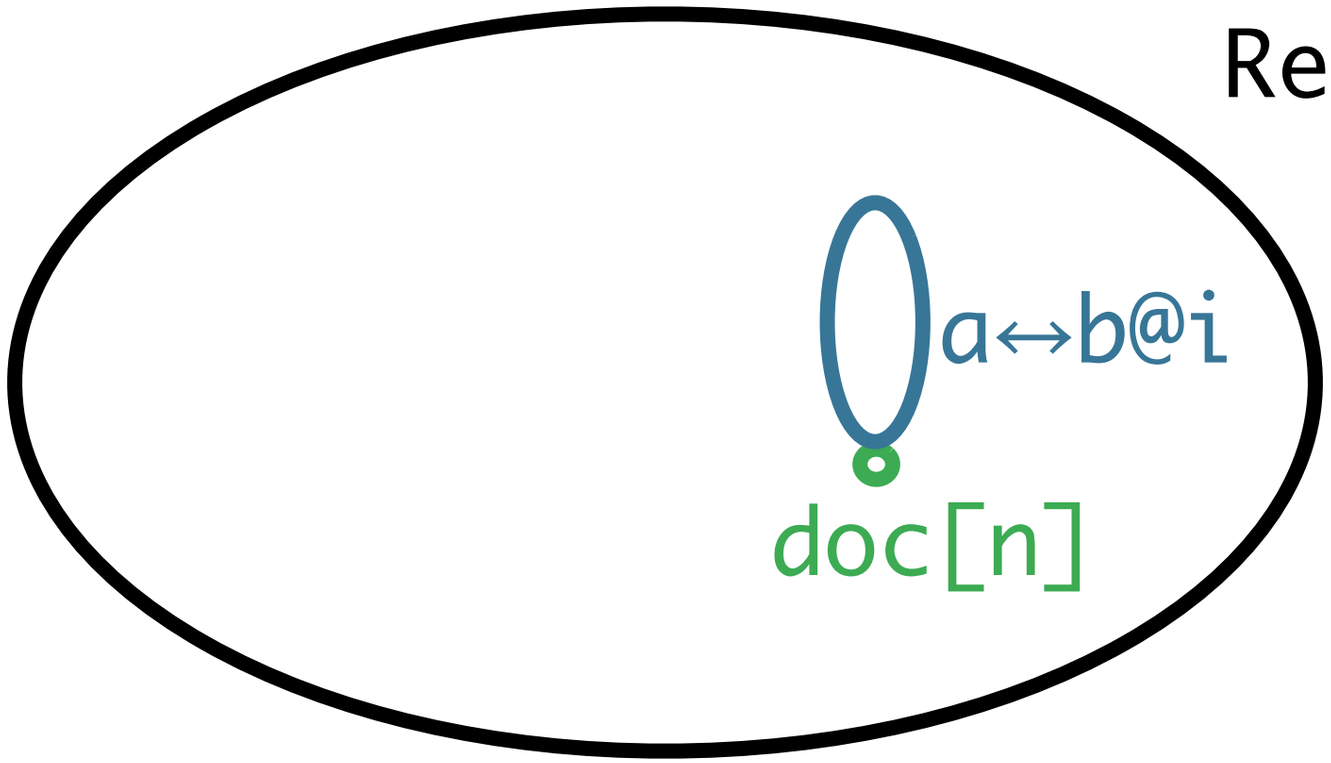


Patches as a HIT



points describe repository contents

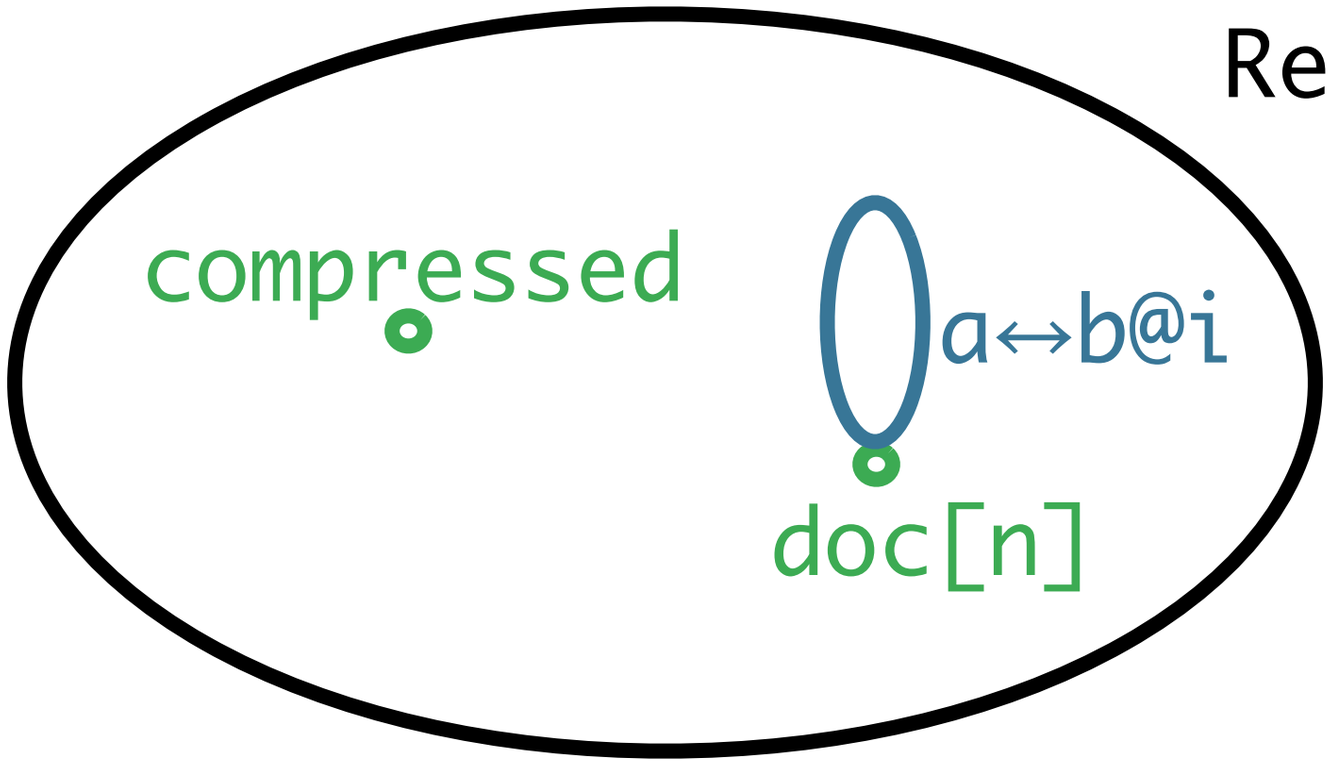
Patches as a HIT



points describe repository contents

paths are patches

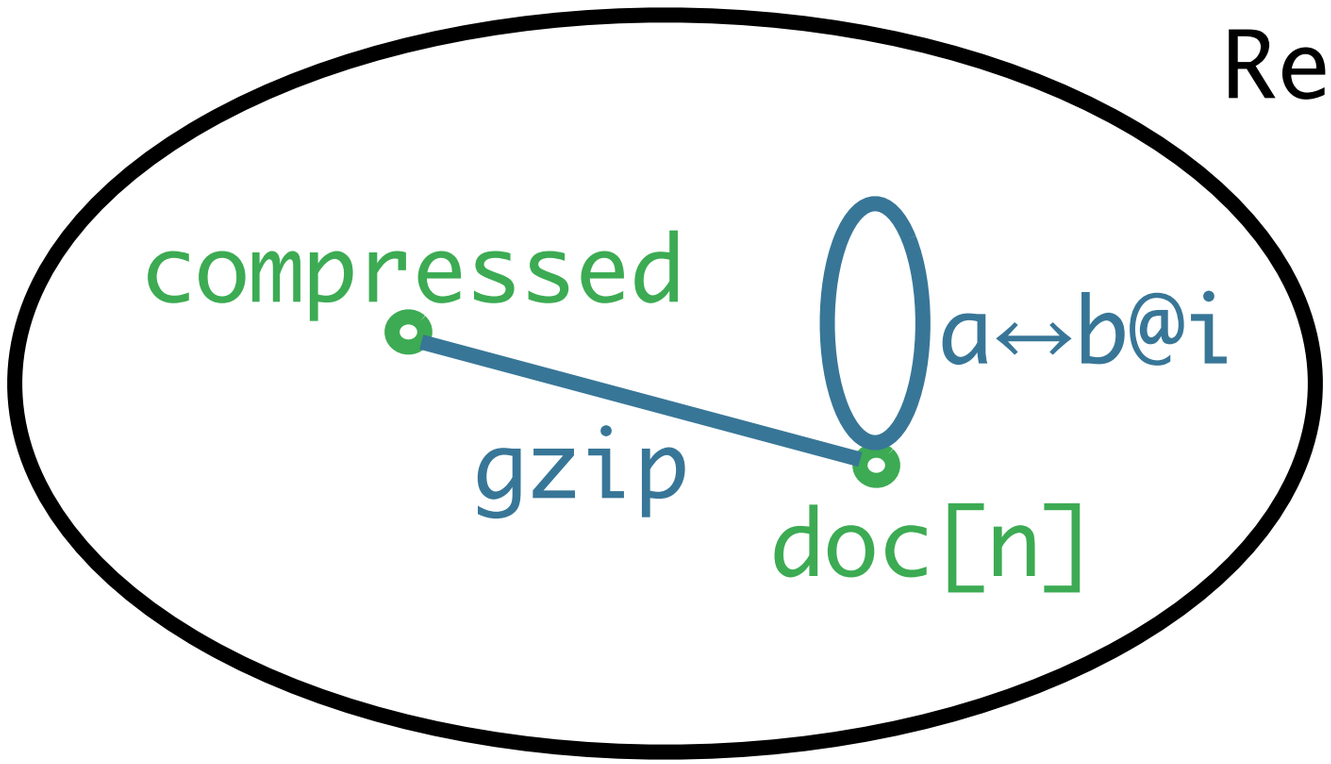
Patches as a HIT



points describe repository contents

paths are patches

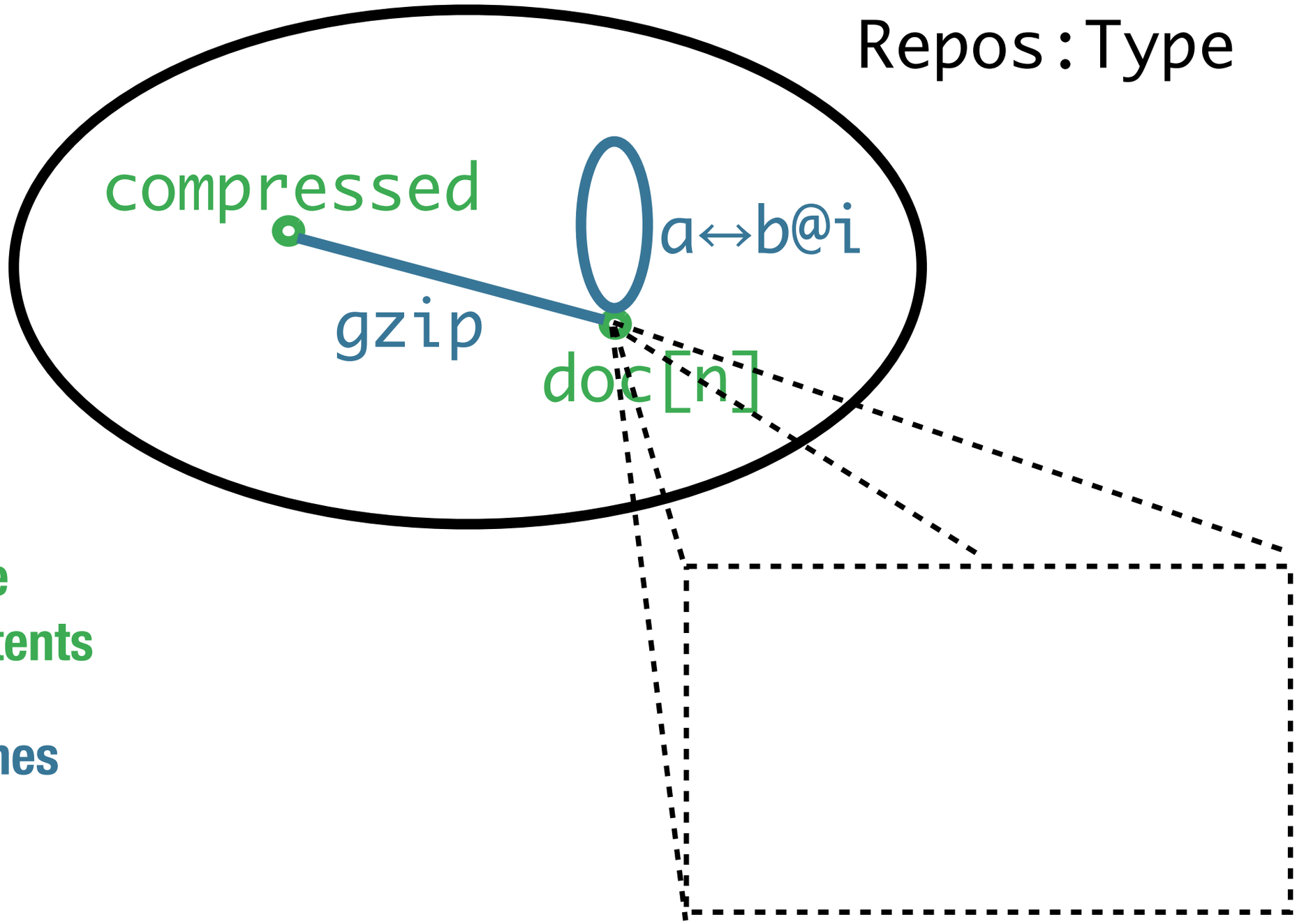
Patches as a HIT



points describe repository contents

paths are patches

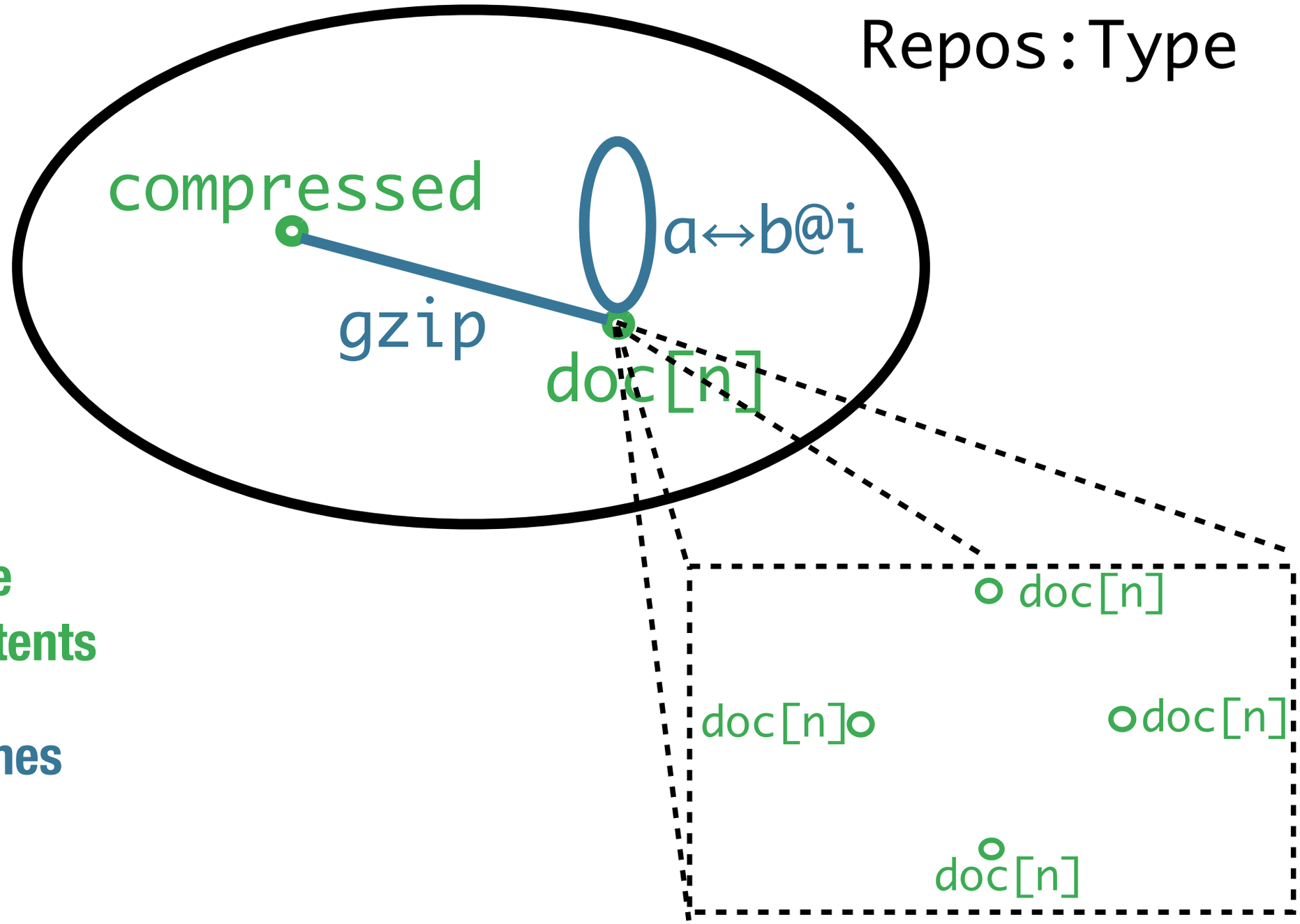
Patches as a HIT



points describe repository contents

paths are patches

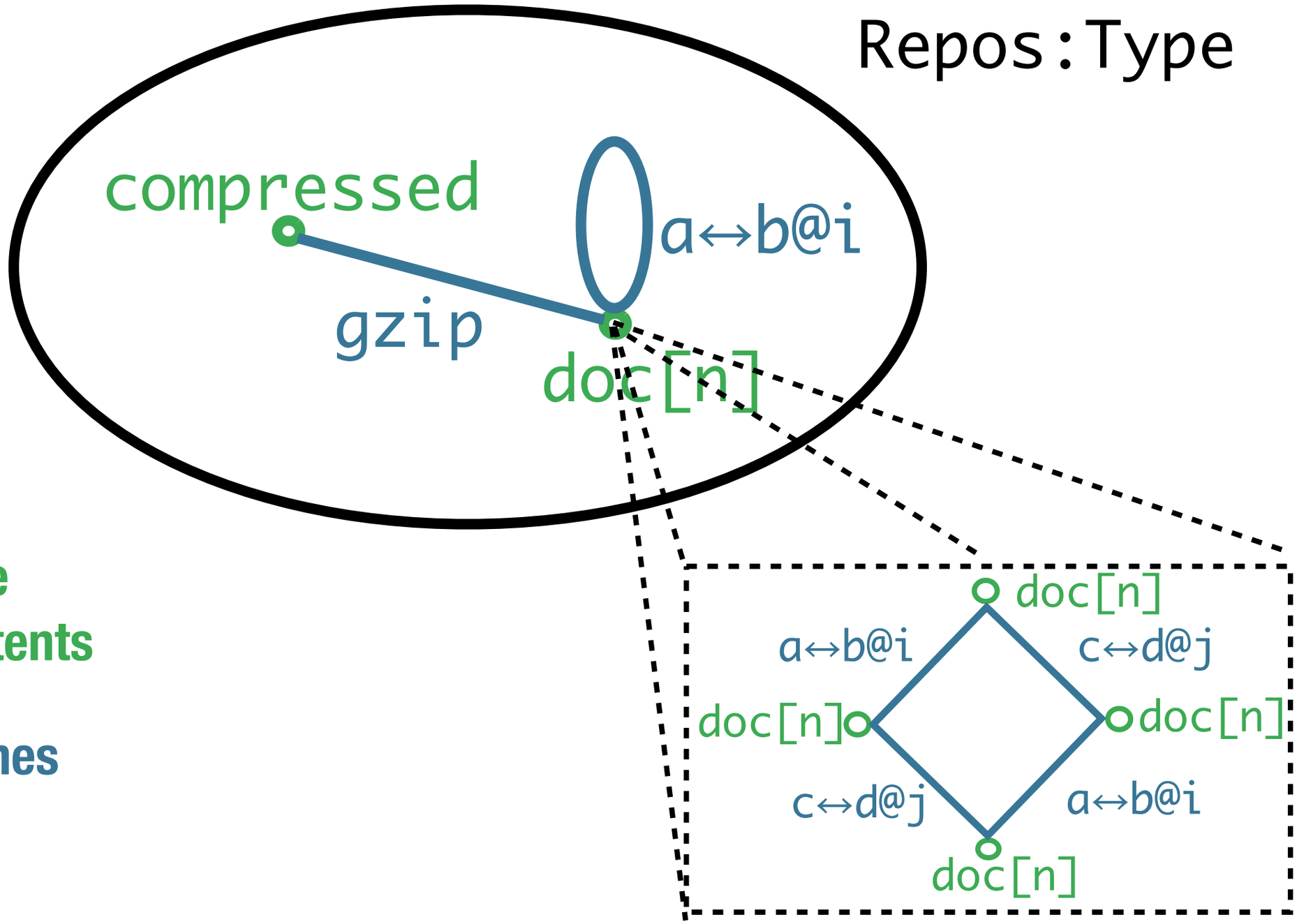
Patches as a HIT



points describe repository contents

paths are patches

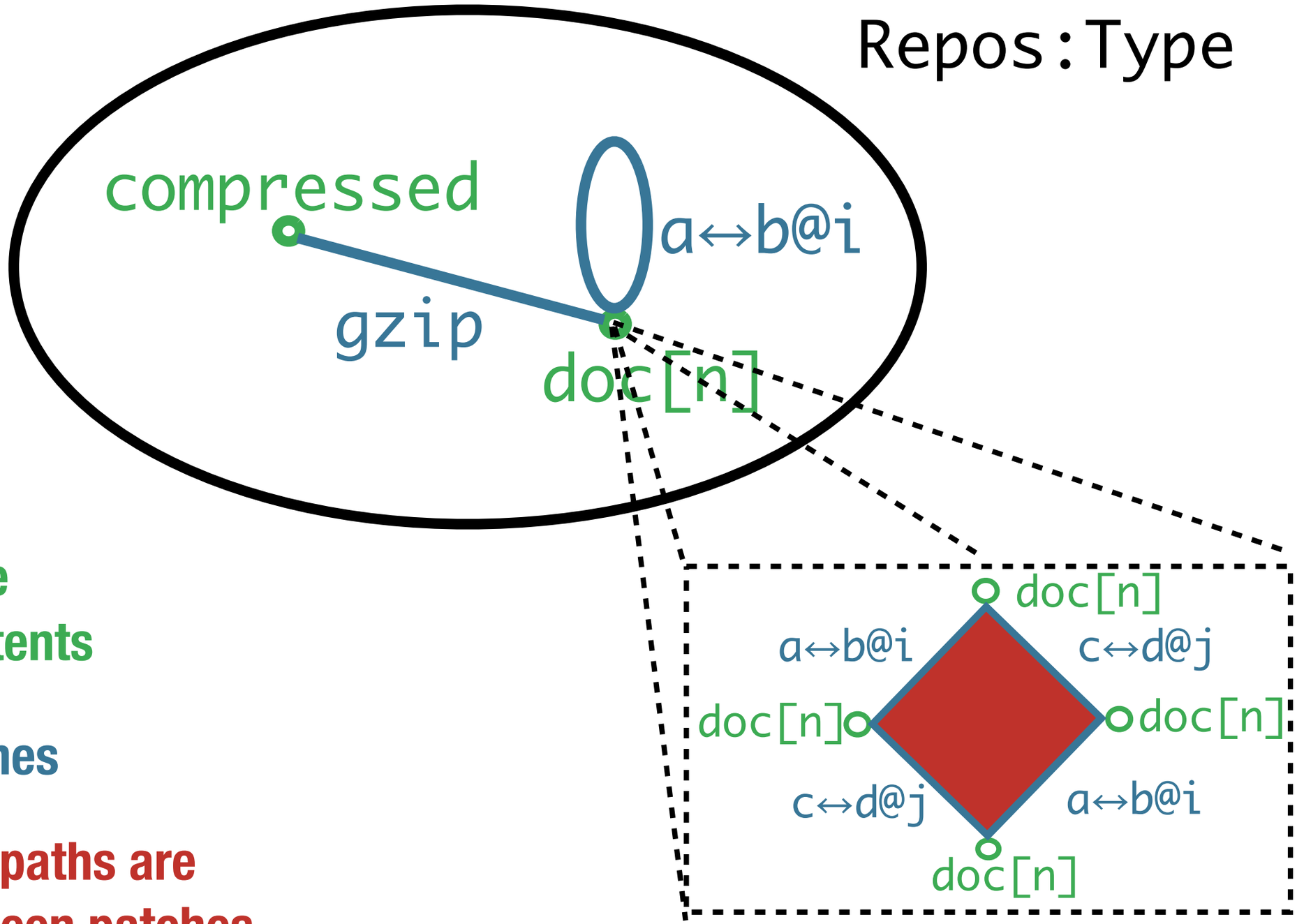
Patches as a HIT



points describe repository contents

paths are patches

Patches as a HIT

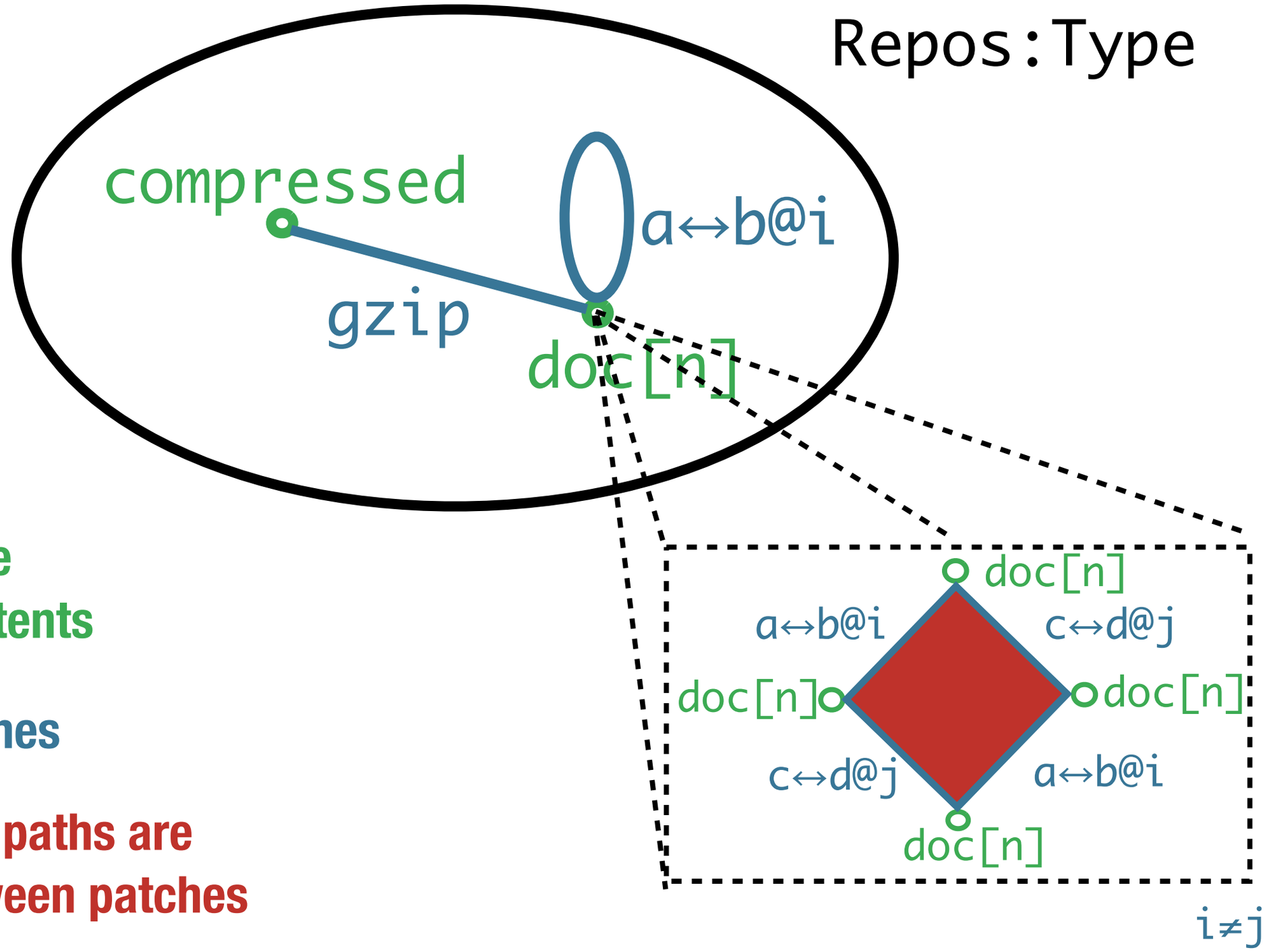


points describe repository contents

paths are patches

paths between paths are equations between patches

Patches as a HIT



points describe repository contents

paths are patches

paths between paths are equations between patches

Generators for HIT

Generators for HIT

```
Repos : Type
```

Generators for HIT

```
Repos : Type

doc[n]      : Repos
compressed : Repos
```

Generators for HIT

```
Repos : Type

doc[n]      : Repos
compressed : Repos

(a↔b@i) : doc[n] = doc[n] if a,b:Char, i<n
gzip : doc[n] = compressed
```

Generators for HIT

```
Repos : Type

doc[n]      : Repos
compressed  : Repos

(a↔b@i) : doc[n] = doc[n] if a,b:Char, i<n
 gzip : doc[n] = compressed

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

Patches Include…

```
id      (identity)
!p      (undo)
q o p   (composition)
```

Patch Equality Includes...

Patch Equality Includes...

Group laws:

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

po(qor) = (poq)or

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

Patch Equality Includes...

Group laws:

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

po(qor) = (poq)or

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

Equiv. relation

```
p=p

p=q if q=p

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

Patch Equality Includes...

Group laws:

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

po(qor) = (poq)or

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

Equiv. relation

```
p=p

p=q if q=p

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

Congruence

```
!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: Repos

Points: doc[n]

Paths:

a↔b@i

Paths between paths:

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

RepoDesc recursion

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

RepoDesc recursion

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

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

RepoDesc recursion

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

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

RepoDesc recursion

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

- map the element generators of `Repos`
  to elements of `A`
- map the equality generators of `Repos`
  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 : Repos → A`
it suffices to give

RepoDesc recursion

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

```
f(doc[n]) := … : A
```

RepoDesc recursion

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

`f(doc[n]) := … : A`

`f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])`

RepoDesc recursion

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

```
f(doc[n]) := … : A

f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])

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

RepoDesc recursion

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

```
f(doc[n]) := … : A
f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])
f₂(compose a b c d i j i≠j) := …
  : f₁((a↔b@i)o(c↔d@j))
  = f₁((c↔d@j)o(a↔b@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 : Repos → A`
it suffices to give

```
f(doc[n]) := … : A

f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])

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

RepoDesc recursion

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

```
f(doc[n]) := … : A
f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])
f₂(compose a b c d i j i≠j) := …
  : f₁((a↔b@i)o(c↔d@j))
  = f₁((c↔d@j)o(a↔b@j))
```

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

RepoDesc recursion

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

`f(doc[n]) := … : A`

`f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])`

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

`=f₁(a↔b@i)o`
`f₁(c↔d@j)`

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

RepoDesc recursion

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

`f(doc[n]) := … : A`

`f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])`

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

RepoDesc recursion

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

```
f(doc[n]) := … : A

f₁(a↔b@i) := … : f(doc[n]) = f(doc[n])

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

All functions on Repos respect patches

All functions on patches respect patch equality

Interpreter

Goal is to define:

```
interp : doc[n] = doc[n]
       → Bijection (Vec Char n) (Vec Char n)
```

Interpreter

Goal is to define:

```
interp : doc[n] = doc[n]
       → Bijection (Vec Char n) (Vec Char n)
interp(id) = (λx.x, …)
interp(q ∘ p) = (interp q) ∘ᵇ (interp p)
interp(!p) = !ᵇ (interp p)
interp(a↔b@i) = swapat a b i
```

Interpreter

Goal is to define:

```
interp : doc[n] = doc[n]
       → Bijection (Vec Char n) (Vec Char n)
interp(id) = (λx.x, …)
interp(q ∘ p) = (interp q) ∘ᵇ (interp p)
interp(!p) = !ᵇ (interp p)
interp(a↔b@i) = swapat a b i
```

But only tool available is RepoDesc recursion: no direct recursion over paths

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

Need to pick A *and define*

```
  I(doc[n]) := … : A
I₁(a↔b@i) := … : I(doc[n]) = I(doc[n])
I₂(compose) := …
```

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

Key idea: pick A = Type *and define*

```
  I(doc[n]) := … : Type
  I₁(a↔b@i) := … : I(doc[n]) = I(doc[n])
  I₂(compose) := …
```

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

Key idea: pick A = Type *and define*

```
  I(doc[n]) := Vec Char n : Type
  I₁(a↔b@i) := … : I(doc[n]) = I(doc[n])
  I₂(compose) := …
```

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

Key idea: pick A = Type *and define*

```
  I(doc[n]) := Vec Char n : Type
  I₁(a↔b@i) := … : Vec Char n = Vec Char n

  I₂(compose) := …
```

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

Key idea: pick A = Type *and define*

```
  I(doc[n]) := Vec Char n : Type
  I₁(a↔b@i) := ua(swapat a b i)
                    : Vec Char n = Vec Char n
  I₂(compose) := …
```

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

Key idea: pick A = Type *and define*

```
  I(doc[n]) := Vec Char n : Type
I₁(a↔b@i) := ua(swapat a b i)
                : Vec Char n = Vec Char n
I₂(compose) := …
```

Voevodky's univalence axiom ⊃
bijective types are equal

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

Key idea: pick A = Type *and define*

```
  I(doc[n]) := Vec Char n : Type
  I₁(a↔b@i) := ua(swapat a b i)
                    : Vec Char n = Vec Char n
  I₂(compose) := <proof about swapat>
```

```
interp : doc[n]=doc[n]
       → Bijection (Vec Char n) (Vec Char n)
```

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

Key idea: pick A = Type *and define*

```
  I(doc[n]) := Vec Char n : Type
I₁(a↔b@i) := ua(swapat a b i)
                    : Vec Char n = Vec Char n
I₂(compose) := <proof about swapat>
```

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

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

Satisfies the desired equations (as propositional equalities):

```
interp(id) = (λx.x, …)
interp(q o p) = (interp q) o_b (interp p)
interp(!p) = !_b (interp p)
interp(a↔b@i) = swapat a b i
```

Summary

Summary

- `I : Repos → Type` interprets `Repos` as `Types`, patches as bijections, satisfying patch equalities

Summary

- `I : Repos → Type` interprets `Repos` 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 : Repos → Type` interprets `Repos` 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`,`!`,...
- Univalence lets you give a computational model of equality proofs (here, patches); guaranteed to satisfy laws

Summary

- `I : Repos → Type` interprets `Repos` 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`,`!`,...

- Univalence lets you give a computational model of equality proofs (here, patches); guaranteed to satisfy laws

- Shorter definition and code:
  1 basic patch & 4 basic axioms of equality, instead of
  4 patches & 14 equations

Outline

1. Computer-checked homotopy theory
2. Computer-checked software
3. But there's a catch!

The Catch

- Operational semantics of univalence and HITs is an open problem in general: can't run these programs yet

- Some progress and some special cases are known:
  Licata&Harper, POPL'12
  Coquand&Barras, '13
  Shulman, '13
  Bezem&Coquand&Huber, '13

- Would support proof automation and programming applications

Conclusion

Papers and code

1. Fundamental group of the circle [LICS'13]
   $\pi_n(S^n) = \mathbb{Z}$ [CPP'13]
   Formal homotopy: `github.com/dlicata335/`

2. Computational interpretation of 2D type theory [POPL'12]

3. Domain-specific program verification logics [thesis+MFPS'11]

4. The HoTT Book: doing math informally in Homotopy Type Theory

5. Blog: `homotopytypetheory.org`

Homotopy Type Theory