What is Homotopy Type Theory?

Dan Licata
Wesleyan University

Synthetic geometry

Euclid's postulates

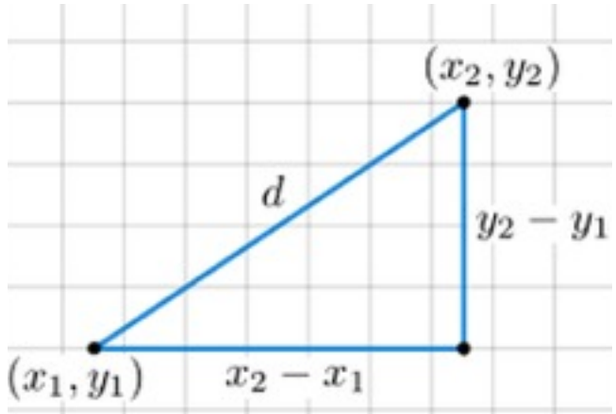
1. **To draw a straight line from any point to any point.**
2. **To produce a finite straight line continuously in a straight line.**
3. **To describe a circle with any center and distance.**
4. **That all right angles are equal to one another.**
5. **Given a line and a point not on it, there is exactly one line through the point that does not intersect the line**

Synthetic geometry

Euclid's postulates

1. **To draw a straight line from any point to any point.**
2. **To produce a finite straight line continuously in a straight line.**
3. **To describe a circle with any center and distance.**
4. **That all right angles are equal to one another.**
5. **Given a line and a point not on it, there is exactly one line through the point that does not intersect the line**

Cartesian



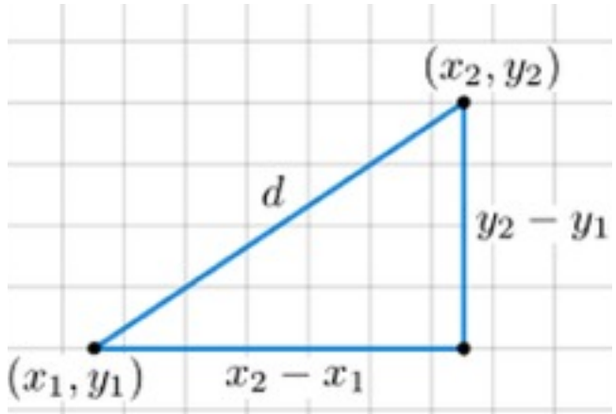
Synthetic geometry

Euclid's postulates

1. **To draw a straight line from any point to any point.**
2. **To produce a finite straight line continuously in a straight line.**
3. **To describe a circle with any center and distance.**
4. **That all right angles are equal to one another.**
5. **Given a line and a point not on it, there is exactly one line through the point that does not intersect the line**

models

Cartesian



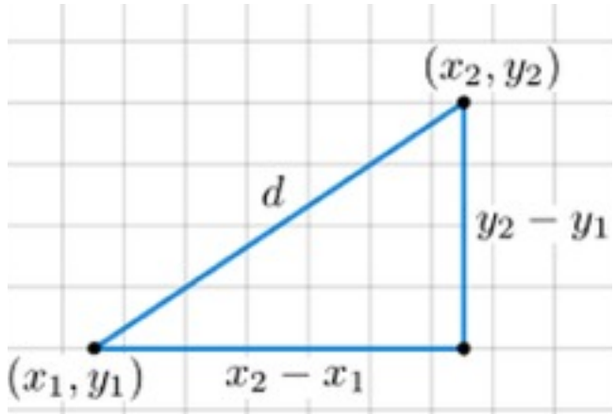
Synthetic geometry

Euclid's postulates

1. **To draw a straight line from any point to any point.**
2. **To produce a finite straight line continuously in a straight line.**
3. **To describe a circle with any center and distance.**
4. **That all right angles are equal to one another.**

models

Cartesian



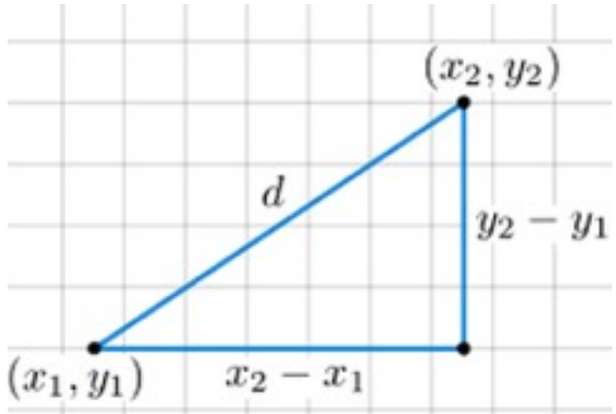
Synthetic geometry

Euclid's postulates

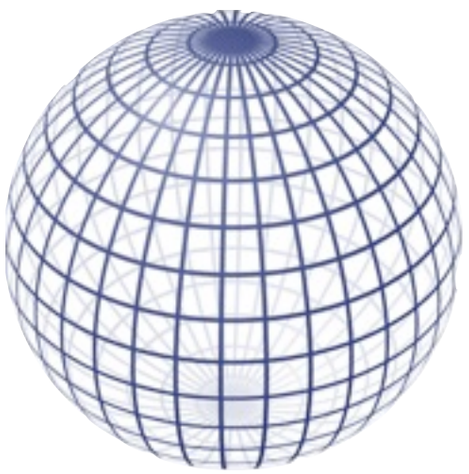
1. **To draw a straight line from any point to any point.**
2. **To produce a finite straight line continuously in a straight line.**
3. **To describe a circle with any center and distance.**
4. **That all right angles are equal to one another.**

models

Cartesian



Spherical



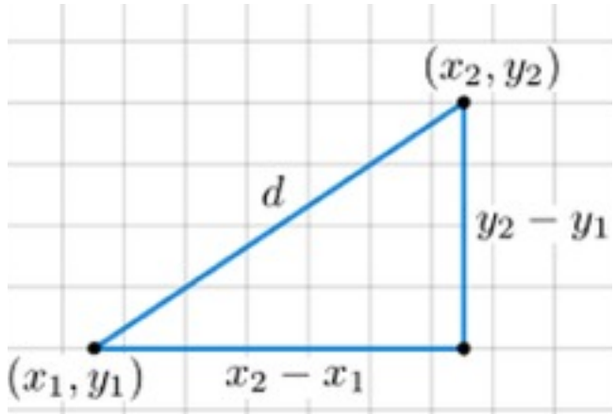
Synthetic geometry

Euclid's postulates

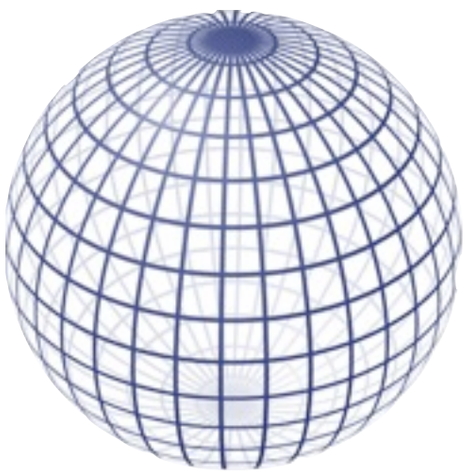
1. **To draw a straight line from any point to any point.**
2. **To produce a finite straight line continuously in a straight line.**
3. **To describe a circle with any center and distance.**
4. **That all right angles are equal to one another.**
5. **Two distinct lines meet at two antipodal points.**

models

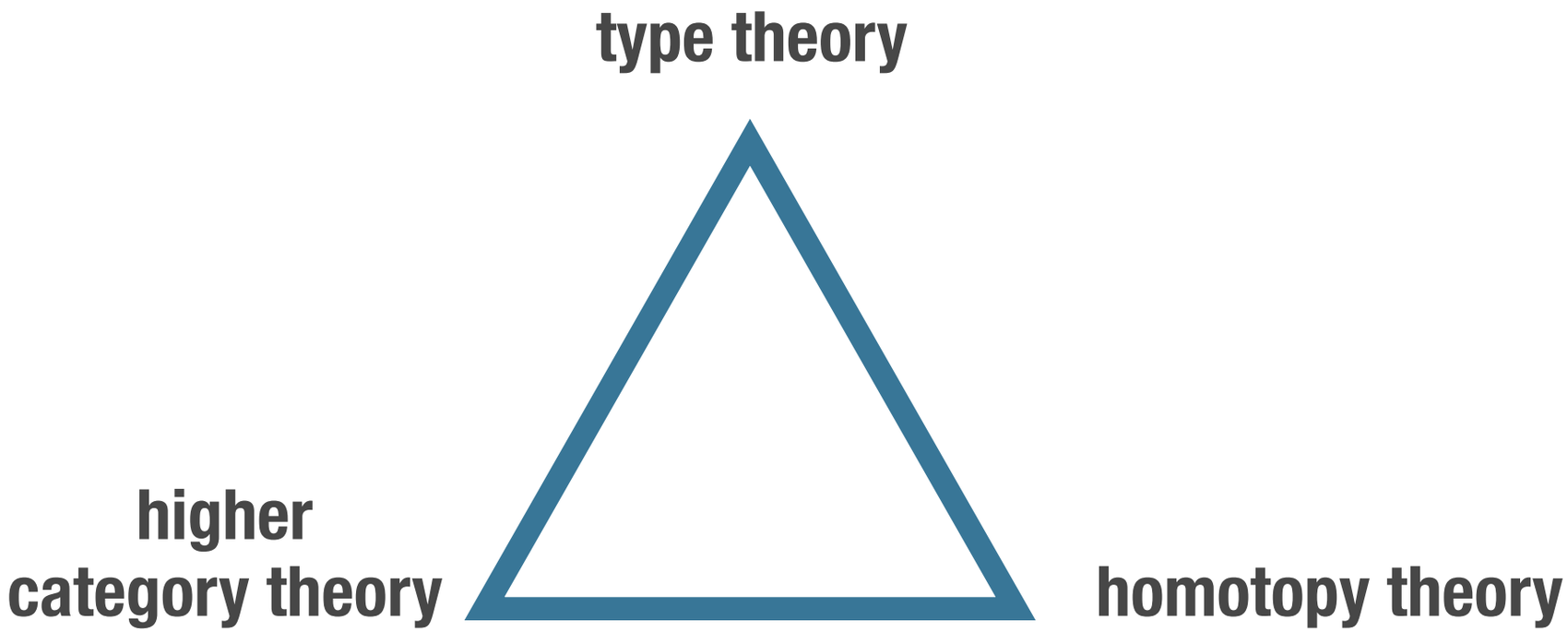
Cartesian



Spherical



Homotopy type theory



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

Homotopy type theory is a synthetic theory of spaces

Equality type

$$x : A$$

$$p : x =_A y \qquad \text{equality type}$$

Equality type

$$x : A$$

$$p : x =_A y$$ equality type

```
Inductive paths {A : Type} (a : A) : A -> Type :=
  idpath : paths a a.
```

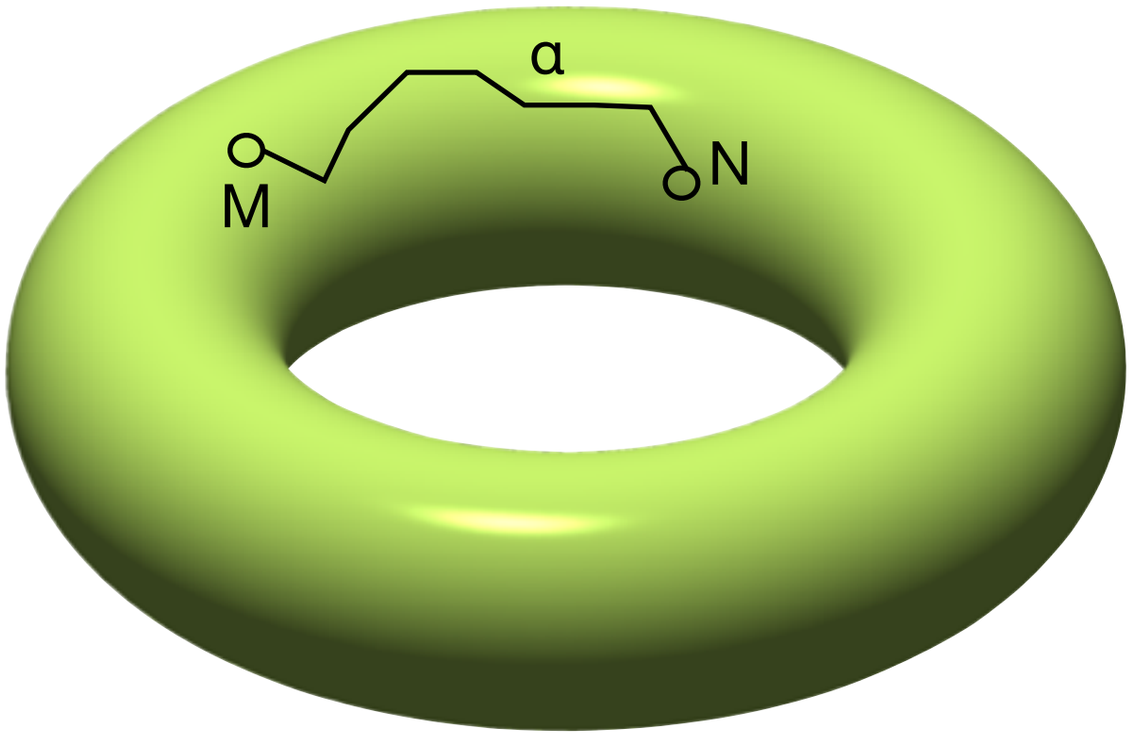
Equality type

$$x : A$$

$$p : x =_A y \qquad \text{equality type}$$

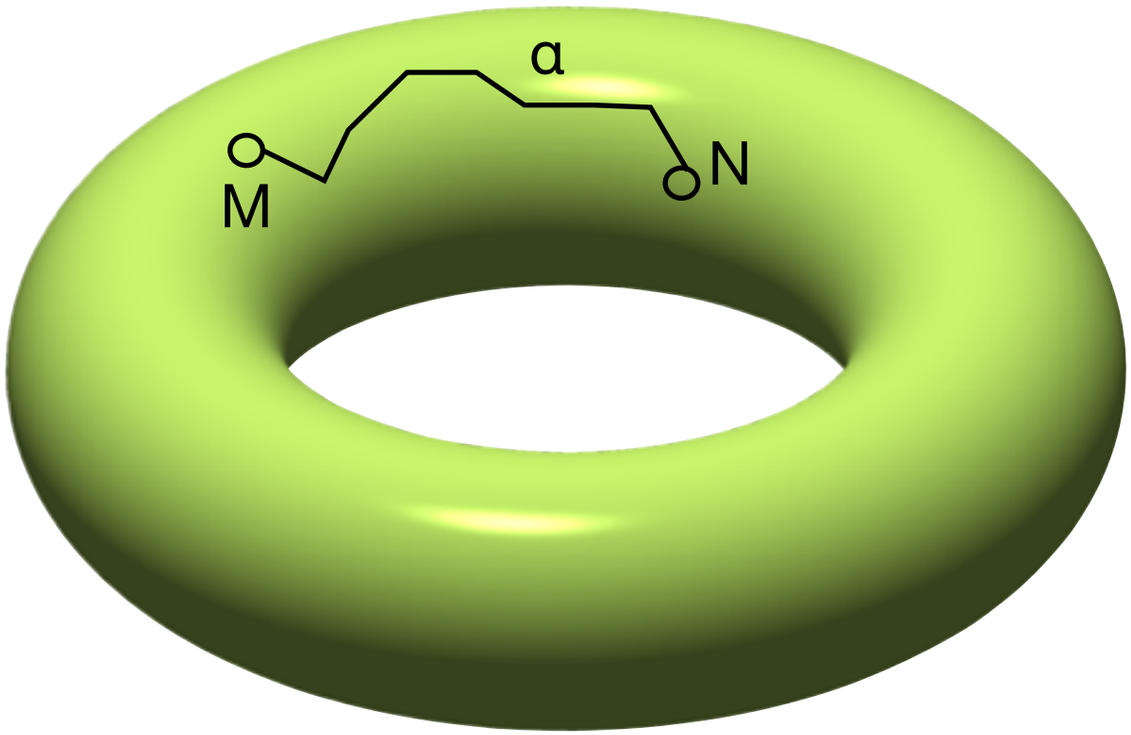
```
Inductive paths {A : Type} (a : A) : A -> Type :=
  idpath : paths a a.

Notation "x = y"
```

Types as spaces



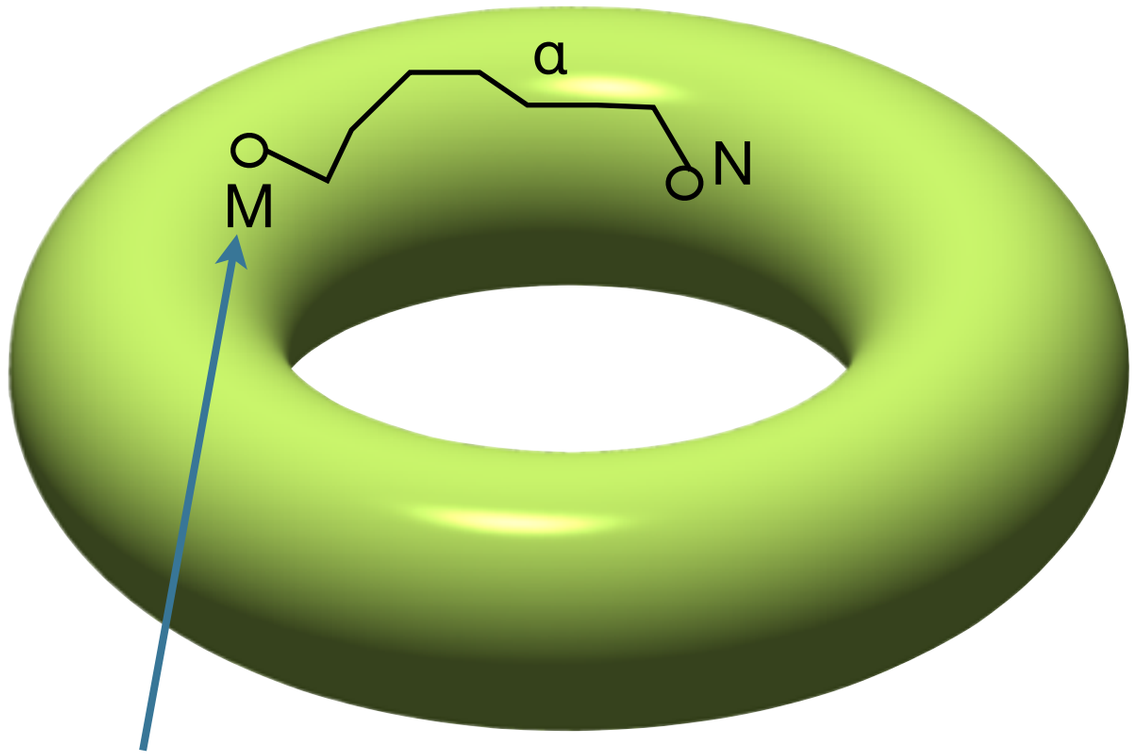
Types as spaces

type A is a space



Types as spaces

type A is a space



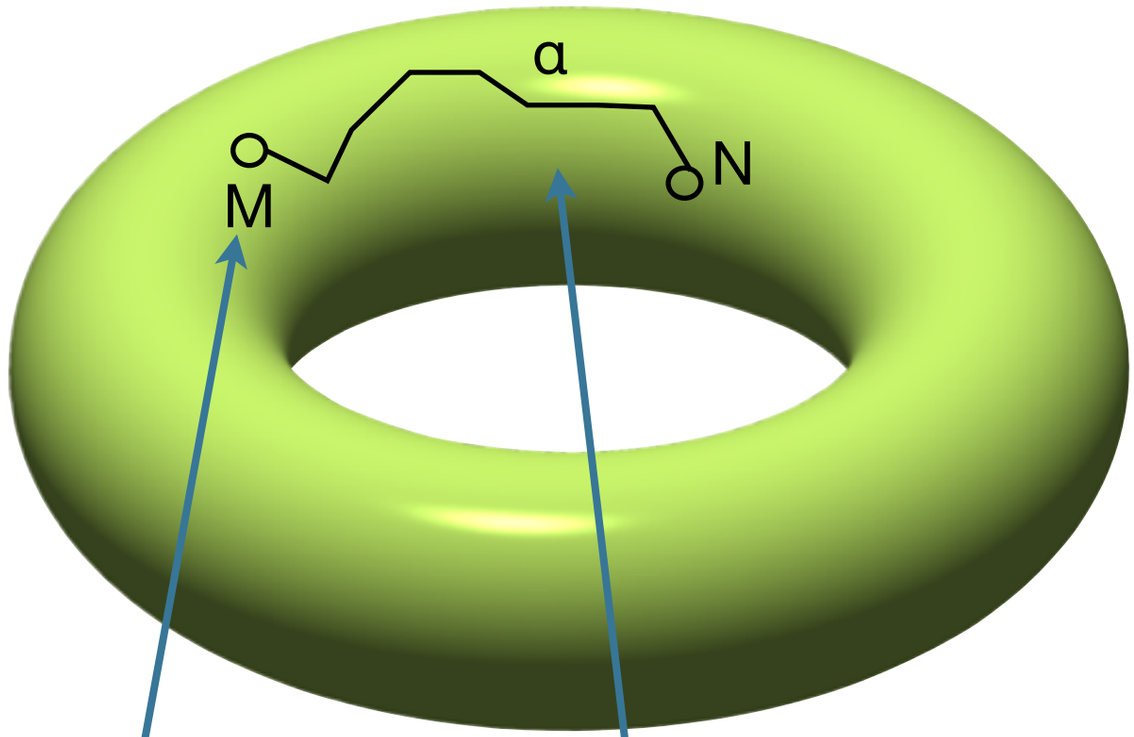
programs

M : A

are points

Types as spaces

type A is a space



programs

$M : A$

are points

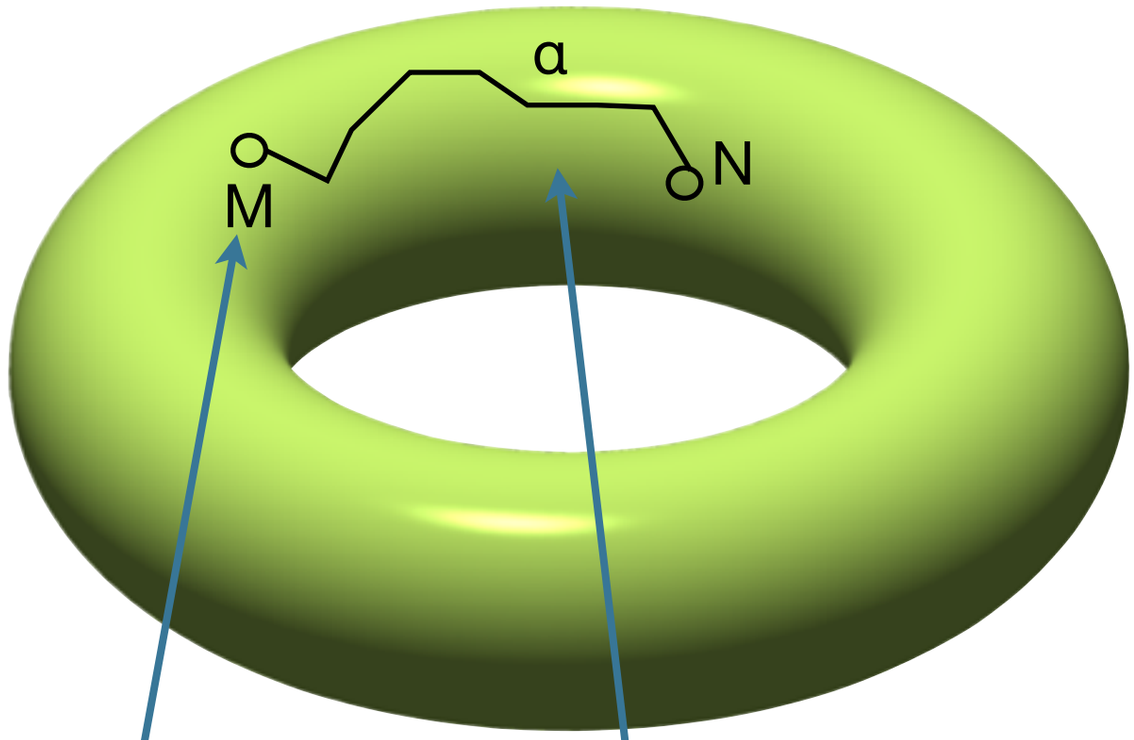
proofs of equality

$\alpha \; : \; M =_A N$

are paths

Types as spaces

type A is a space



programs

$M : A$

are points

proofs of equality

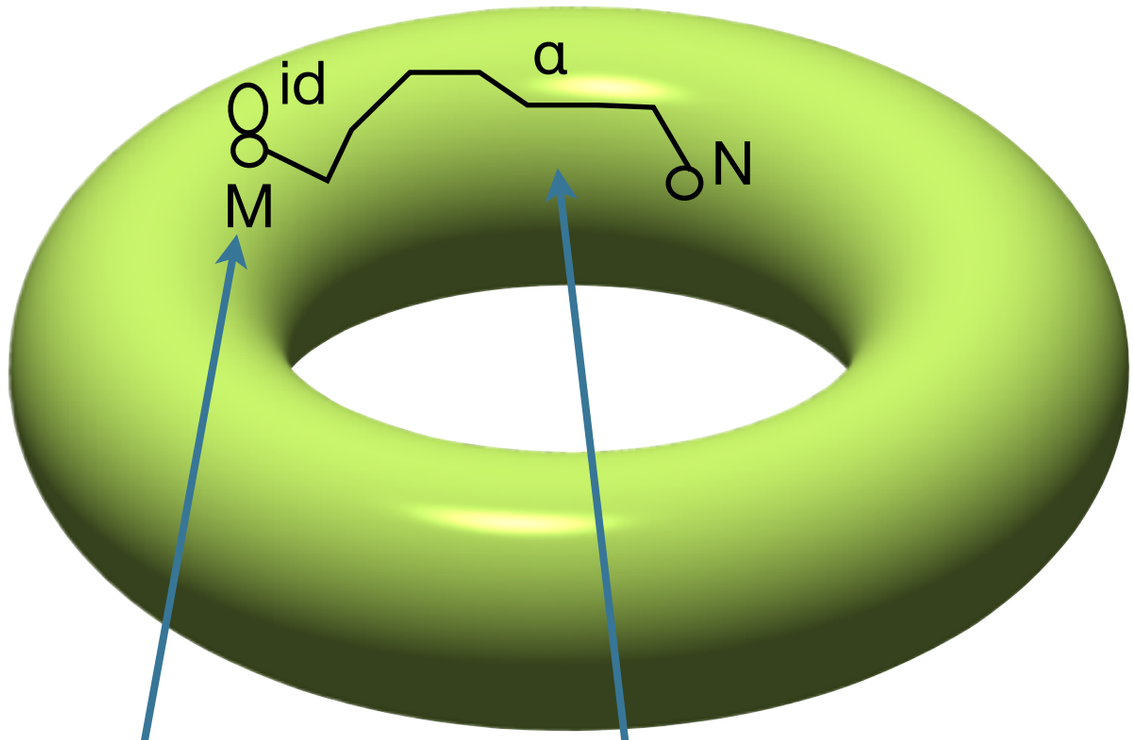
$\alpha : M =_A N$

are paths

path operations

Types as spaces

type A is a space



programs

$M : A$

are points

proofs of equality

$\alpha : M =_A N$

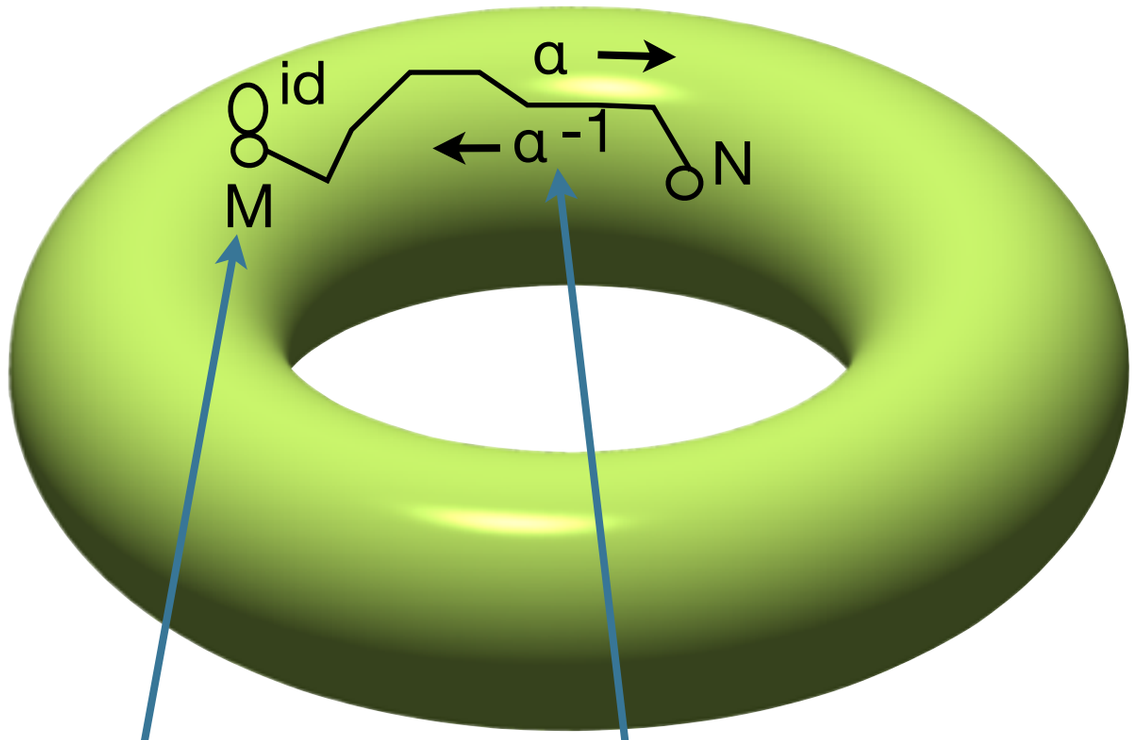
are paths

path operations

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

Types as spaces

type A is a space



programs
M : A
are points

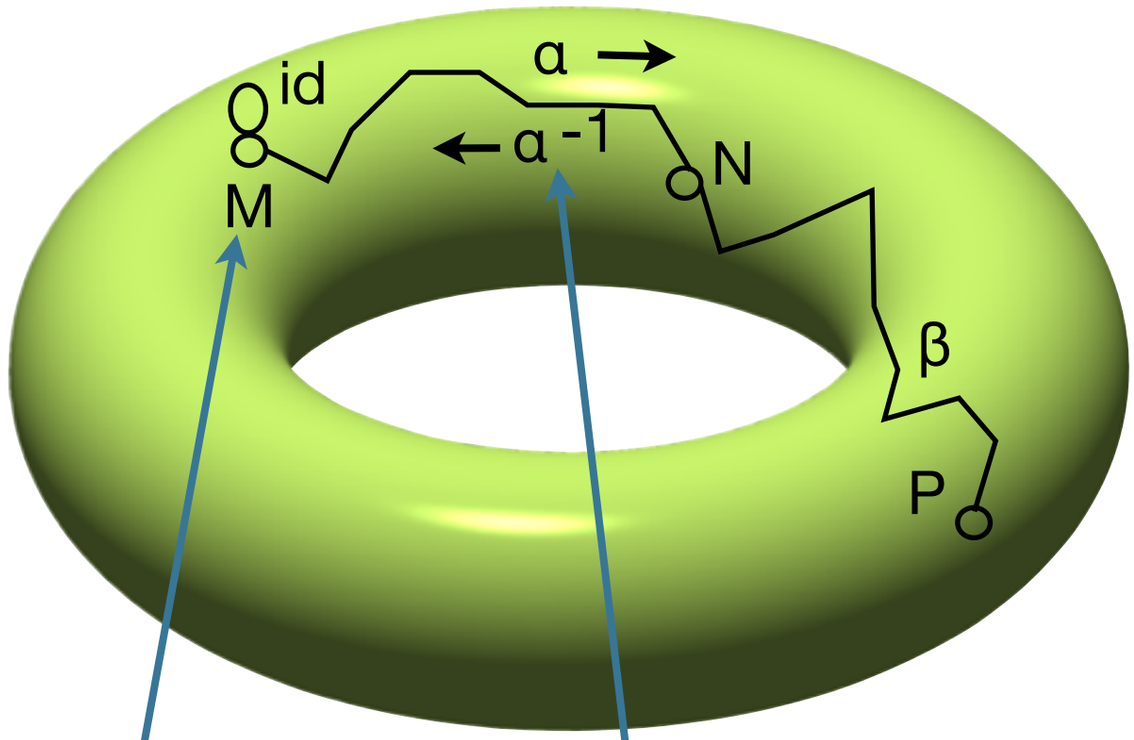
proofs of equality
α : M =A N
are paths

path operations

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

Types as spaces

type A is a space



programs

M : A

are points

proofs of equality

$\alpha : M =_A N$

are paths

path operations

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

Homotopy

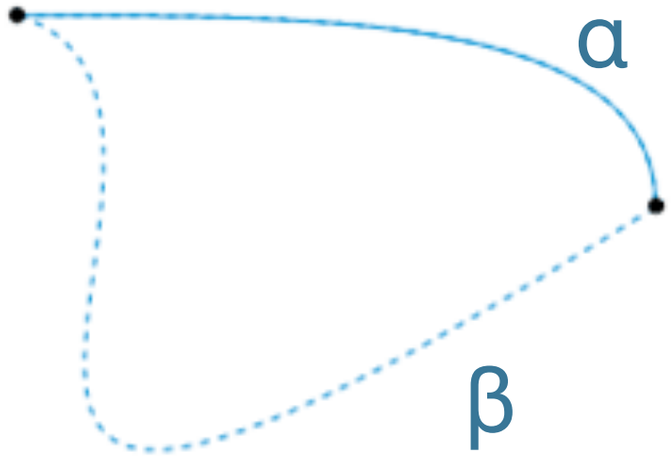
Deformation of one path into another

α

β

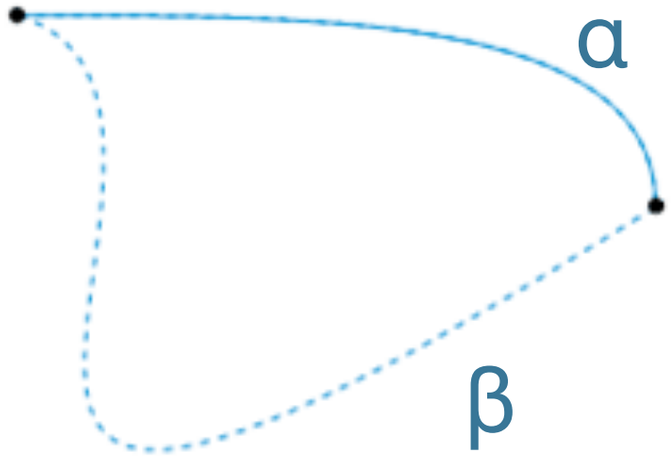
Homotopy

Deformation of one path into another



Homotopy

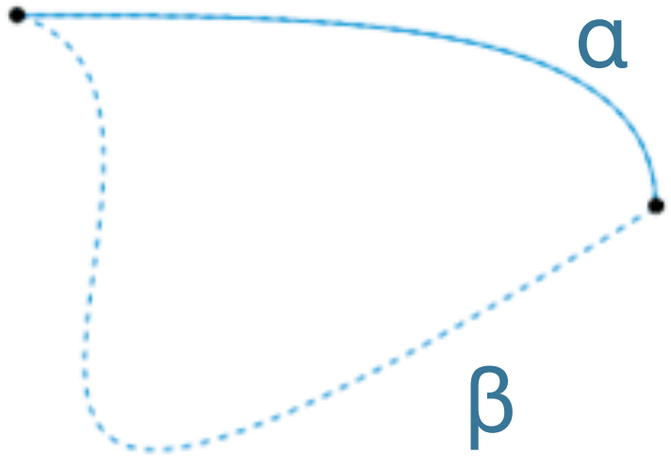
Deformation of one path into another



= 2-dimensional *path between paths*

Homotopy

Deformation of one path into another

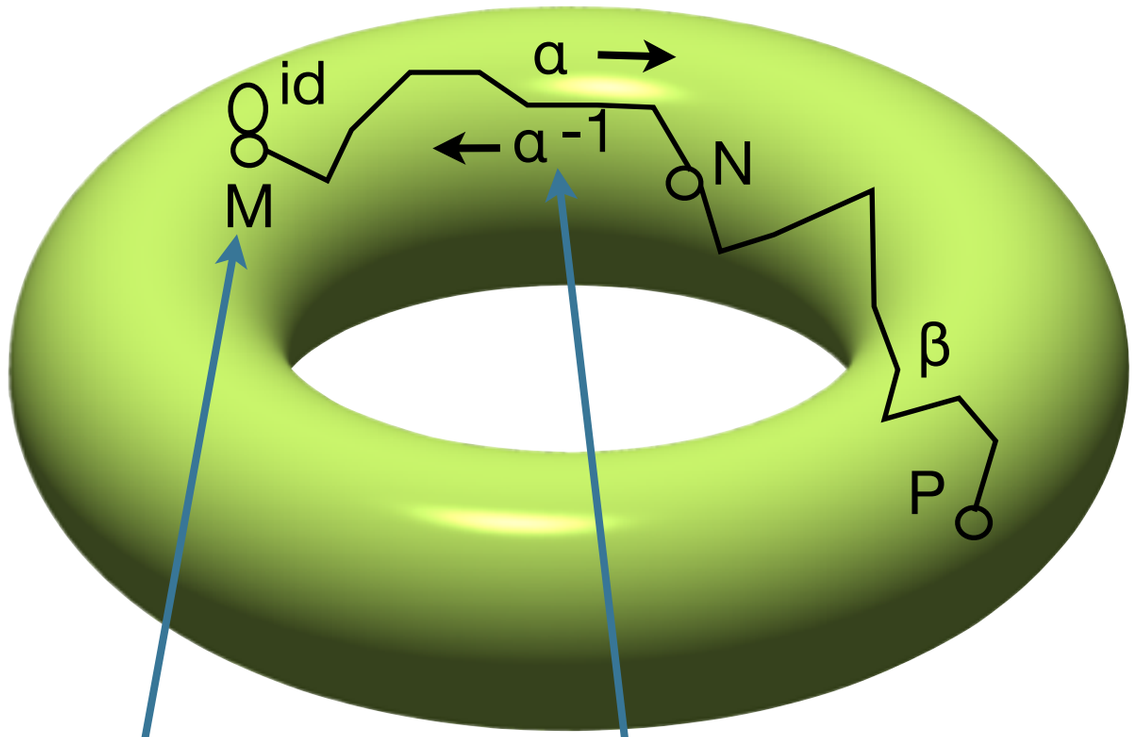


$\delta : \alpha =_{x=y} \beta$

= 2-dimensional *path between paths*

Types as spaces

type A is a space



programs
M : A
are points

proofs of equality
α : M =A N
are paths

path operations

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

homotopies

```
ul : id o α =M=N α
il : α⁻¹ o α =M=M id
asc : γ o (β o α)
        =M=P (γ o β) o α
```

Path induction `paths_ind`

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`,

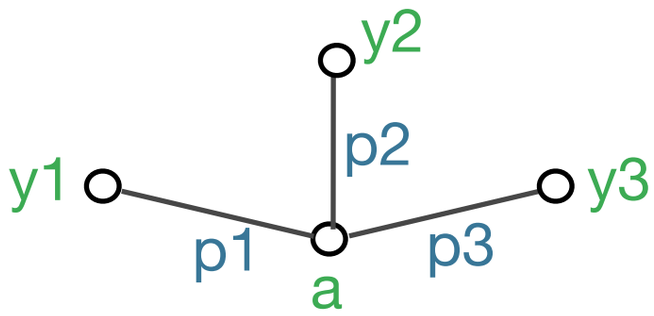
suffices to give an element of

`C(a,id)`

Path induction

`paths_ind`

Type of paths from a to somewhere



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)`

Equality type

$$x : A$$

$$p : x =_A y$$

$$? : p_1 =_{x=y} p_2$$

Equality type

$$x : A$$

$$p : x =_A y$$

$$? : p_1 =_{x=y} p_2$$

Uniqueness of Identity Proofs (UIP)

```
Definition UIP_ :=
  forall (x y:U) (p1 p2:x = y), p1 = p2.
```

Equality type

$$x : A$$

$$p : x =_A y$$

$$? : p_1 =_{x=y} p_2$$

Uniqueness of Identity Proofs (UIP)

```
Definition UIP_ :=
  forall (x y:U) (p1 p2:x = y), p1 = p2.
```

Proof-relevant equality

$$x : A$$

$$p : x =_A y$$

$$q : p_1 =_{x=y} p_2$$

Proof-relevant equality

$$x : A$$

$$p : x =_A y$$

$$q : p_1 =_{x=y} p_2$$

$$q_1 =_{p1=p2} q_2$$

Proof-relevant equality

$$x : A$$

$$p : x =_A y$$

$$q : p_1 =_{x=y} p_2$$

$$r : q_1 =_{p1=p2} q_2$$

Proof-relevant equality

$$x : A$$

$$p : x =_A y$$

$$q : p_1 =_{x=y} p_2$$

$$r : q_1 =_{p1=p2} q_2$$

$$\vdots$$

Homotopy groups of spheres

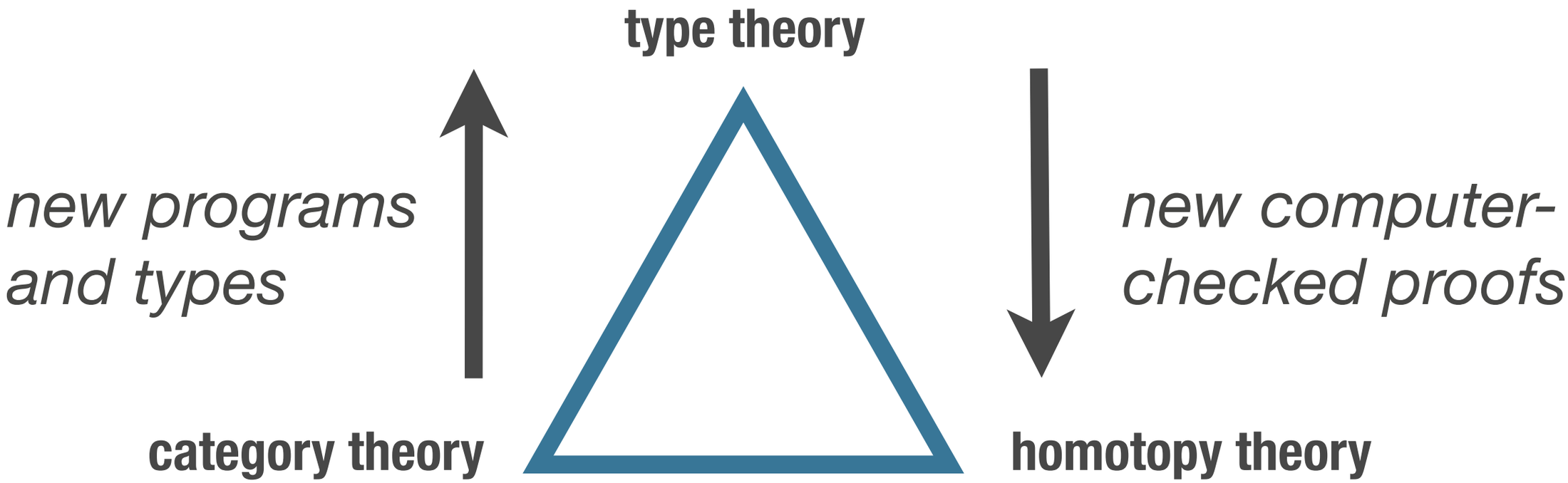
k^{th} homotopy group

n-dimensional sphere

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}
S^0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S^1	$\mathbb{Z}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S^2	0	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$
S^3	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$
S^4	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}\times\mathbb{Z}_{12}$	$\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{24}\times\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^3$	$\mathbb{Z}_{120}\times\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^5$
S^5	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{30}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^3$	$\mathbb{Z}_{72}\times\mathbb{Z}_2$
S^6	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_{60}$	$\mathbb{Z}_{24}\times\mathbb{Z}_2$	$\mathbb{Z}_2^3$
S^7	0	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	0	$\mathbb{Z}_2$	$\mathbb{Z}_{120}$	$\mathbb{Z}_2^3$
S^8	0	0	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	0	$\mathbb{Z}_2$	$\mathbb{Z}\times\mathbb{Z}_{120}$

[image from wikipedia]

Homotopy type theory



Univalence [Voevodsky]

Univalence [Voevodsky]

- *Equivalence of types* is a generalization to spaces of bijection of sets

Univalence [Voevodsky]

- *Equivalence of types* is a generalization to spaces of bijection of sets

- Univalence axiom:
  equality of types ($A =_{\mathsf{Type}} B$) is (equivalent to)
  equivalence of types (`Equiv A B`)

Univalence [Voevodsky]

- *Equivalence of types* is a generalization to spaces of bijection of sets

- Univalence axiom:
  equality of types ($A =_{\mathsf{Type}} B$) is (equivalent to)
  equivalence of types (`Equiv A B`)

- ∴ all structures/properties respect equivalence

Univalence [Voevodsky]

- *Equivalence of types* is a generalization to spaces of bijection of sets

- Univalence axiom:
  equality of types ($A =_{\mathsf{Type}} B$) is (equivalent to)
  equivalence of types (`Equiv A B`)

- ∴ all structures/properties respect equivalence

- Not by collapsing equivalence,
  but by exploiting proof-relevant equality:
  path induction *has computational content*

Univalence

- Transporting along an equality is a generic program that lifts equivalences
- Can do parametricity reasoning about modules
- Provides “right” equality for mathematical structures (groups, categories, …)

Higher inductive types

[Bauer,Lumsdaine,Shulman,Warren]

New way of forming types:

Inductive type specified by generators
not only for points (elements), but also for paths

Higher inductive types

- Subsume quotient types, which have been problematic in intensional type theory
- Direct constructive definitions of spaces and other mathematical concepts
- Some nascent programming applications
- New constructive definition of real numbers

HoTT Coq

- Indices matter [Grayson]
- Universe polymorphism [Sozeau,Tabareau]
- Private data types [Bertot]

HoTT Coq

- Indices matter [Grayson]
- Universe polymorphism [Sozeau,Tabareau]
- Private data types [Bertot]

- Can postulate univalence as axiom
- Can postulate/implement individual higher inductive types

Eventually

- Native implementation of higher inductive types [Barras, Shulman,Lumsdaine]
- Computational implementation of univalence and higher inductive types [Coquand,Huber,Bezem,Barras, Licata,Harper,Brunerie,Shulman, Altenkirch,Kaposi,Polansky…]

hProps and hSets

Prop

Prop plays several roles:

- Run-time erasability
- Can assume uniqueness of elements:
  `∀ A:Prop, ∀ p q:A, p=q`
- Can assume classical axioms such as excluded middle, choice, etc.
- Impredicativity

Prop vs hProp

	Prop	hProp
mechanism	built-in sort	definable predicate
erasability	yes	no
uniqueness	assume	yes
classicality	assume	assume
impredicativity	yes	assume*

Prop vs hProp

```
Definition hProp (A:Type) : Type :=
    forall x y : A, (x = y)
```

Prop vs hProp

```
Definition hProp (A:Type) : Type :=
    forall x y : A, (x = y)
```

- `forall x:A,B(x)` if `B(x)` is always an hProp

Prop vs hProp

```
Definition hProp (A:Type) : Type :=
    forall x y : A, (x = y)
```

- `forall x:A,B(x)` if `B(x)` is always an hProp
- record whose components are hProps

Prop vs hProp

```
Definition hProp (A:Type) : Type :=
    forall x y : A, (x = y)
```

- `forall x:A,B(x)` if `B(x)` is always an hProp
- record whose components are hProps
- `Wx:A.B` where `B(x)` is always an hProp

Prop vs hProp

```
Definition hProp (A:Type) : Type :=
    forall x y : A, (x = y)
```

- `forall x:A,B(x)` if `B(x)` is always an hProp
- record whose components are hProps
- `Wx:A.B` where `B(x)` is always an hProp
- **not** `bool`, sums, etc.

Prop vs hProp

```
Definition hProp (A:Type) : Type :=
   forall x y : A, (x = y)
```

- `forall x:A,B(x)` if `B(x)` is always an hProp **funext**
- record whose components are hProps
- `Wx:A.B` where `B(x)` is always an hProp
- **not** `bool`, sums, etc.

Prop vs hProp

```
Definition hProp(A:Type):Type :=
   forall x y : A, x = y
```

Prop vs hProp

```
Definition hProp(A:Type):Type :=
   forall x y : A, x = y
```

- $||A||_{-1}$: make an hProp from any type A, e.g.
  $||A + B||_{-1}$ is irrelevant ‘or’
  $||\{ x : A \& B \}||_{-1}$ is ‘exists’ with irrelevant witness

Prop vs hProp

```
Definition hProp(A:Type):Type :=
   forall x y : A, x = y
```

- $||A||_{-1}$: make an hProp from any type A, e.g.
  $||A + B||_{-1}$ is irrelevant ‘or’
  $||\{ x : A \& B \}||_{-1}$ is ‘exists’ with irrelevant witness

- hProp A is an hProp

Prop vs hProp

```
Definition hProp(A:Type):Type :=
   forall x y : A, x = y
```

- $||A||_{-1}$: make an hProp from any type A, e.g.
  $||A + B||_{-1}$ is irrelevant 'or'
  $||\{ x : A \& B \}||_{-1}$ is 'exists' with irrelevant witness

- hProp A is an hProp

- unique existence: {x:A & P(x)}, where P(x) is an hProp and any two x and y satisfying P are equal

Prop vs hProp

```
Definition hProp(A:Type):Type :=
   forall x y : A, x = y
```

- $||A||_{-1}$: make an hProp from any type A, e.g.
  $||A + B||_{-1}$ is irrelevant ‘or’
  $||\{ x : A \& B \}||_{-1}$ is ‘exists’ with irrelevant witness

- hProp A is an hProp

- unique existence: {x:A & P(x)}, where P(x) is an hProp and any two x and y satisfying P are equal

```
(∃! x:Nat & P(x)) → Nat
```

Prop vs hProp

```
Definition hProp(A:Type):Type :=
   forall x y : A, x = y
```

- $||A||_{-1}$: make an hProp from any type A, e.g.
  $||A + B||_{-1}$ is irrelevant 'or'
  $||\{ x : A \& B \}||_{-1}$ is 'exists' with irrelevant witness

- `hProp A` is an hProp

- unique existence: `{x:A & P(x)}`, where `P(x)` is an hProp and any two x and y satisfying P are equal

```
(∃! x:Nat & P(x)) → Nat
```

not erasable

hSets

```
Definition hSet (A : Type) : Type :=
    forall x y : A,
    forall p q : x = y, p = q
```

hSets

```
Definition hSet (A : Type) : Type :=
   forall x y : A,
   forall p q : x = y, p = q
```

- `forall x:A,B(x)` if A and B are hSets

hSets

```
Definition hSet (A : Type) : Type :=
   forall x y : A,
   forall p q : x = y, p = q
```

- `forall x:A,B(x)` if A and B are hSets
- record whose components are hSets

hSets

```
Definition hSet (A : Type) : Type :=
   forall x y : A,
   forall p q : x = y, p = q
```

- `forall x:A,B(x)` if A and B are hSets
- record whose components are hSets
- `Wx:A.B` where `B(x)` is always an hSet

hSets

```
Definition hSet (A : Type) : Type :=
   forall x y : A,
   forall p q : x = y, p = q
```

- `forall x:A,B(x)` if A and B are hSets
- record whose components are hSets
- `Wx:A.B` where `B(x)` is always an hSet
- `A + B` where A and B are hSets

hSets

```
Definition hSet (A : Type) : Type :=
   forall x y : A,
   forall p q : x = y, p = q
```

- `forall x:A,B(x)` if A and B are hSets
- record whose components are hSets
- `Wx:A.B` where `B(x)` is always an hSet
- `A + B` where A and B are hSets
- therefore `Nat`, lists/trees/… of hSets

hSets

```
Definition hSet (A : Type) : Type :=
   forall x y : A,
   forall p q : x = y, p = q
```

- `forall x:A,B(x)` if A and B are hSets
- record whose components are hSets
- `Wx:A.B` where `B(x)` is always an hSet
- `A + B` where A and B are hSets
- therefore `Nat`, lists/trees/… of hSets

UIP as a type class, closed under all the types you know and love

Truncation levels

```
Definition isTrunc(n:Level)(A:Type):Type :=
    match n with
      -1 => hProp A
   | n+1 => forall x y : A,isTrunc n (x=y)
```

$x : A$

$p : x =_A y$

$q : p_1 =_{x=y} p_2$

$r : q_1 =_{p1=p2} q_2$

$\vdots$

at what level does iterated equality type become trivial?

Univalence

Equivalence

```
Class IsEquiv {A B : Type} (f : A -> B) :=
BuildIsEquiv {
  equiv_inv : B -> A ;
  eisretr : forall x:A, f(equiv_inv x) = x ;
  eissect : forall x:A, equiv_inv(f x) = x ;
  …;
}
```

Equivalence

```
Class IsEquiv {A B : Type} (f : A -> B) :=
BuildIsEquiv {
  equiv_inv : B -> A ;
  eisretr : forall x:A, f(equiv_inv x) = x ;
  eissect : forall x:A, equiv_inv(f x) = x ;
  …;
}
```

make `IsEquiv(f)` into an hProp;
trivial for hSets

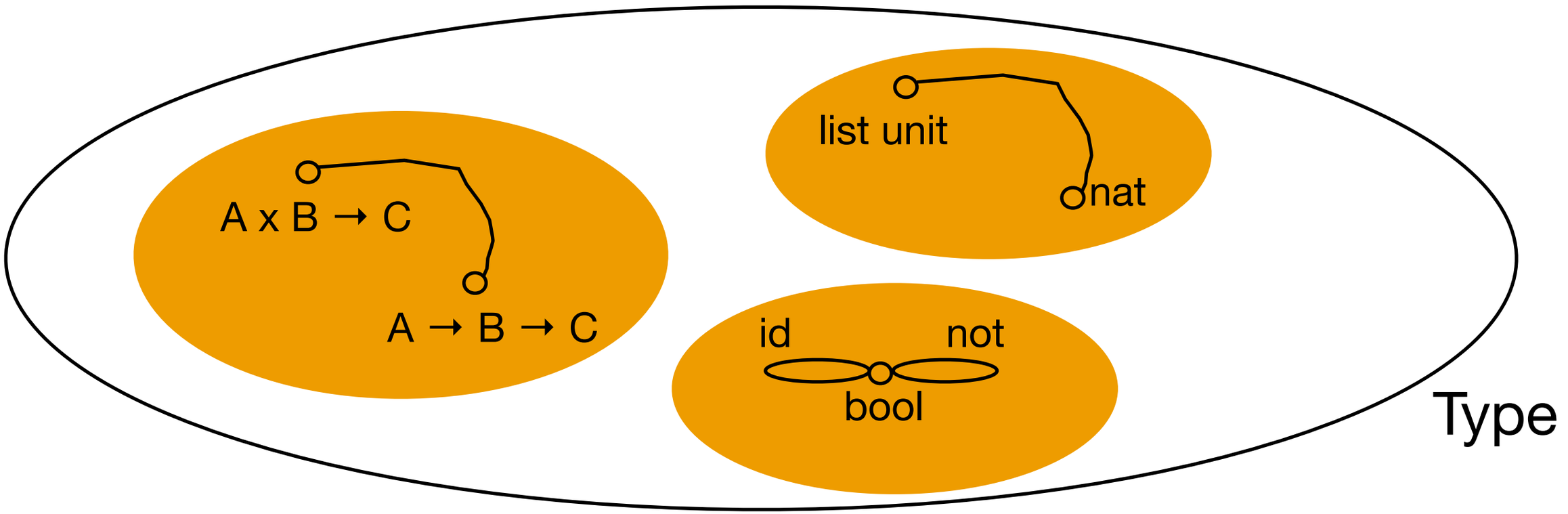
Equivalence

```
Class IsEquiv {A B : Type} (f : A -> B) :=
BuildIsEquiv {
  equiv_inv : B -> A ;
  eisretr : forall x:A, f(equiv_inv x) = x ;
  eissect : forall x:A, equiv_inv(f x) = x ;
  …;
}
```

make `IsEquiv(f)` into an hProp;
trivial for hSets

```
Definition Equiv (A B : Type) :=
  {f : A → B & IsEquiv(f)}
```

Univalence



$A =_{\mathsf{Type}} B$ **is (equivalent to)** $\mathtt{Equiv}\ A\ B$

$\therefore$ all structures/properties respect equivalence

Univalence

1. **Transport**
2. Parametricity reasoning about modules
3. Equality of structures

transport

```
Definition transport
  {A : Type} (C : A → Type)
  {x y : A} (p : x = y) : C x → C y :=
    match p with idpath => (fun u => u) end
```

transport

```
Definition transport
  {A : Type} (C : A → Type)
  {x y : A} (p : x = y) : C x → C y :=
    match p with idpath => (fun u => u) end
```

1. `transport`: any `C:A→Type` respects equality in A *by a function that can potentially do work*

transport

```
Definition transport
  {A : Type} (C : A → Type)
  {x y : A} (p : x = y) : C x → C y :=
    match p with idpath => (fun u => u) end
```

1. `transport`: any `C:A→Type` respects equality in A *by a function that can potentially do work*

2. univalence: equivalence induces equality in Type

transport

```
Definition transport
  {A : Type} (C : A → Type)
  {x y : A} (p : x = y) : C x → C y :=
    match p with idpath => (fun u => u) end
```

1. `transport`: any `C:A→Type` respects equality in A *by a function that can potentially do work*

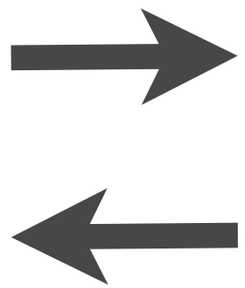
2. univalence: equivalence induces equality in `Type`

3. so any `C : Type → Type` respects equivalence

Example

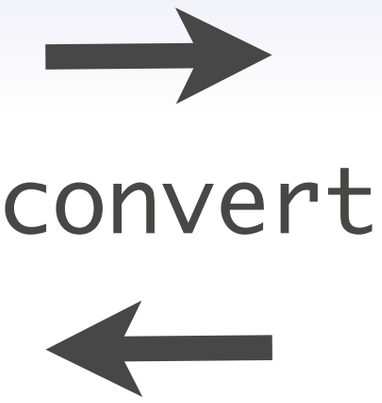
Convert dates between European and US formats, inside a data structure

[{key=4,n="John", d=(30,5,1956)},
{key=8,n="Hugo",d=(29,12,1978)},
{key=15,n="James",d=(1,7,1968)},
{key=16,n="Sayid",d=(2,10,1967)},
{key=23,n="Jack",d=(3,12,1969)},
{key=42,n="Sun",d=(20,3,1980)}]



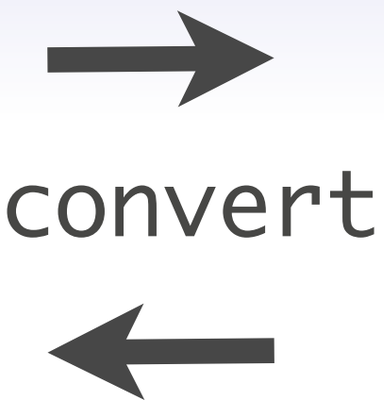
[{key=4,n="John",d=(5,30,1956)},
{key=8,n="Hugo",d=(12,29,1978)},
{key=15,n="James",d=(7,1,1968)},
{key=16,n="Sayid",d=(10,2,1967)},
{key=23,n="Jack",d=(12,3,1969)},
{key=42,n="Sun",d=(3,20,1980)}]

[{key=4,n="John", d=(30,5,1956)},
{key=8,n="Hugo",d=(29,12,1978)},
{key=15,n="James",d=(1,7,1968)},
{key=16,n="Sayid",d=(2,10,1967)},
{key=23,n="Jack",d=(3,12,1969)},
{key=42,n="Sun",d=(20,3,1980)}]



[{key=4,n="John",d=(5,30,1956)},
{key=8,n="Hugo",d=(12,29,1978)},
{key=15,n="James",d=(7,1,1968)},
{key=16,n="Sayid",d=(10,2,1967)},
{key=23,n="Jack",d=(12,3,1969)},
{key=42,n="Sun",d=(3,20,1980)}]

[{key=4,n="John", d=(30,5,1956)},
{key=8,n="Hugo",d=(29,12,1978)},
{key=15,n="James",d=(1,7,1968)},
{key=16,n="Sayid",d=(2,10,1967)},
{key=23,n="Jack",d=(3,12,1969)},
{key=42,n="Sun",d=(20,3,1980)}]

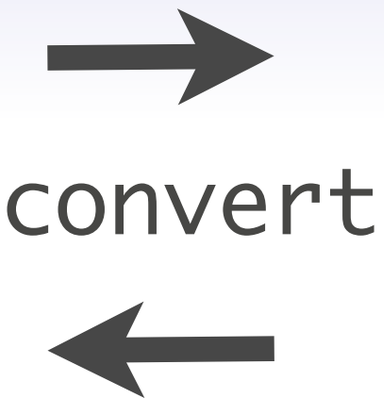


[{key=4,n="John",d=(5,30,1956)},
{key=8,n="Hugo",d=(12,29,1978)},
{key=15,n="James",d=(7,1,1968)},
{key=16,n="Sayid",d=(10,2,1967)},
{key=23,n="Jack",d=(12,3,1969)},
{key=42,n="Sun",d=(3,20,1980)}]

1. Define a function

```
swapfn(x,y) := (y,x)
```

[{key=4,n="John", d=(30,5,1956)},
{key=8,n="Hugo",d=(29,12,1978)},
{key=15,n="James",d=(1,7,1968)},
{key=16,n="Sayid",d=(2,10,1967)},
{key=23,n="Jack",d=(3,12,1969)},
{key=42,n="Sun",d=(20,3,1980)}]



[{key=4,n="John",d=(5,30,1956)},
{key=8,n="Hugo",d=(12,29,1978)},
{key=15,n="James",d=(7,1,1968)},
{key=16,n="Sayid",d=(10,2,1967)},
{key=23,n="Jack",d=(12,3,1969)},
{key=42,n="Sun",d=(3,20,1980)}]

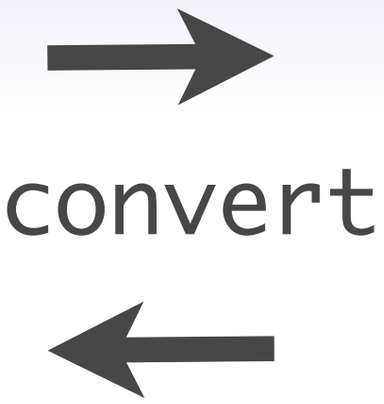
1. Define a function
   `swapfn(x,y) := (y,x)`
2. Define `swap : (A × B) = (B × A) :=`
   `ua(swapfn,swapfn,self-inv)`

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]

[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

1. Define a function
   `swapfn(x,y) := (y,x)`
2. Define `swap : (A × B) = (B × A) :=`
   `ua(swapfn,swapfn,self-inv)`
3. Define a type family describing where to swap:
   `There(X)=List{key:int, n:string, d:X×int}`

[{key=4,n="John", d=(30,5,1956)},
{key=8,n="Hugo",d=(29,12,1978)},
{key=15,n="James",d=(1,7,1968)},
{key=16,n="Sayid",d=(2,10,1967)},
{key=23,n="Jack",d=(3,12,1969)},
{key=42,n="Sun",d=(20,3,1980)}]



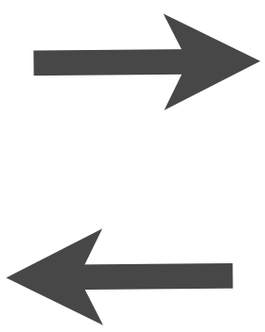
[{key=4,n="John",d=(5,30,1956)},
{key=8,n="Hugo",d=(12,29,1978)},
{key=15,n="James",d=(7,1,1968)},
{key=16,n="Sayid",d=(10,2,1967)},
{key=23,n="Jack",d=(12,3,1969)},
{key=42,n="Sun",d=(3,20,1980)}]

1. Define a function
   `swapfn(x,y) := (y,x)`
2. Define `swap : (A × B) = (B × A) :=`
   `ua(swapfn,swapfn,self-inv)`
3. Define a type family describing where to swap:
   `There(X)=List{key:int, n:string, d:X×int}`
4. Define
   `convert(db) := transport`$_{\texttt{There}}$`(swap,db)`

Computational interpretation of transport:

```
There(X)=List{key:int, n:string, d:X×int}

   transport_There(swap,db)
```

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]

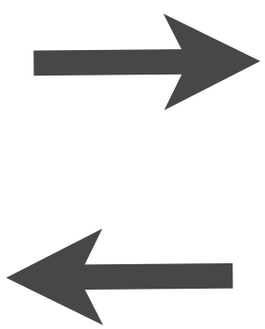


[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

Computational interpretation of transport:

```
There(X)=List{key:int, n:string, d:X×int}

   transport_There(swap,db)
=  List.map (transport_There1 swap) db
```

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]

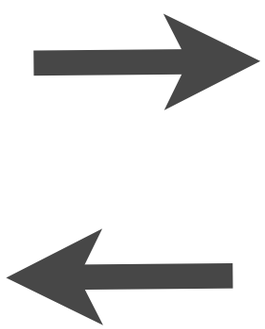


[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

Computational interpretation of transport:

```
There1(X)={key:int, n:string, d:X×int}

   transportThere(swap,db)
=  List.map (transportThere1 swap) db
```

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]

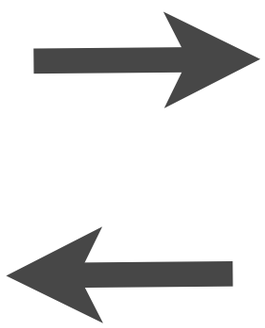


[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

Computational interpretation of transport:

```
There1(X)={key:int, n:string, d:X×int}

   transport_There(swap,db)
=  List.map (transport_There1 swap) db
=  List.map ({key,n,(d,m,y)} =>
    {key,n, (                    ,y)}) db
```

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]

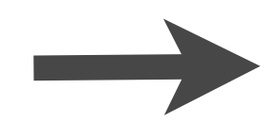


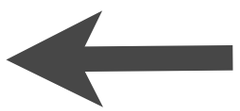
[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

Computational interpretation of transport:

```
There1(X)={key:int, n:string, d:X×int}

   transportThere(swap,db)
=  List.map (transportThere1 swap) db
=  List.map ({key,n,(d,m,y)} =>
     {key,n, (transptHere(swap,(d,m)),y)}) db
```

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]



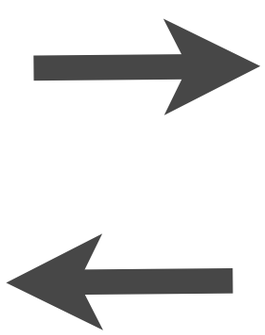


[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

Computational interpretation of transport:

```
Here(X)=X

   transportThere(swap,db)
=  List.map (transportThere1 swap) db
=  List.map ({key,n,(d,m,y)} =>
     {key,n, (transptHere(swap,(d,m)),y)}) db
```

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]

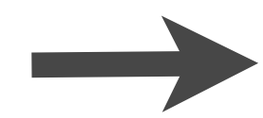


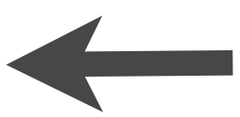
[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

Computational interpretation of transport:

```
Here(X)=X

   transportThere(swap,db)
=  List.map (transportThere1 swap) db
=  List.map ({key,n,(d,m,y)} =>
    {key,n, (transptHere(swap,(d,m)),y)}) db
=  List.map ({key,n,(d,m,y)} =>
             {key,n,(m,d,y)}) db
```

[{key=4,n="John", d=(30,5,1956)},
 {key=8,n="Hugo",d=(29,12,1978)},
 {key=15,n="James",d=(1,7,1968)},
 {key=16,n="Sayid",d=(2,10,1967)},
 {key=23,n="Jack",d=(3,12,1969)},
 {key=42,n="Sun",d=(20,3,1980)}]





[{key=4,n="John",d=(5,30,1956)},
 {key=8,n="Hugo",d=(12,29,1978)},
 {key=15,n="James",d=(7,1,1968)},
 {key=16,n="Sayid",d=(10,2,1967)},
 {key=23,n="Jack",d=(12,3,1969)},
 {key=42,n="Sun",d=(3,20,1980)}]

```
Class Monoid (m : Type) := {
  mempty : m ;
  mappend : m -> m -> m ;
  mappend_mempty_left :>
    left_neutral mappend mempty ;
  mappend_mempty_right :>
    right_neutral mappend mempty ;
  mappend_assoc :> associative mappend
}
```

```
Class Monoid (m : Type) := {
  mempty : m ;
  mappend : m -> m -> m ;
  mappend_mempty_left :>
    left_neutral mappend mempty ;
  mappend_mempty_right :>
    right_neutral mappend mempty ;
  mappend_assoc :> associative mappend
}
```

`transport`$_{\texttt{Monoid}}$: write a monoid structure on B given a `Monoid A` and an equivalence between A and B

Univalence

1. Transport

2. Parametricity reasoning about modules

3. Equality of structures

Parametricity

```
Record Dict := BuildDict {
  dict : hSet ;
  insert : dict → (key × value) → dict;
  … }
```

$D1 =_{Dict} D2$ there is a relation between `dict(D1)` and `dict(D2)` that is preserved by the operations

Parametricity

```
Record Dict := BuildDict {
  dict : hSet ;
  insert : dict → (key × value) → dict;
  … }
```

$D1 =_{Dict} D2$ there is a between `dict(D1)` and `dict(D2)` that is preserved by the operations

Parametricity

```
Record Dict := BuildDict {
  dict : hSet ;
  insert : dict → (key × value) → dict;
  … }
```

$D1 =_{Dict} D2$ there is a bijection between `dict(D1)` and `dict(D2)` that is preserved by the operations

Parametricity

```
AssocList : Dict        RedBlackTree : Dict
```

Parametricity

```
AssocList : Dict        RedBlackTree : Dict
```

reasoning

Parametricity

```
AssocList : Dict
```

reasoning

```
RedBlackTree : Dict
```

fast

Parametricity

```
AssocList : Dict        RedBlackTree : Dict
```

reasoning

fast

```
Client(D : Dict)
```

Parametricity

`AssocList : Dict`

reasoning

`RedBlackTree : Dict`

fast

`Client(D : Dict)`

1. Give a bijection between `dict(RedBlackTree)` and `dict(AssocList)` that is preserved by the operations

Parametricity

`AssocList : Dict`

reasoning

`RedBlackTree : Dict`

fast

`Client(D : Dict)`

1. Give a bijection between
   `dict(RedBlackTree)` and `dict(AssocList)`
   that is preserved by the operations
2. For any correctness spec P,
   to prove `P(Client(RedBlackTrees))`
   suffices to show `P(Client(AssocList))`

Univalence

1. Transport
2. Parametricity reasoning about modules
3. **Equality of structures**

Equality of structures

- Two groups are equal iff there is a group isomorphism between them
- Two categories are equal iff they are equivalent
- Two functors are equal iff they are naturally isomorphic
- …

Univalence

1. Transport
2. Parametricity reasoning about modules
3. Equality of structures

Higher inductive types

Higher inductive types

1. **Quotient types**
2. Spaces
3. Programming applications

Quotients

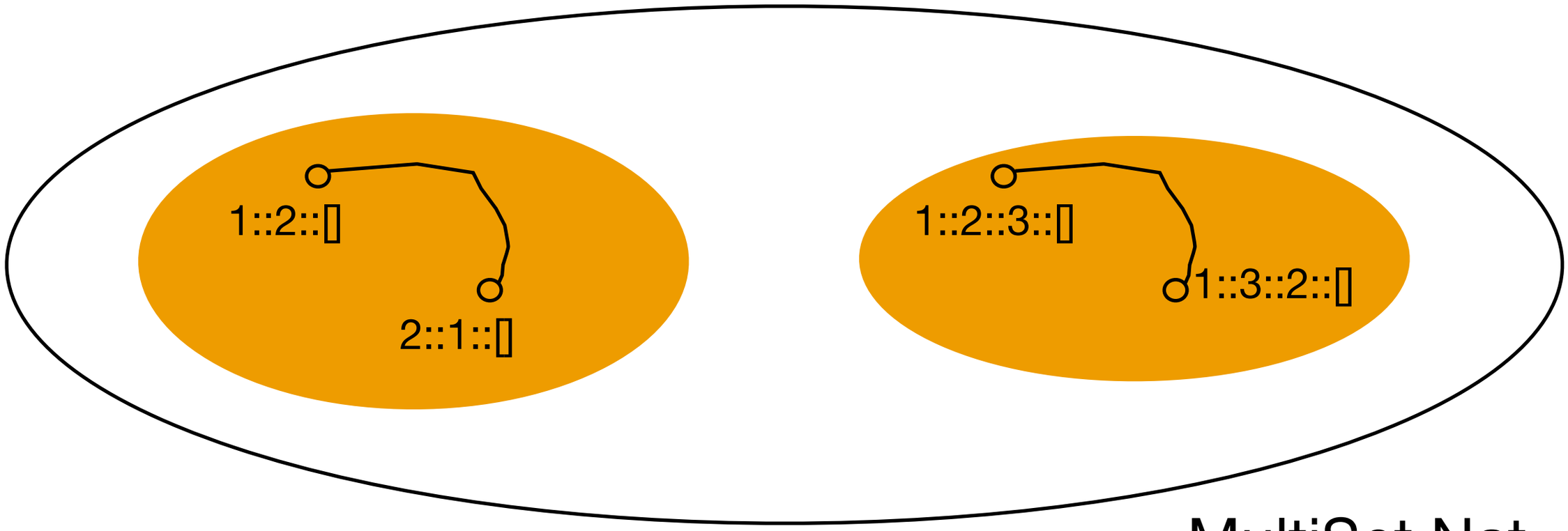
```
HigherInductive MultiSet(A:hSet) : hSet :=
  [] : MultiSet A
| :: : A → MultiSet A → MultiSet A
| ex : forall x y : A, forall xs:MultiSet A,
         x :: y :: xs = y :: x :: xs
```

Quotients

```
HigherInductive MultiSet(A:hSet) : hSet :=
  [] : MultiSet A
| :: : A → MultiSet A → MultiSet A
| ex : forall x y : A, forall xs:MultiSet A,
       x :: y :: xs = y :: x :: xs
```

path constructor

Quotients



Functions act on paths

```
Definition ap
  {A B : Type} (f : A → B)
  {x y : A} (p : x = y) : f x = f y :=
    match p with idpath => idpath end
```

all functions take equals to equals

Functions act on paths

For any `f : MultiSet Nat → X`

```
ap f (ex 1 2 (3::[])) :
  f [1,2,3] = f [2,1,3]

ap (fun y => f(1::y)) (ex 2 3 []) :
  f [1,2,3] = f [1,3,2]
```

Quotients

```
Fixpoint append {A:Type}
(xs : MultiSet A) (ys : MultiSet A) : MultiSet A
:=

match xs with

  [] => ys

| x :: xs => x :: append xs ys
| ex x y xs => ex x y (append xs ys)
```

Quotients

```
Fixpoint append {A:Type}
(xs : MultiSet A) (ys : MultiSet A) : MultiSet A
:=

match xs with

  [] => ys

| x :: xs => x :: append xs ys
| ex x y xs => ex x y (append xs ys)
```

case for path constructor

Quotients

```
Fixpoint append {A:Type}
(xs : MultiSet A) (ys : MultiSet A) : MultiSet A
:=

match xs with

   [] => ys

| x :: xs => x :: append xs ys
| ex x y xs => ex x y (append xs ys)
```

case for path constructor

Need to show:

x :: y :: append xs ys =

y :: x :: append xs ys

Quotients

```
MS_rec {A B : Type}
       (n : A)
       (c : A → MultiSet A → B → B)
       (e : forall x y, xs, b : B,
              c x (y :: xs) (c y xs b)
                = c y (x :: xs) (c x xs b))
       : MultiSet A → B

apppend xs ys :=
  MS-rec ys (fun x _ xs' => x :: xs')
            (fun x y _ xs' =>
              ex x y xs')
```

Quotients

```
MS_rec {A B : Type}
       (n : A)
       (c : A → MultiSet A → B → B)
       (e : forall x y, xs, b : B,
              c x (y :: xs) (c y xs b)
                = c y (x :: xs) (c x xs b))
       : MultiSet A → B

apppend xs ys :=
  MS-rec ys (fun x _ xs' => x :: xs')
            (fun x y _ xs' =>
              ex x y xs')
```

Need to show:

x :: y :: xs' = y :: x :: xs'

Quotients

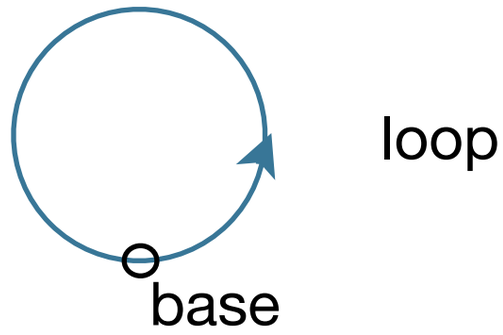
- Free congruence given by some point generators ([],::) and path generators (ex).
- Can use this to define general quotient type A / R where R : A → A → hProp is an equivalence relation
- No more setoids!

Higher inductive types

1. Quotient types
2. **Spaces**
3. Programming applications

The circle

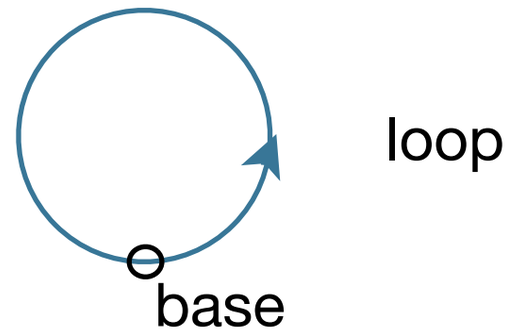
Circle S^1 is a **higher inductive type** generated by



The circle

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

```
base : S¹
loop : base = base
```

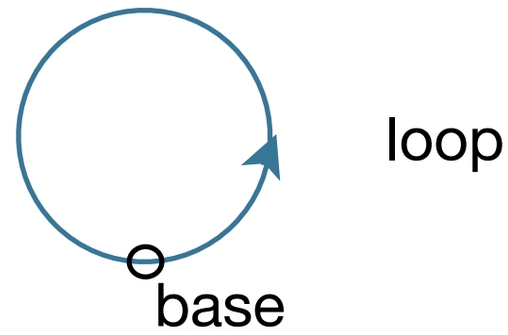


The circle

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

point `base : S¹`

`loop : base = base`

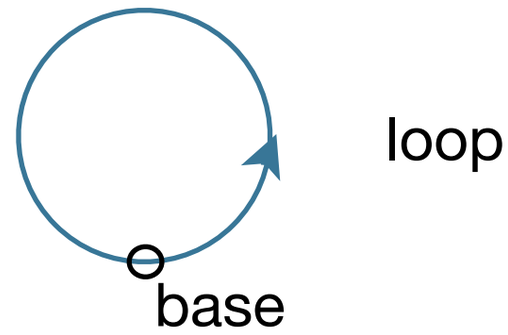


The circle

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

point `base : S¹`

path `loop : base = base`

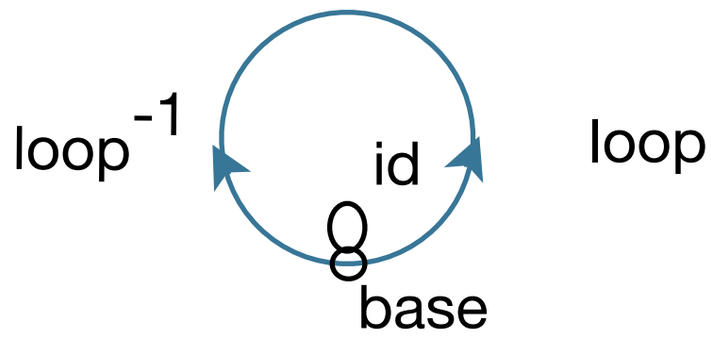


The circle

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`

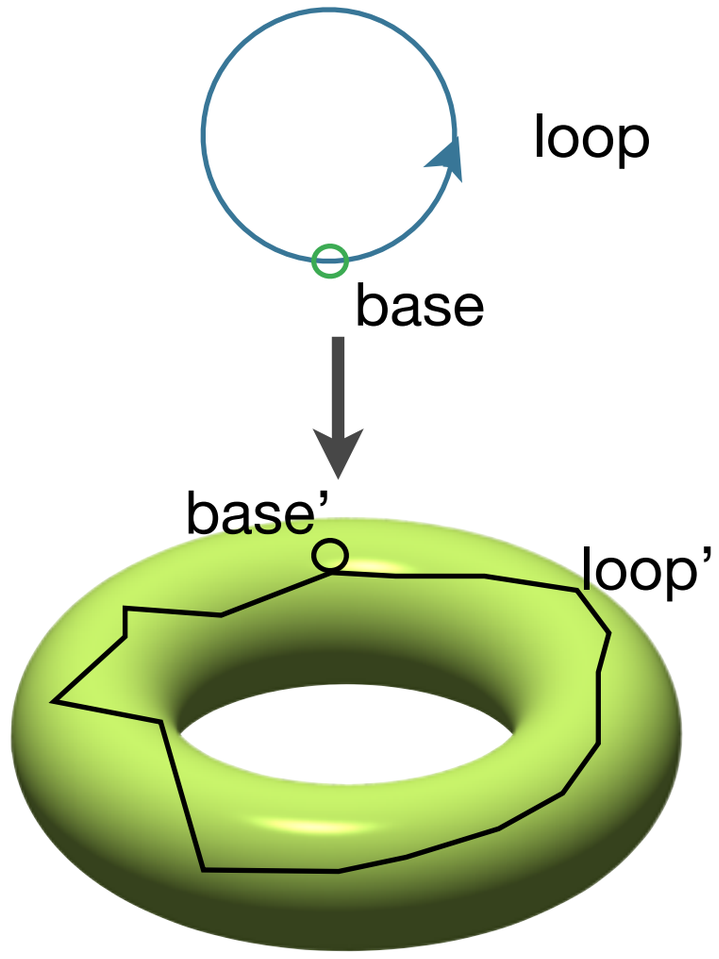
`...`

The circle

Circle recursion:

function $S^1 \to X$ determined by

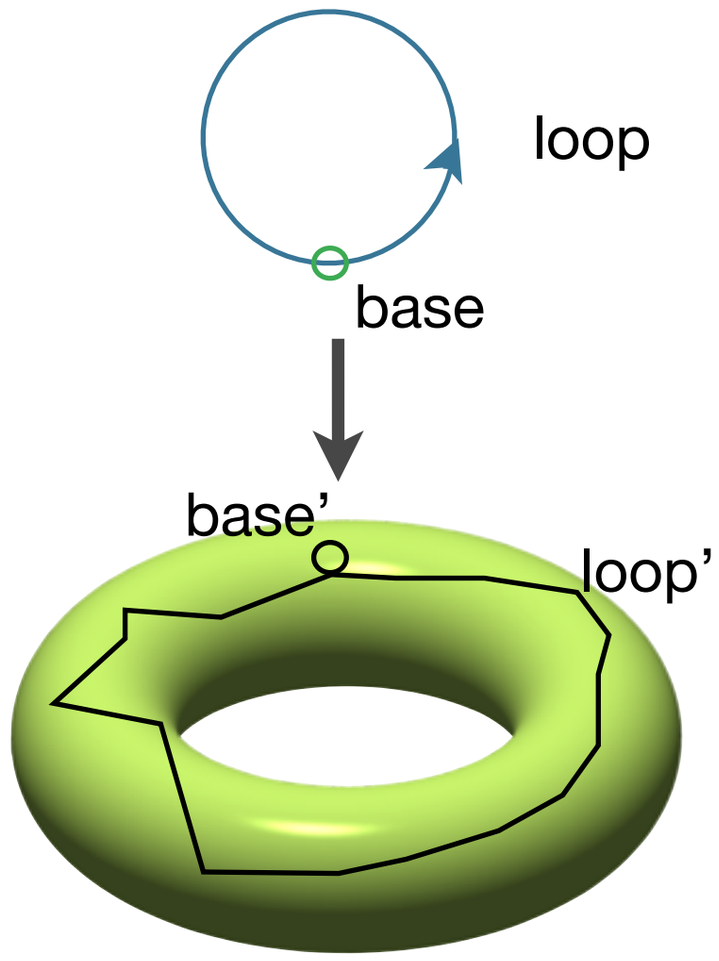
```
base' : X
loop' : base' = base'
```



The circle

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

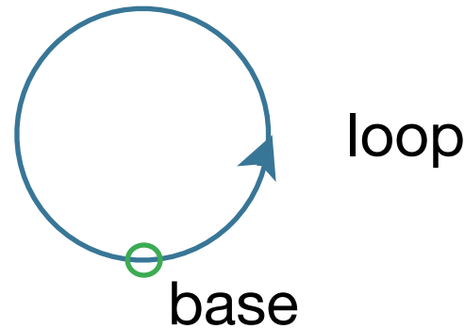
```
base' : X
loop' : base' = base'
```



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

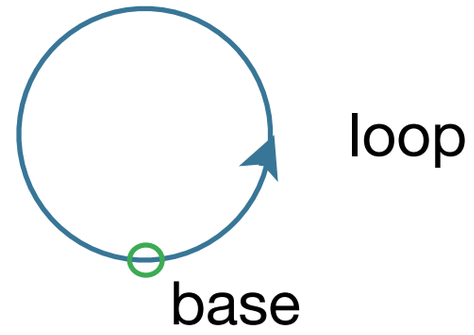
Fundamental group of circle

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



Fundamental group of circle

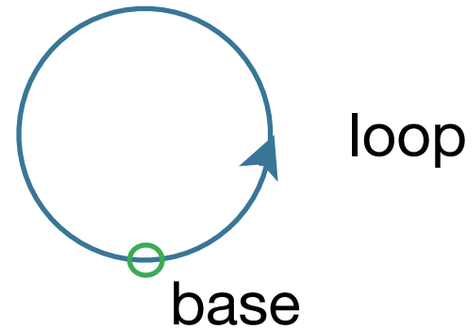
How many different loops are there on the circle, up to homotopy?




```
id
```

Fundamental group of circle

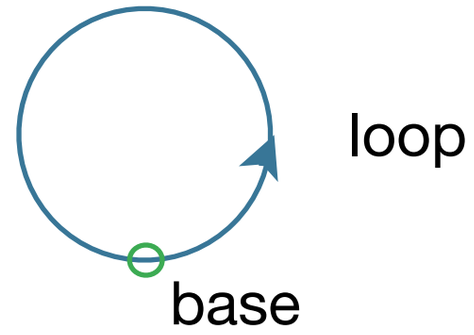
How many different loops are there on the circle, up to homotopy?




```
id
loop
```

Fundamental group of circle

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



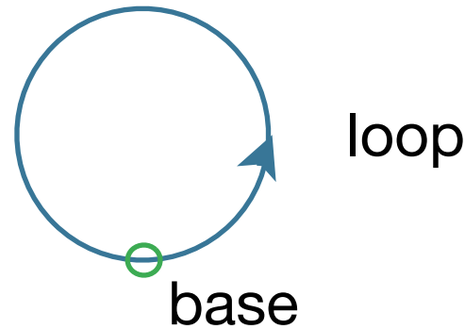
`id`

`loop`

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

Fundamental group of circle

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



`id`

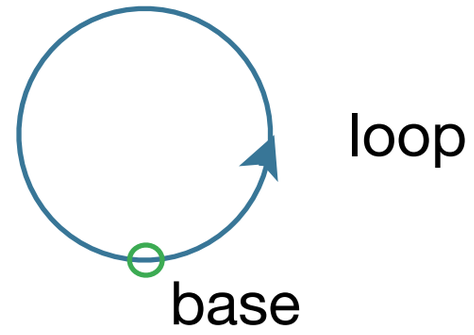
`loop`

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

`loop o loop`

Fundamental group of circle

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



`id`

`loop`

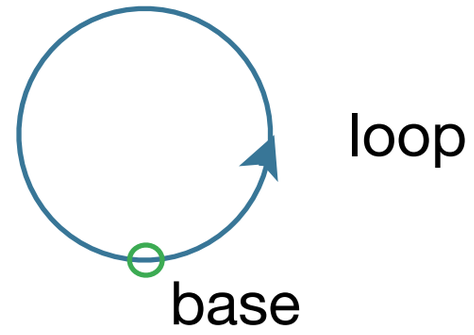
$\texttt{loop}^{-1}$

`loop o loop`

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

Fundamental group of circle

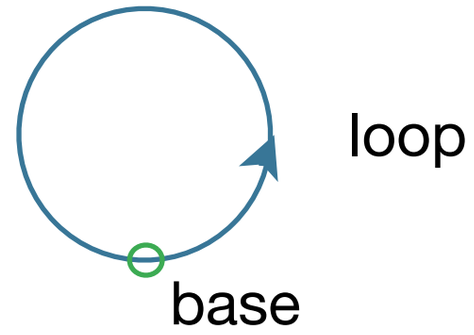
How many different loops are there on the circle, up to homotopy?




```
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`

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

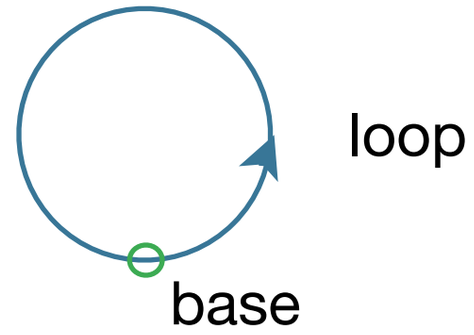
`loop o loop`

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

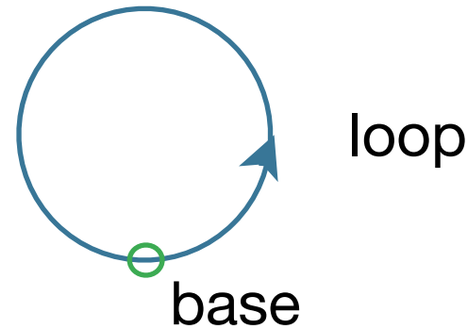
`loop o loop`

`loop⁻¹ o loop⁻¹`

`loop o loop⁻¹ = id`

Fundamental group of circle

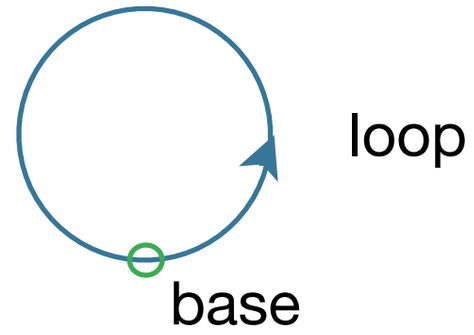
How many different loops are there on the circle, up to homotopy?



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

Fundamental group of circle

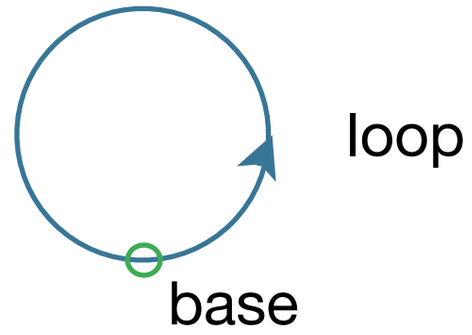
How many different loops are there on the circle, up to homotopy?



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

Fundamental group of circle

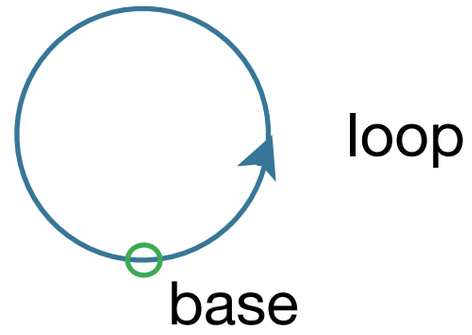
How many different loops are there on the circle, up to homotopy?



`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?

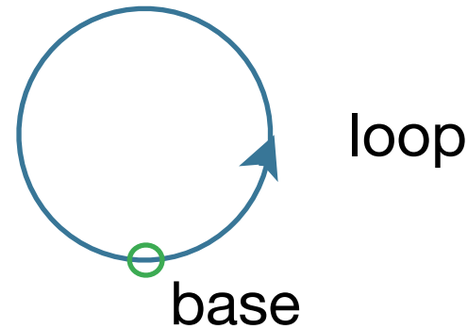


`id`	`0`
`loop`	`1`
`loop⁻¹`	`-1`
`loop o loop`	`2`
`loop⁻¹ o loop⁻¹`	`-2`

`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`

Homotopy groups of spheres

k^{th} homotopy group

n-dimensional sphere

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}
S^0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S^1	$\mathbb{Z}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S^2	0	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$
S^3	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$
S^4	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}\times\mathbb{Z}_{12}$	$\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{24}\times\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^3$	$\mathbb{Z}_{120}\times\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^5$
S^5	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{30}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^3$	$\mathbb{Z}_{72}\times\mathbb{Z}_2$
S^6	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_{60}$	$\mathbb{Z}_{24}\times\mathbb{Z}_2$	$\mathbb{Z}_2^3$
S^7	0	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	0	$\mathbb{Z}_2$	$\mathbb{Z}_{120}$	$\mathbb{Z}_2^3$
S^8	0	0	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	0	$\mathbb{Z}_2$	$\mathbb{Z}\times\mathbb{Z}_{120}$

[image from wikipedia]

Homotopy groups of spheres

k^{th} homotopy group

n-dimensional sphere

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}
S^0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S^1	$\mathbb{Z}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S^2	0	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$
S^3	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{12}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_3$	$\mathbb{Z}_{15}$	$\mathbb{Z}_2$	$\mathbb{Z}_2^2$	$\mathbb{Z}_{12}\times\mathbb{Z}_2$	$\mathbb{Z}_{84}\times\mathbb{Z}_2^2$	$\mathbb{Z}_2^2$
S^4	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$									
S^5	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$							
S^6	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0					
S^7	0	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	0			
S^8	0	0	0	0	0	0	0	$\mathbb{Z}$	$\mathbb{Z}_2$	$\mathbb{Z}_2$	$\mathbb{Z}_{24}$	0	0	$\mathbb{Z}_2$	

[image from wikipedia]

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)

Blakers-Massey

$T^2 = S^1 \times S^1$

Van Kampen

Covering spaces

Whitehead for n-types

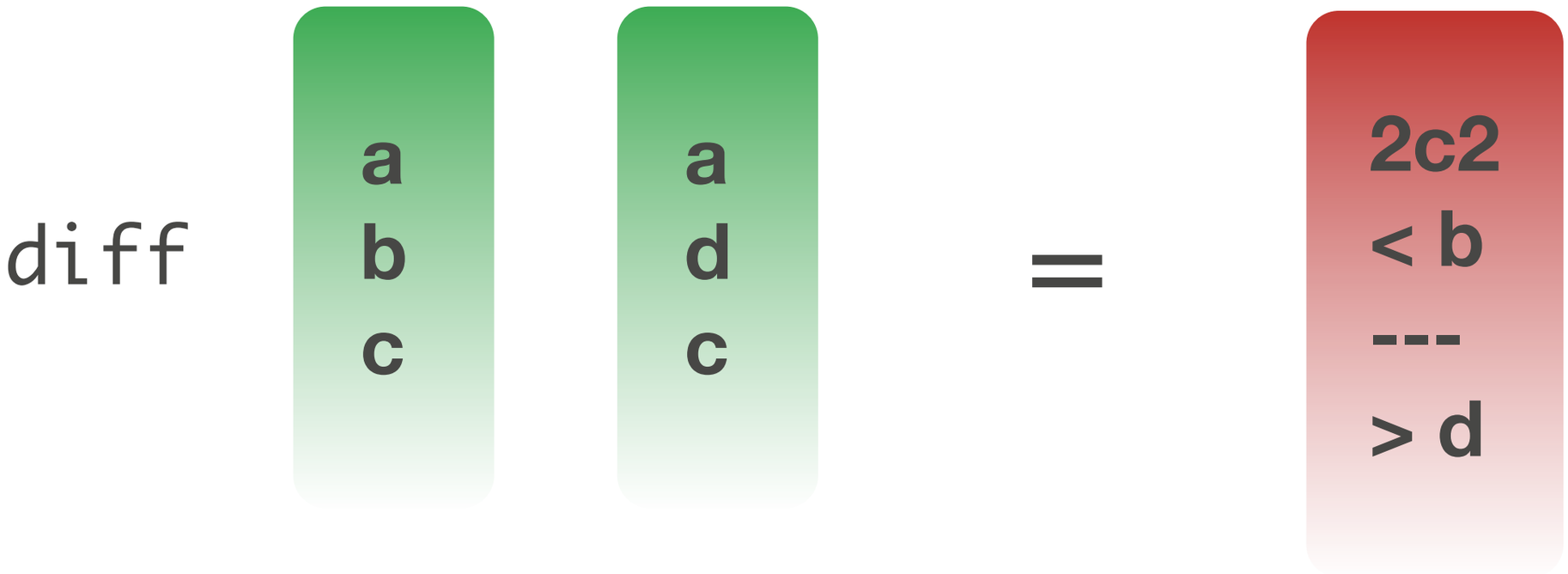
Cohomology axioms

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

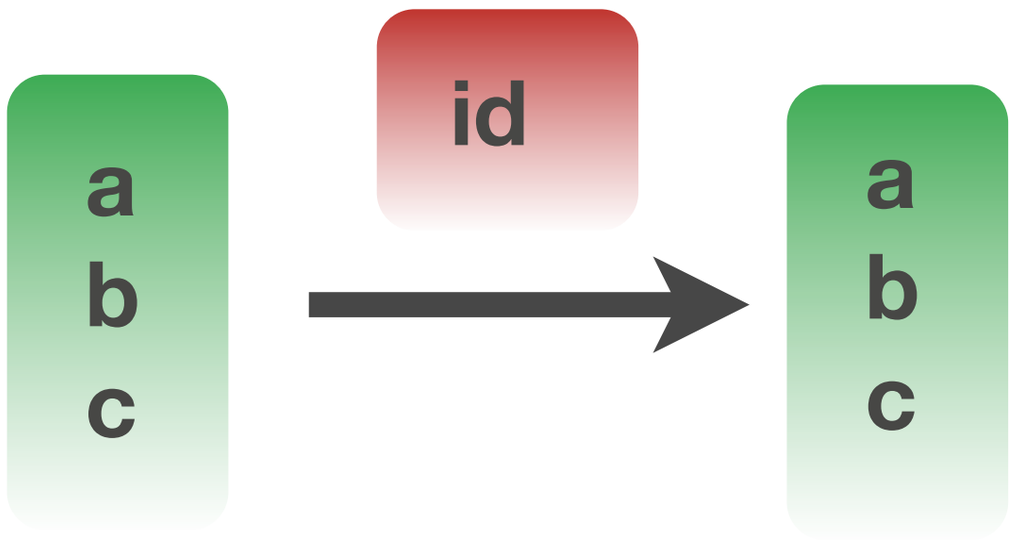
Higher inductive types

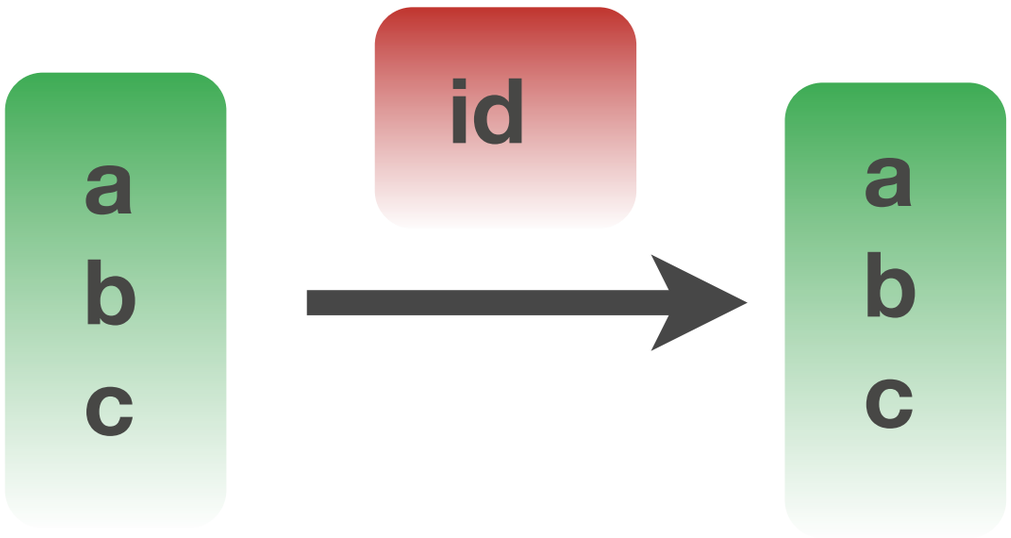
1. Quotient types
2. Spaces
3. **Programming applications**

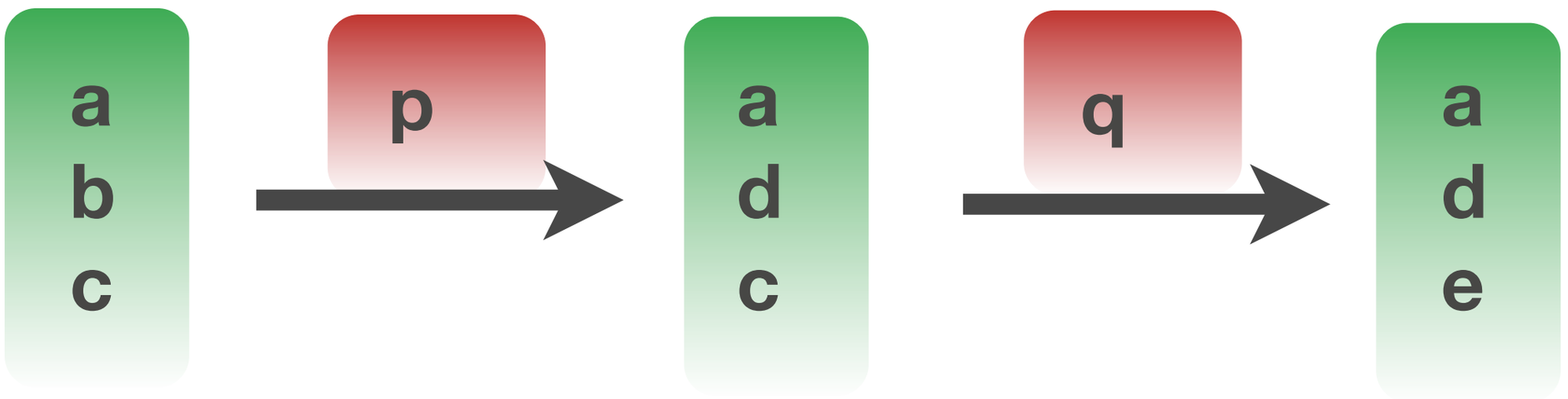
Patches

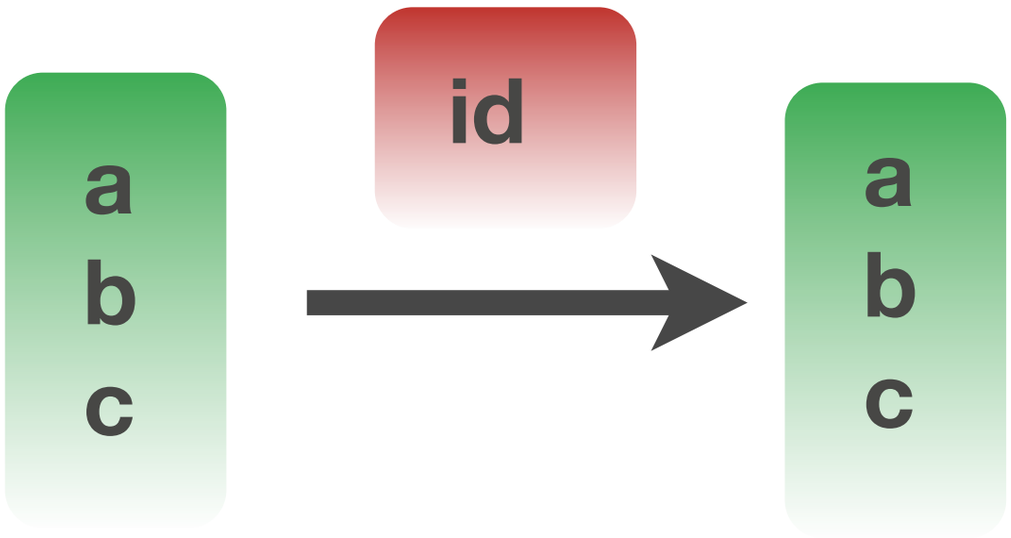


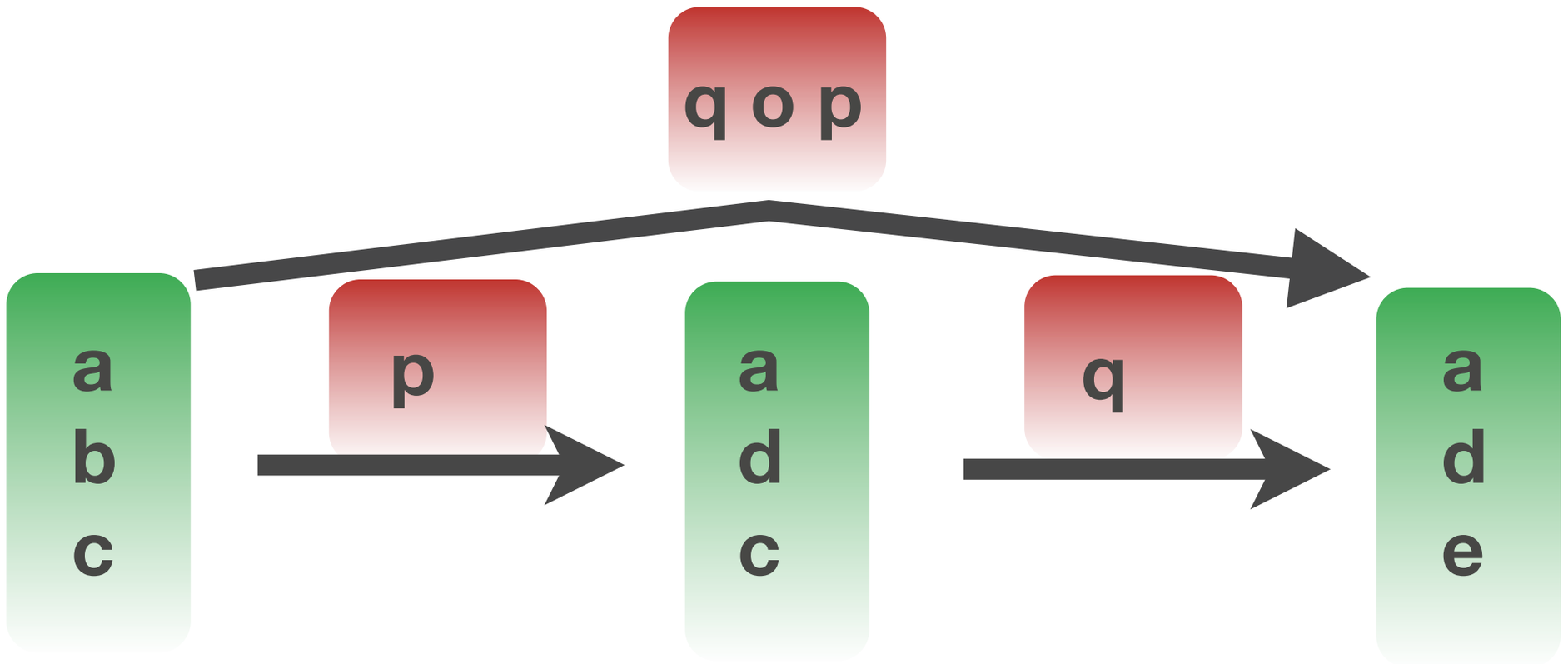
- Version control
- Collaborative editing

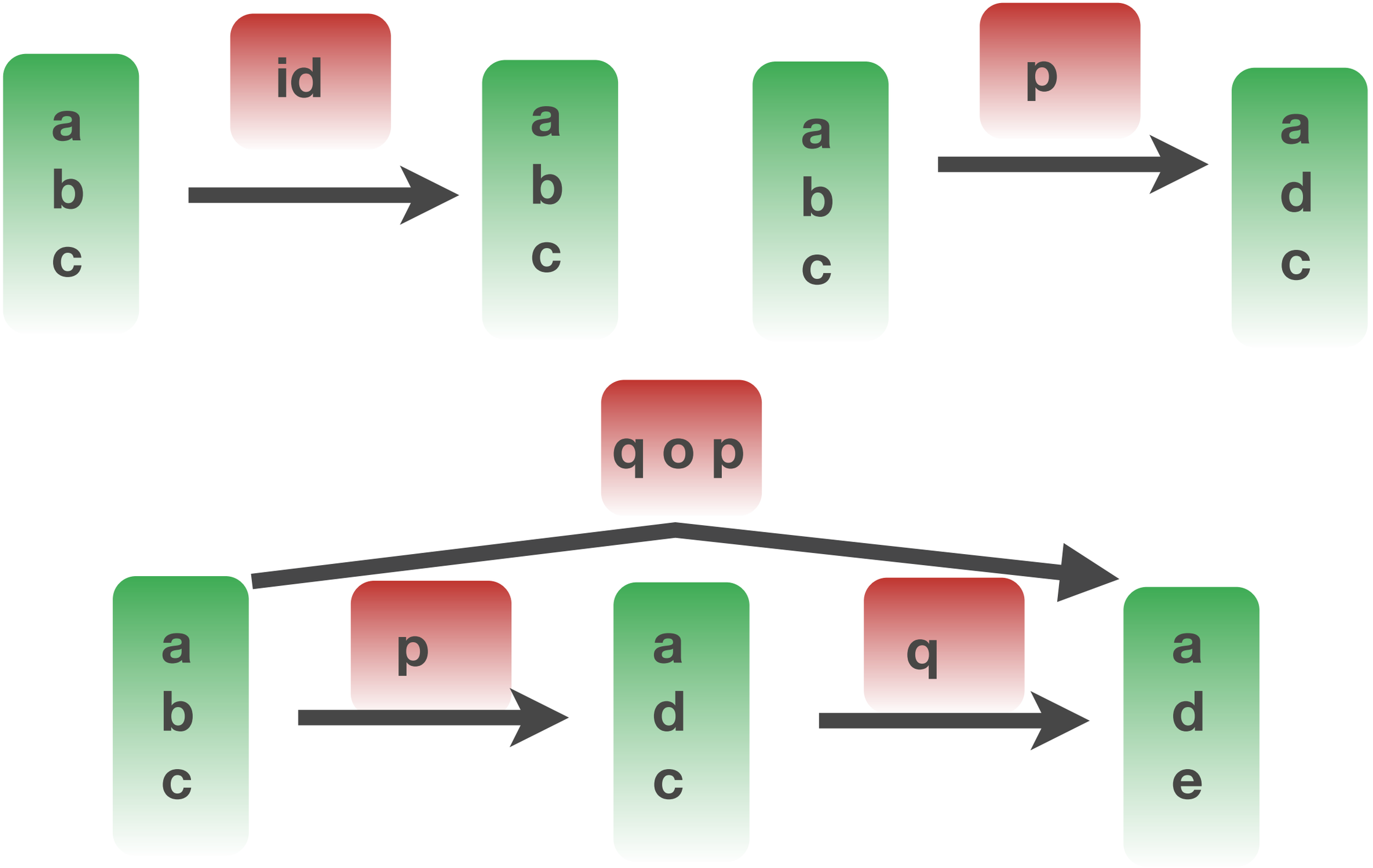

a
b
c
id
a
b
c

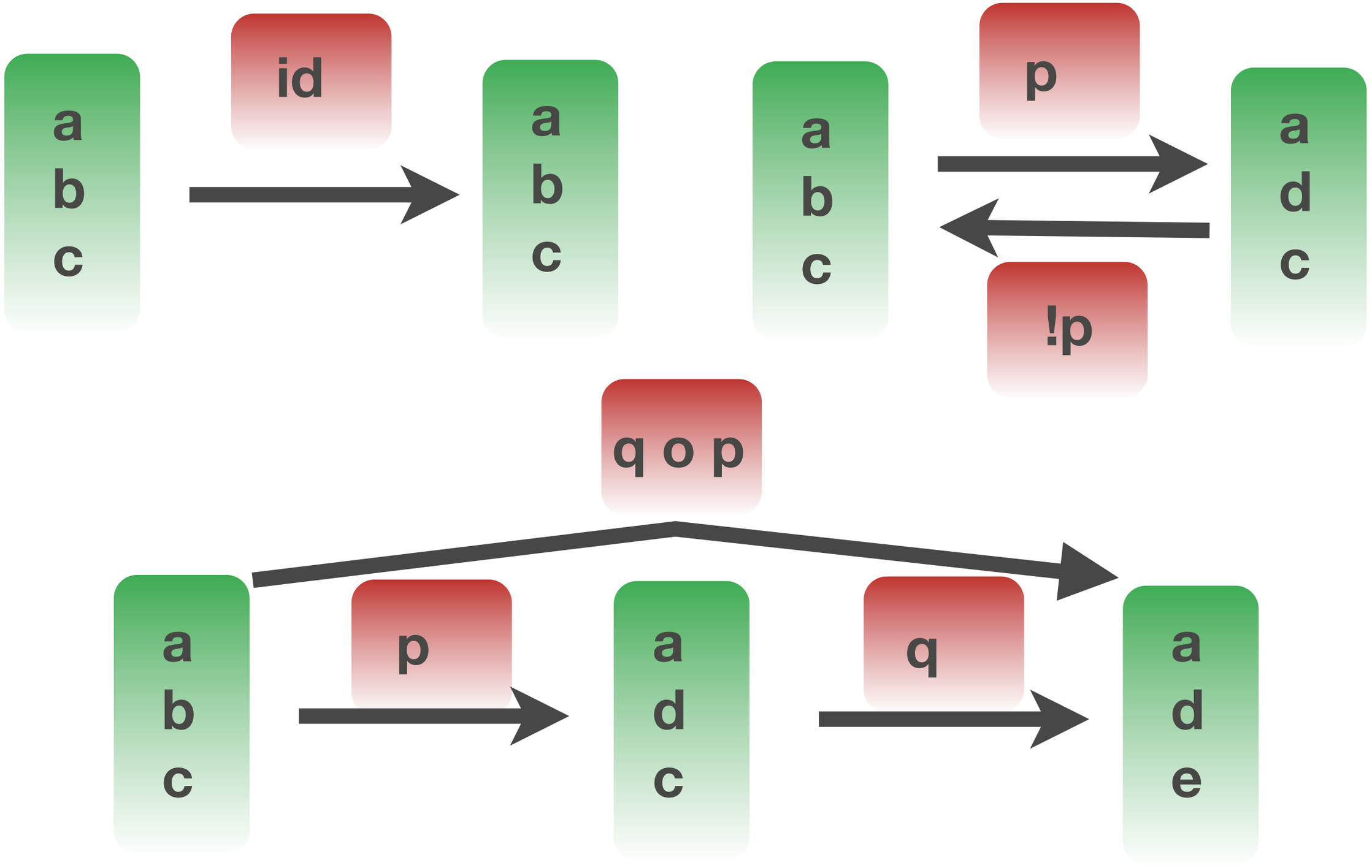

a
b
c
id
a
b
c

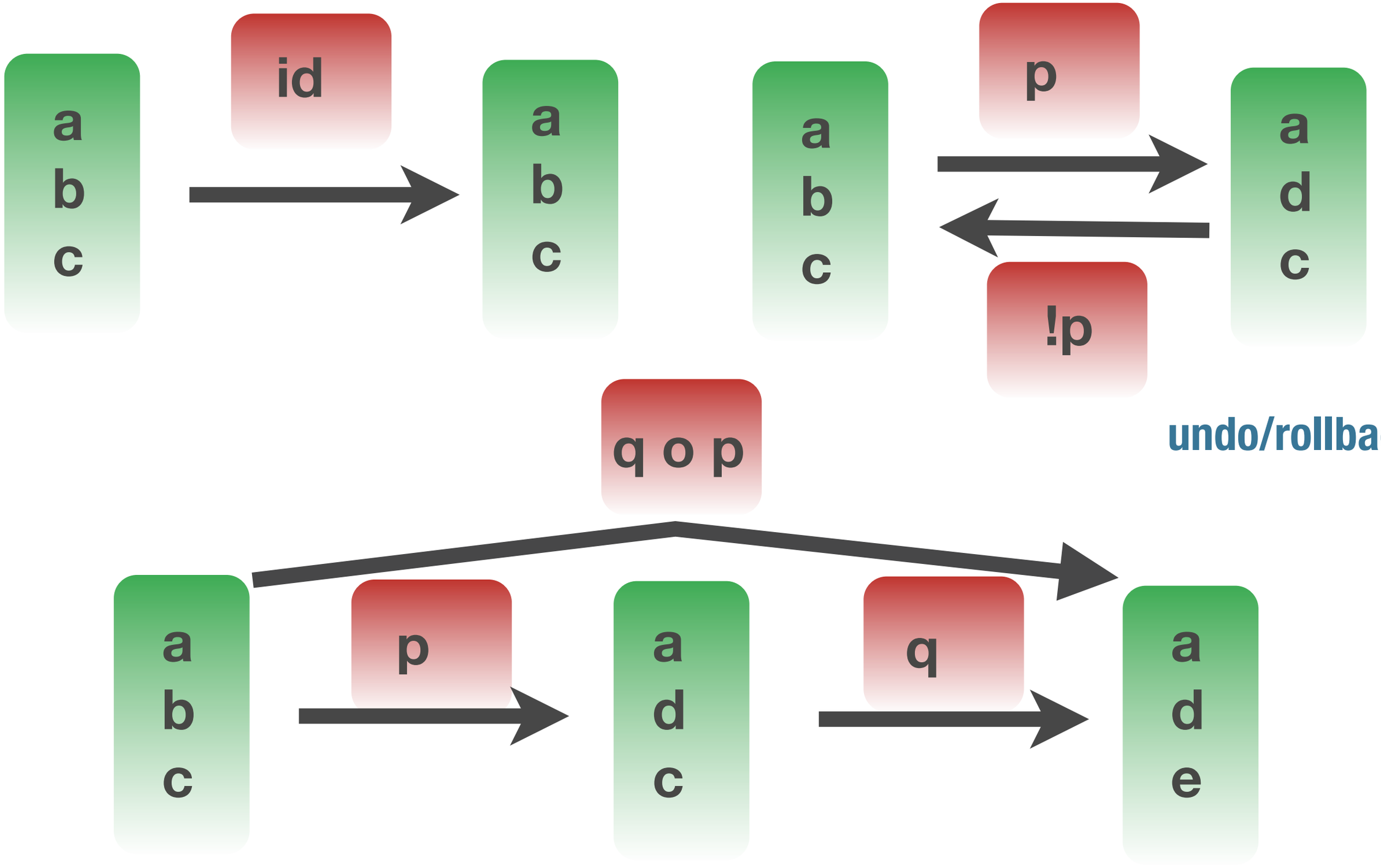

a
b
c
p
a
d
c
q
a
d
e


a
b
c
id
a
b
c

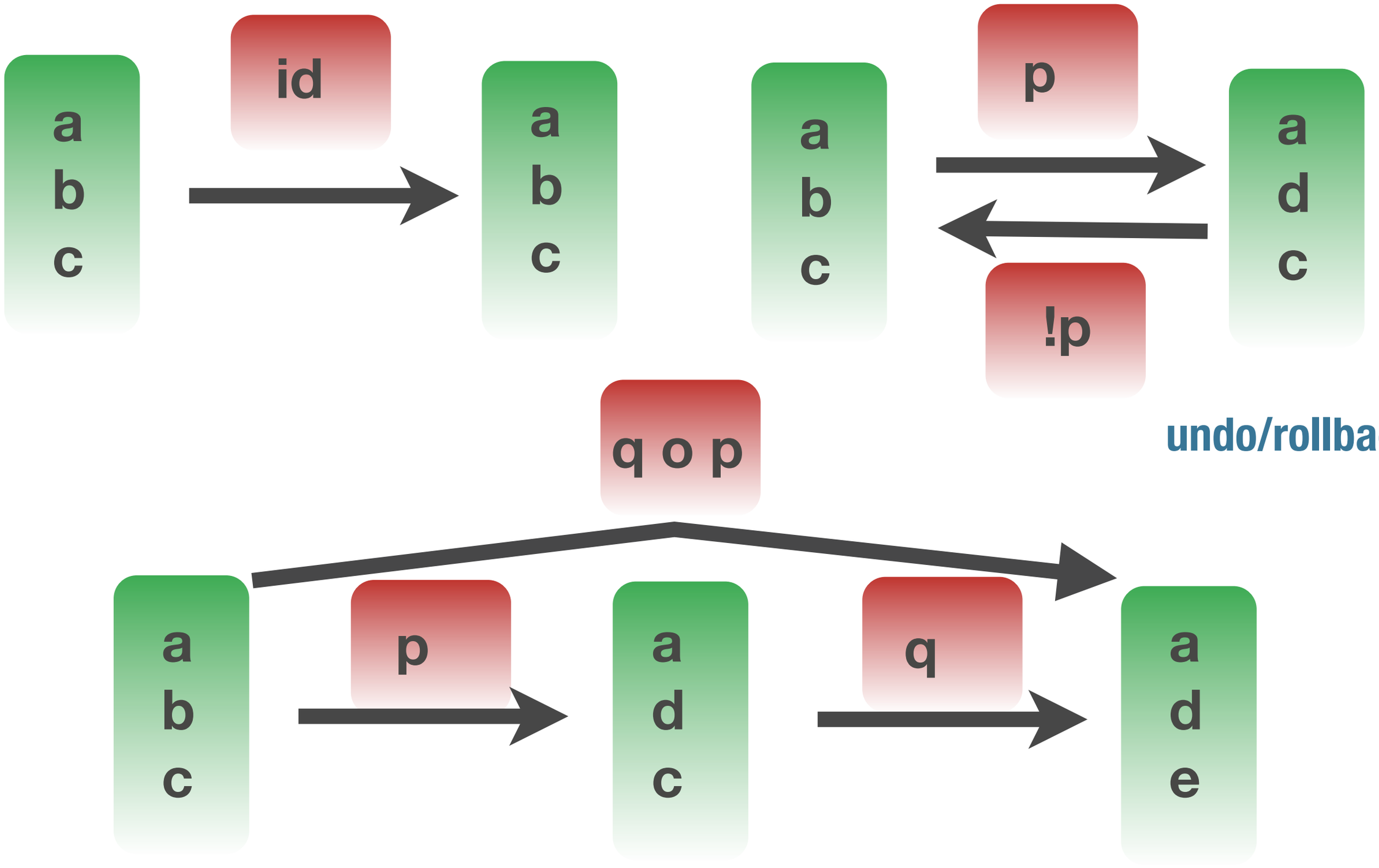

q o p
a
b
c
p
a
d
c
q
a
d
e


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



A patch theory

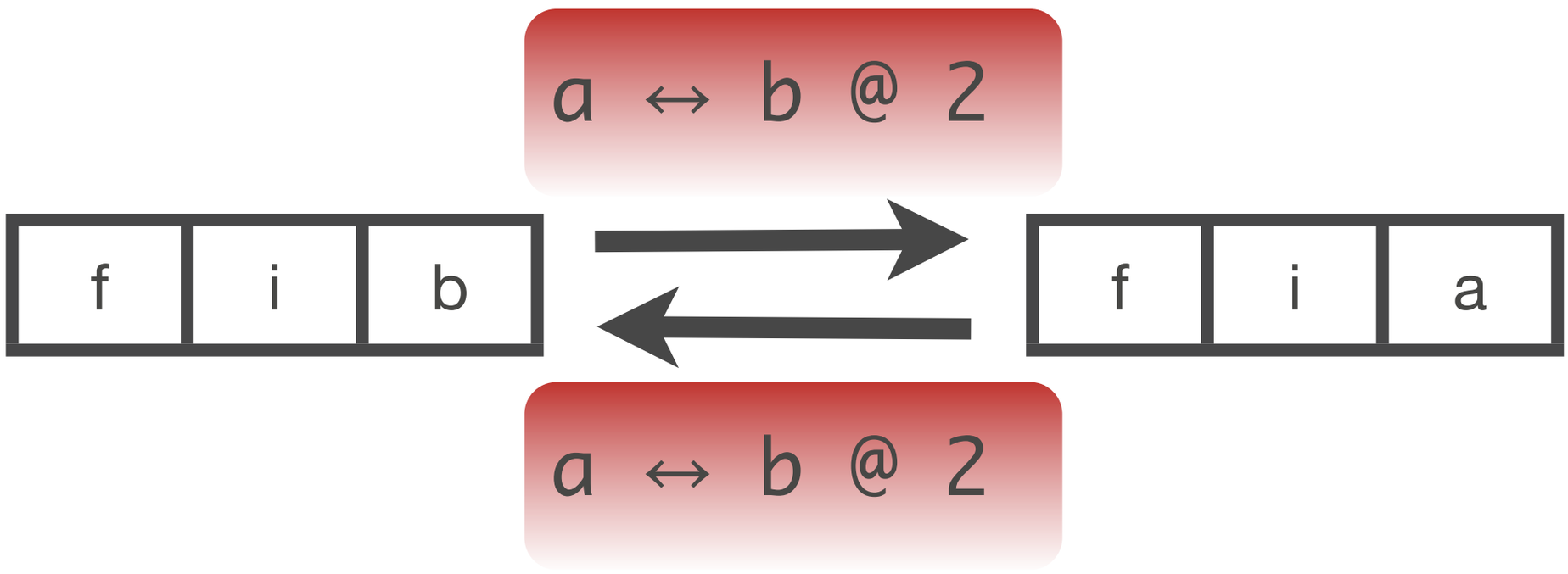
$$a \leftrightarrow b \ @ \ 2$$

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

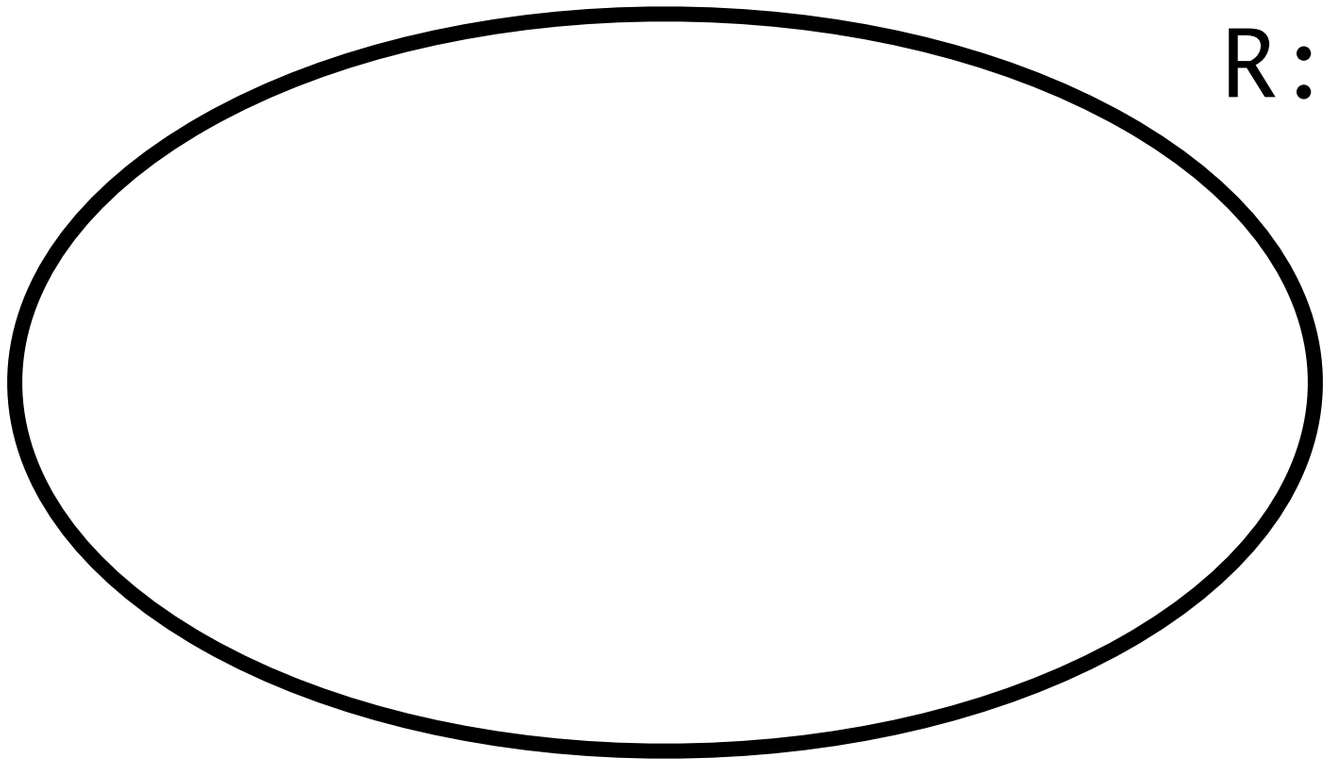
A patch theory

- Repository is a char vector of length n

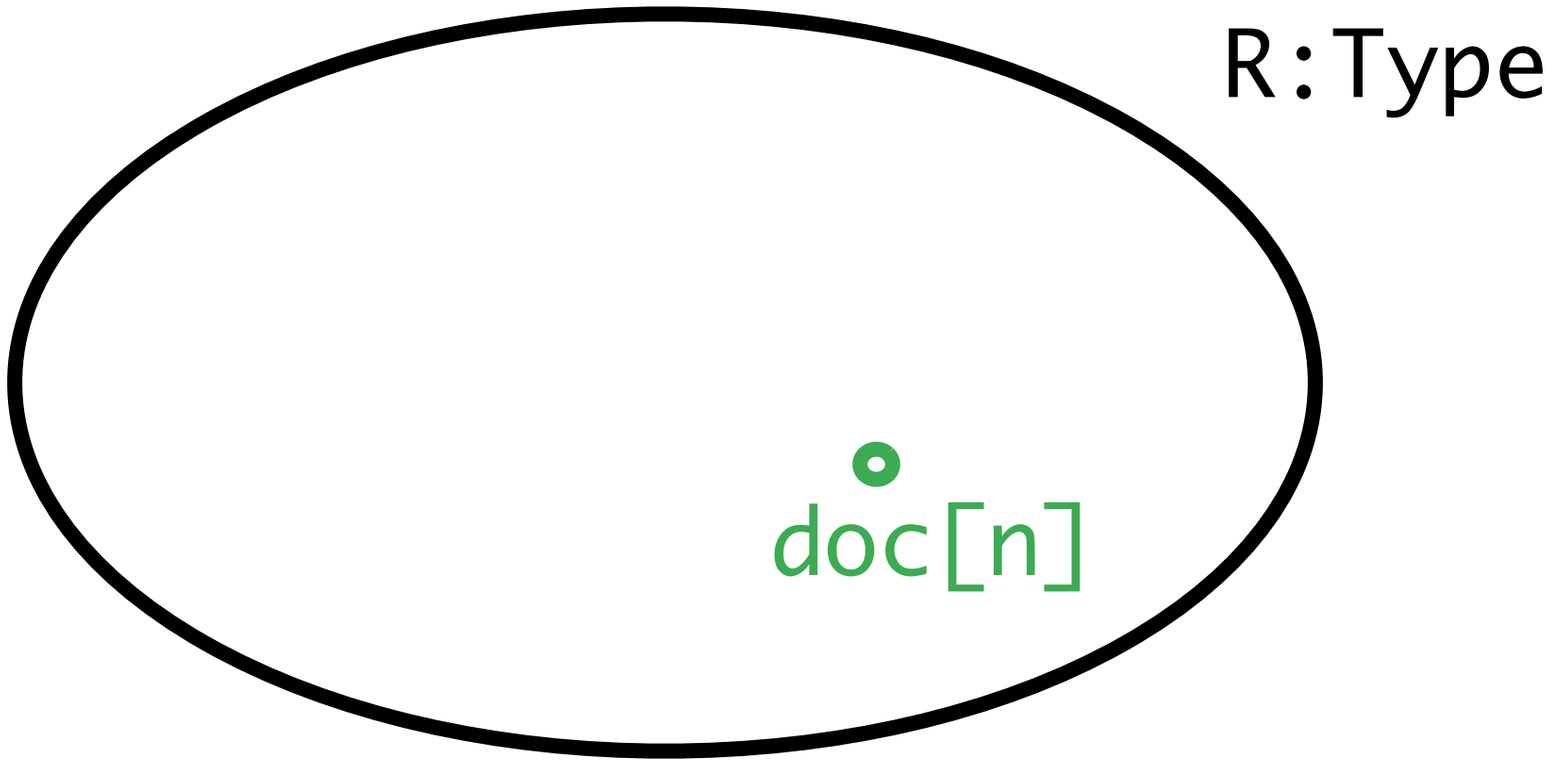
- Basic patch is `a ↔ b @ i` where `i<n`



A patch theory as a HIT

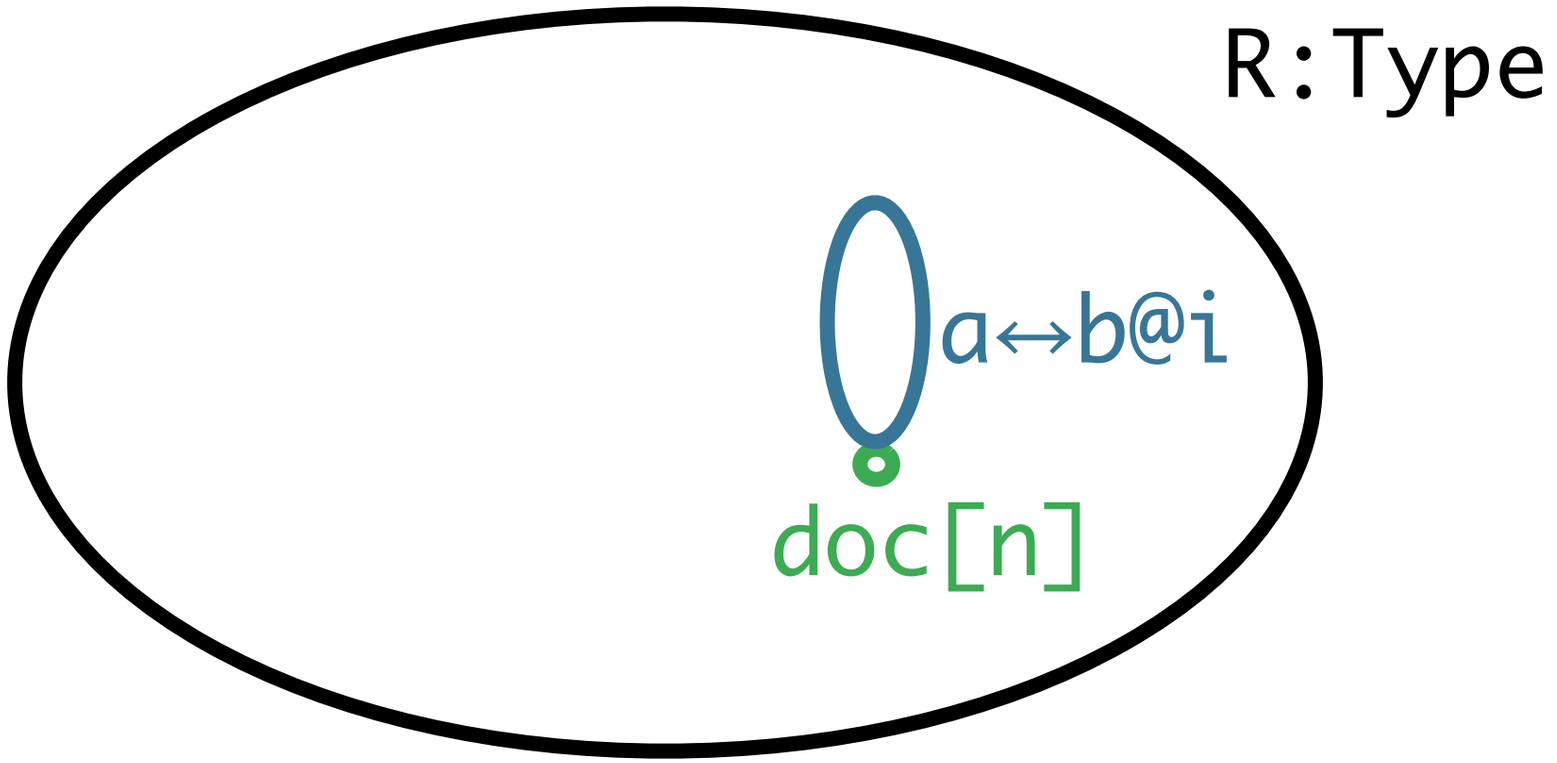


A patch theory as a HIT



points describe repository contents

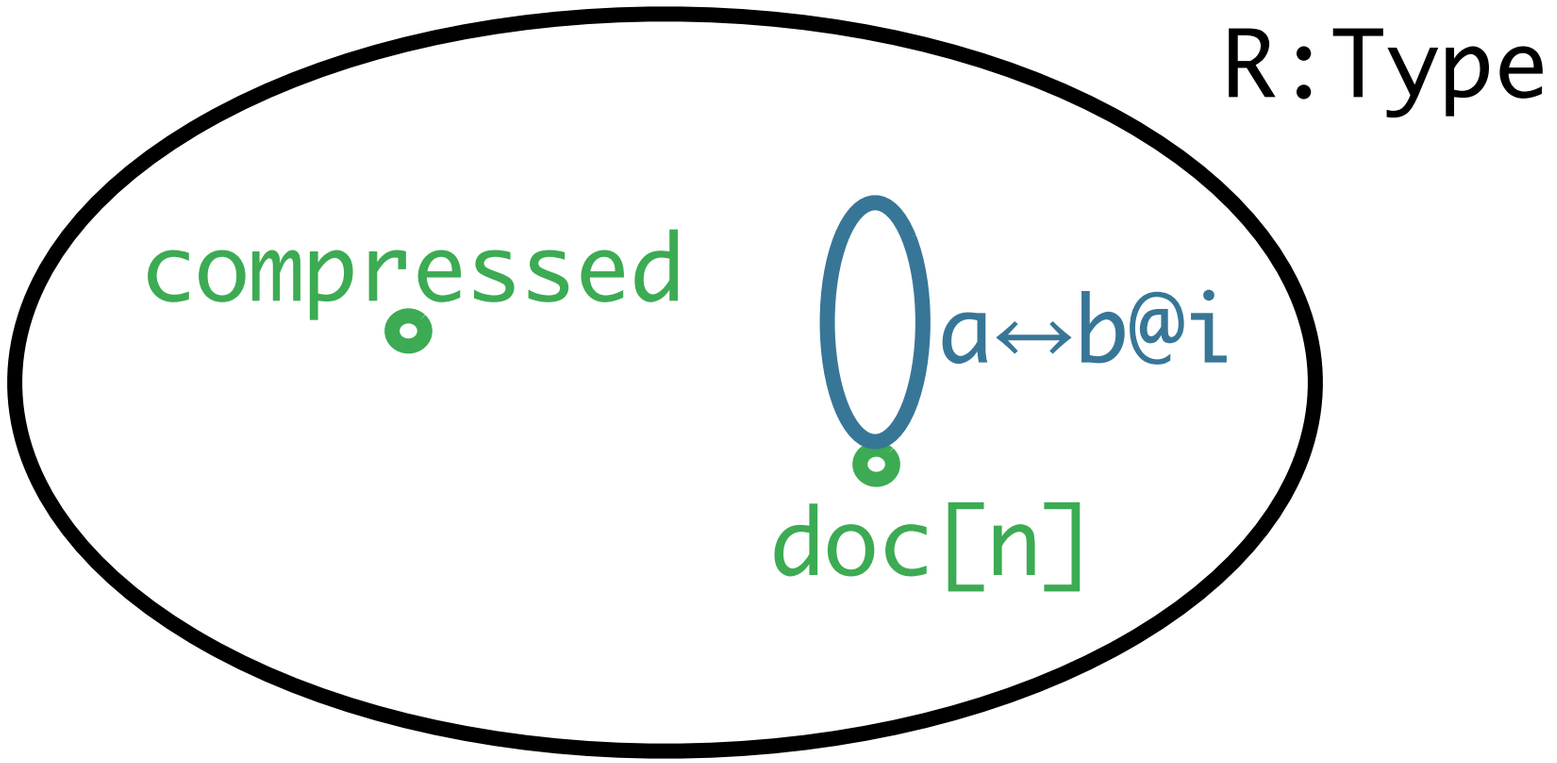
A patch theory as a HIT



points describe repository contents

paths are patches

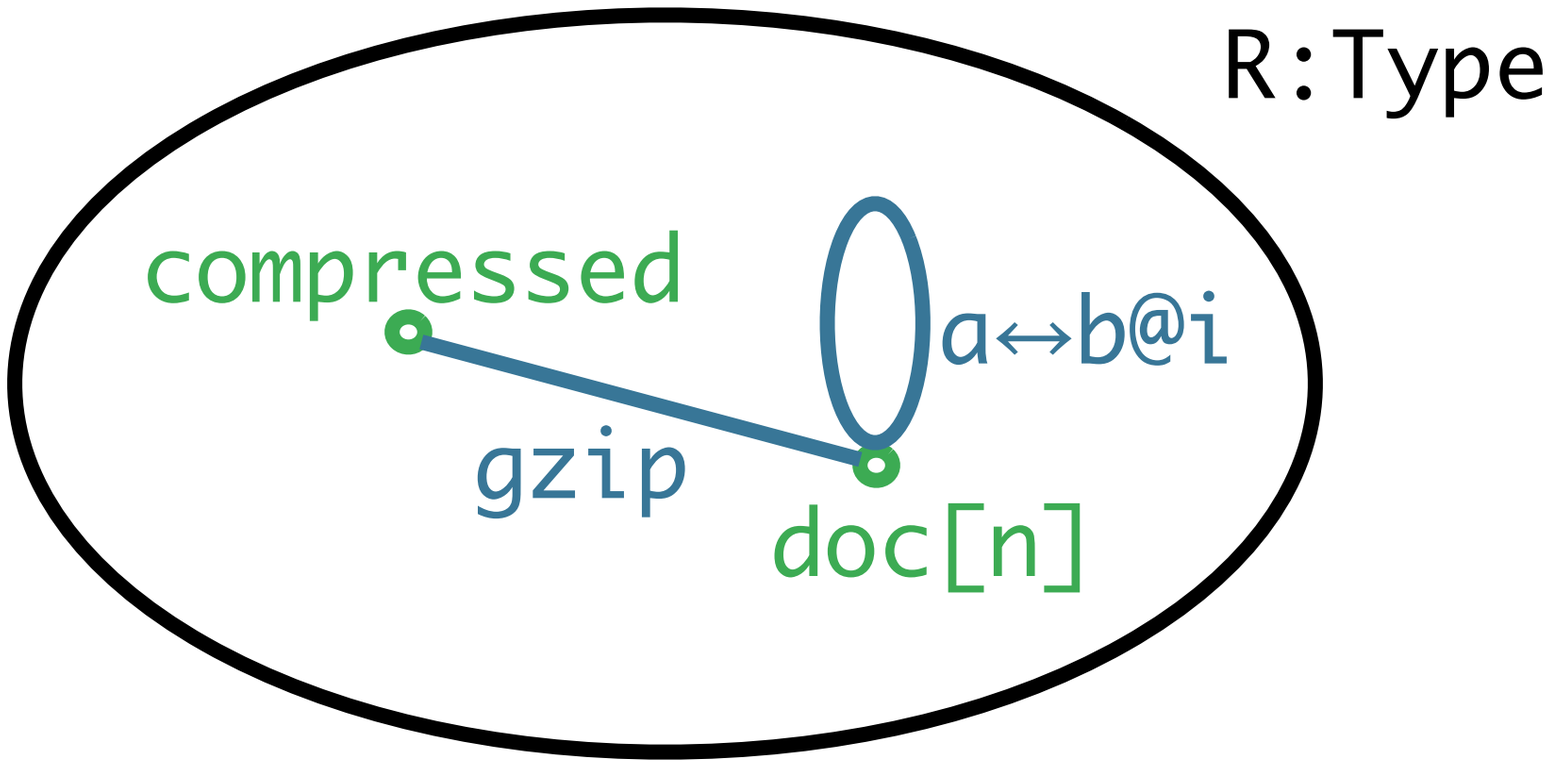
A patch theory as a HIT



points describe repository contents

paths are patches

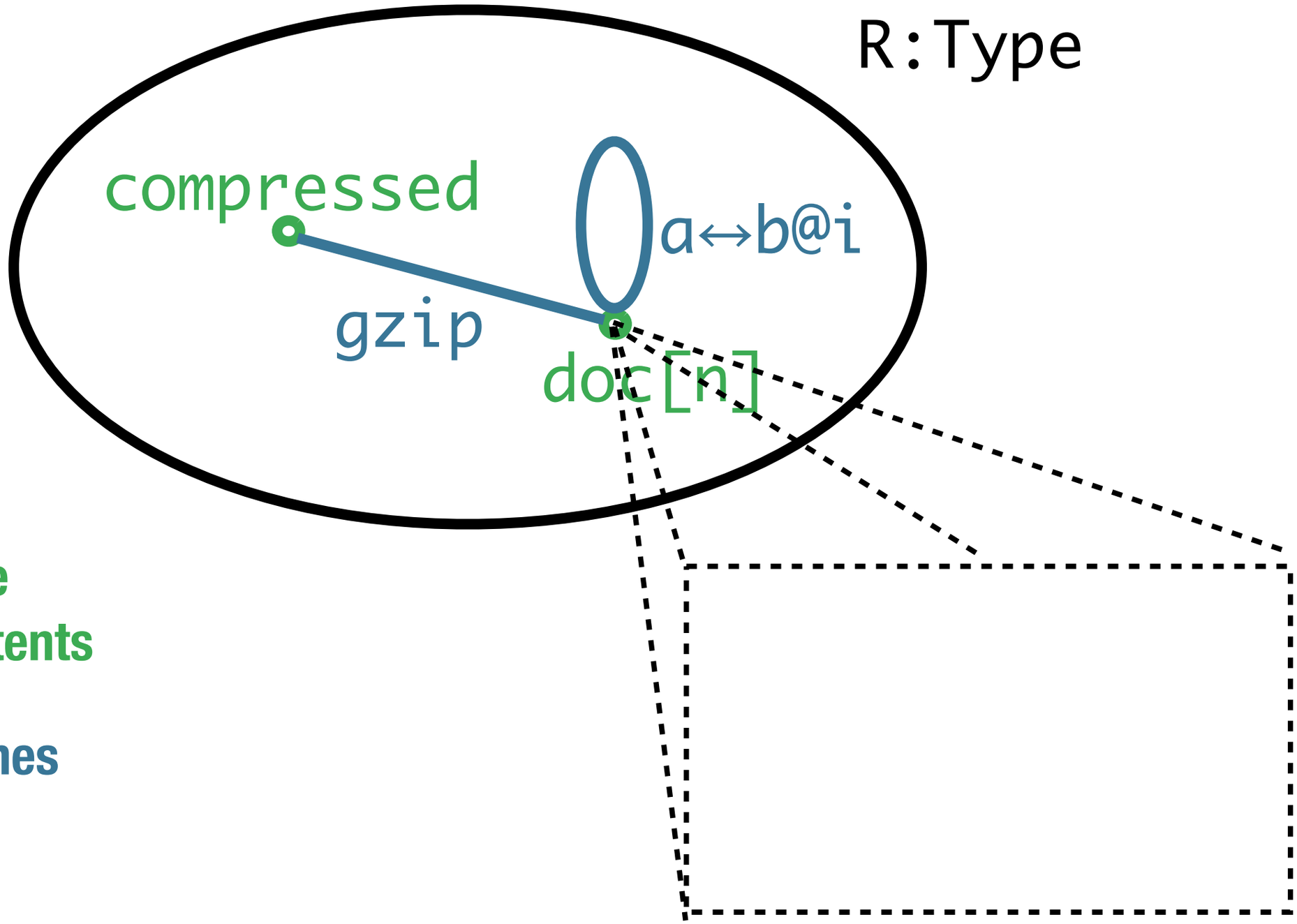
A patch theory as a HIT



points describe repository contents

paths are patches

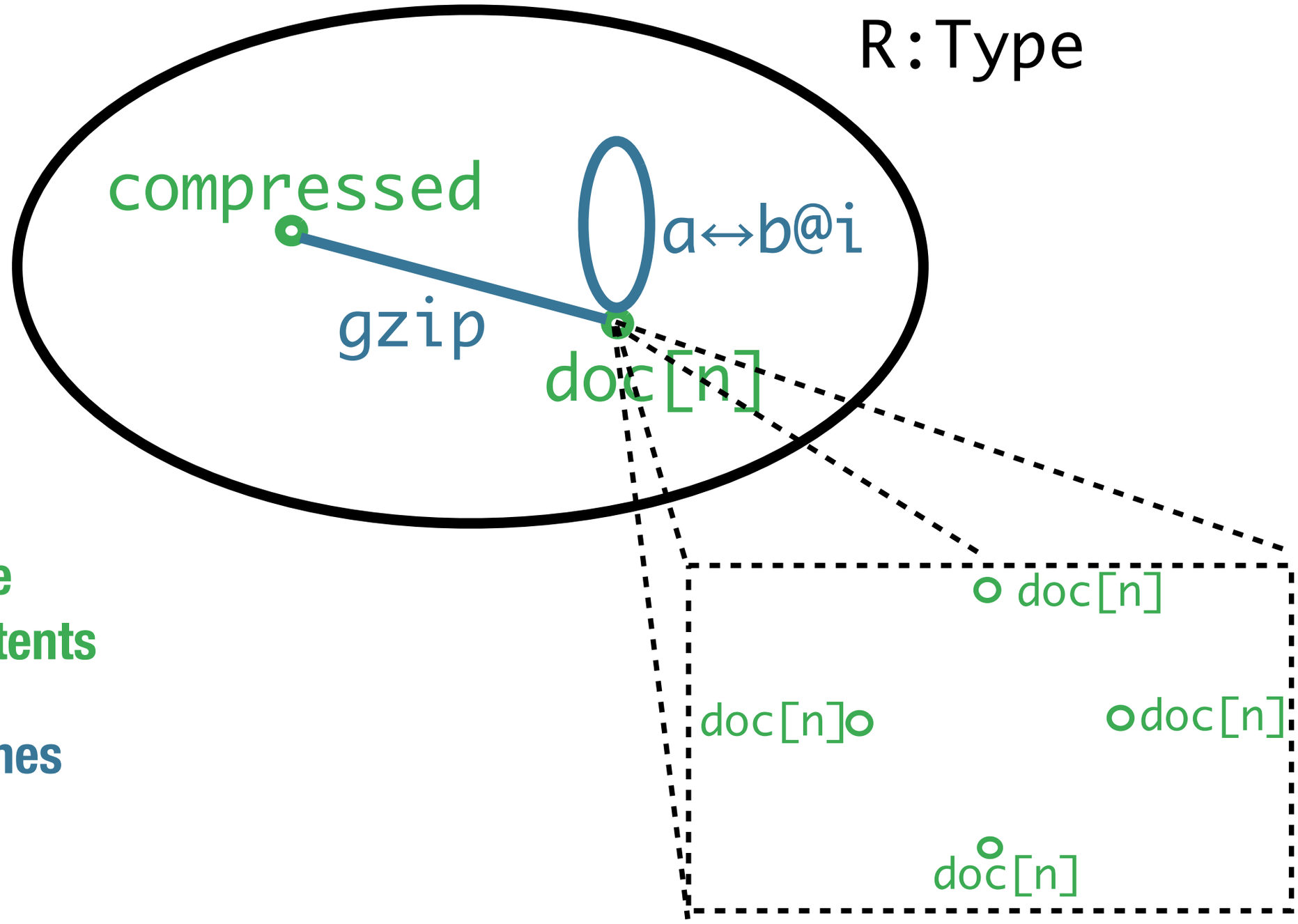
A patch theory as a HIT



points describe repository contents

paths are patches

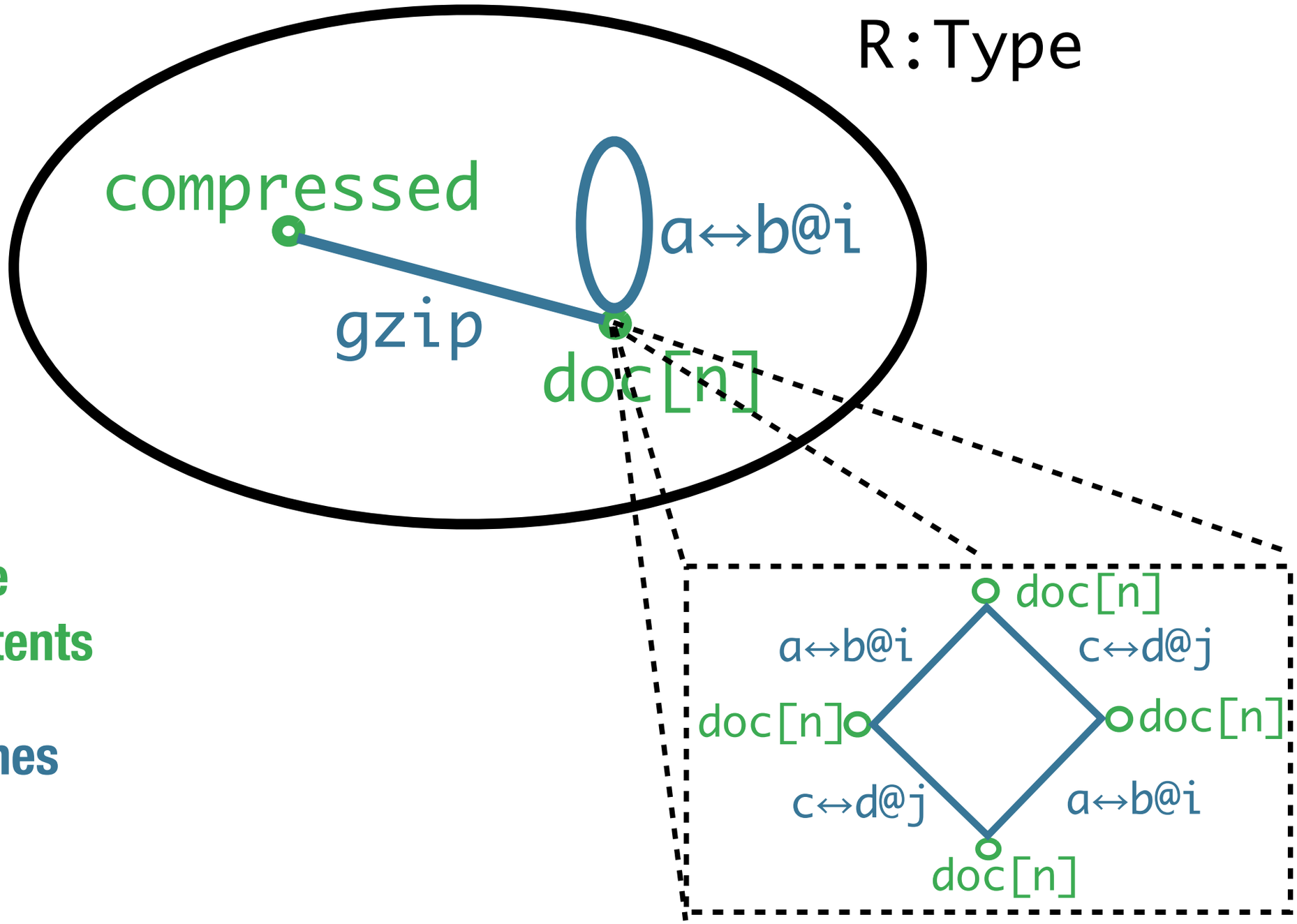
A patch theory as a HIT



points describe repository contents

paths are patches

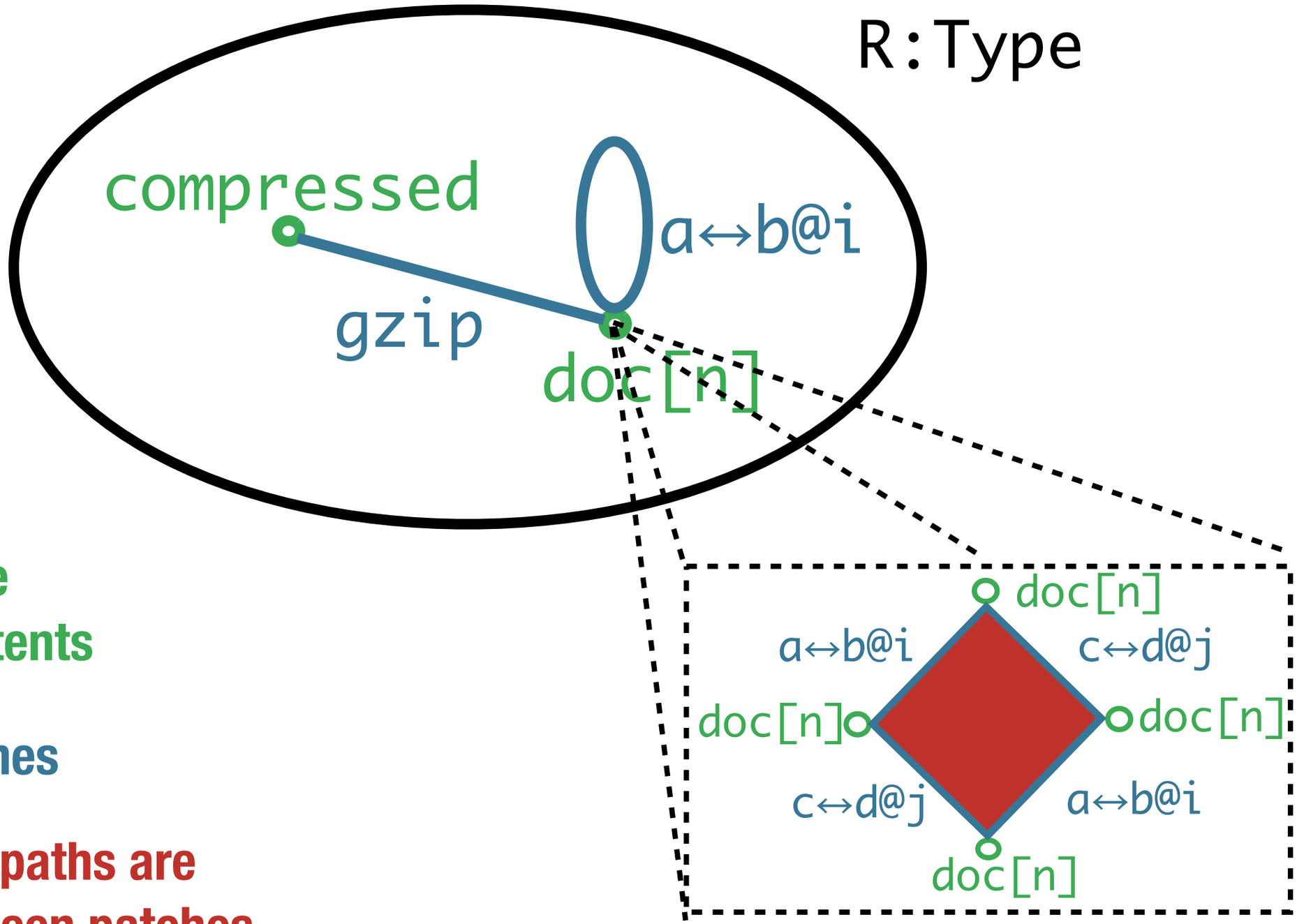
A patch theory as a HIT



points describe repository contents

paths are patches

A patch theory as a HIT

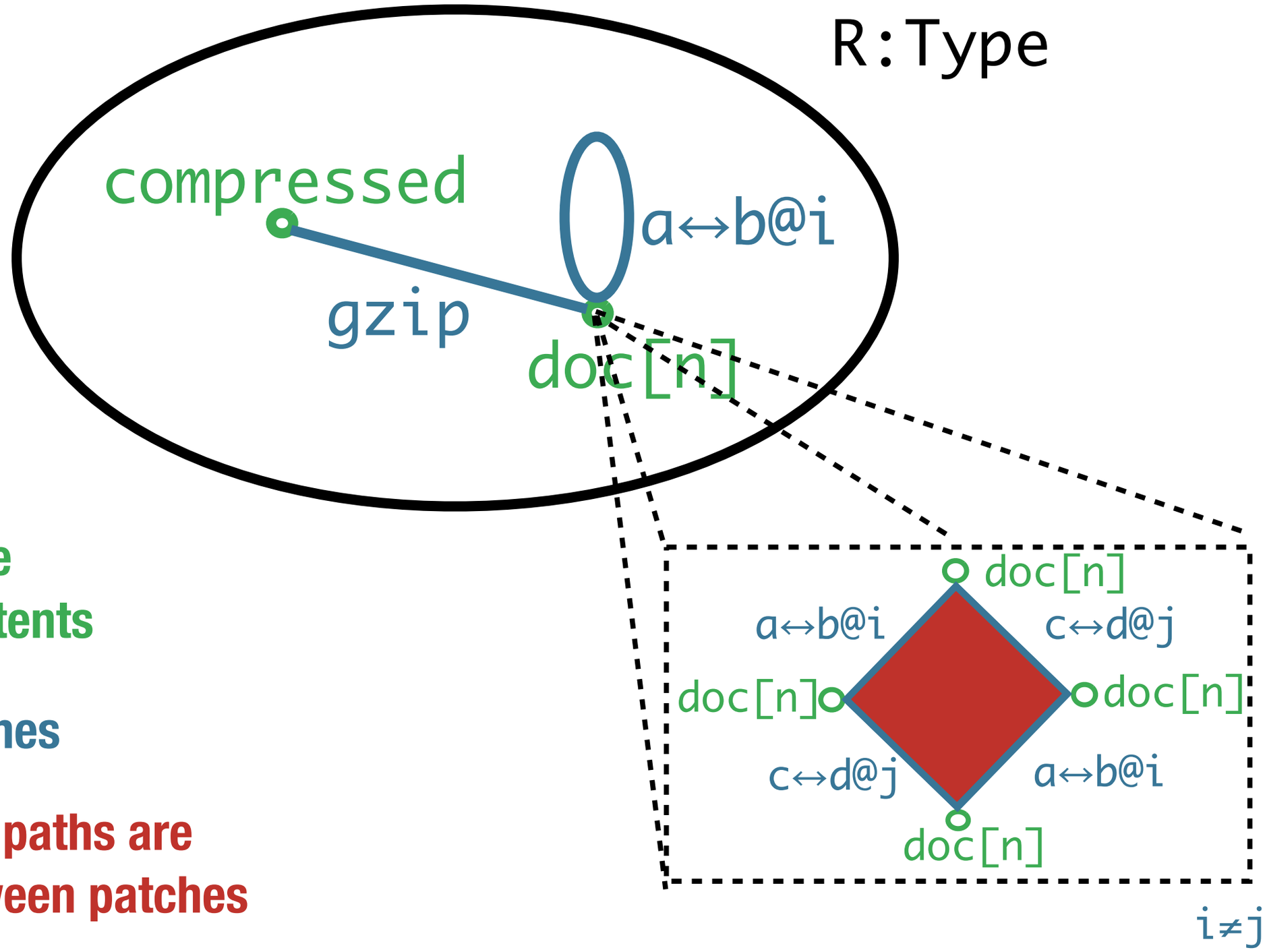


points describe repository contents

paths are patches

paths between paths are equations between patches

A patch theory as a HIT



points describe repository contents

paths are patches

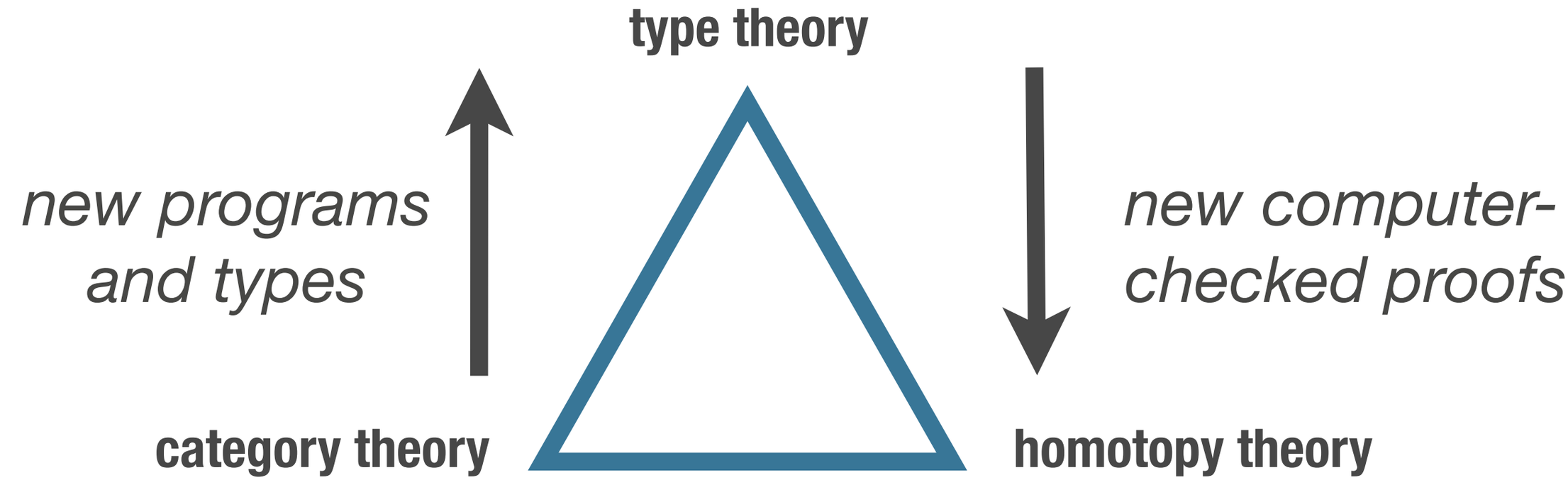
paths between paths are equations between patches

Higher inductive types

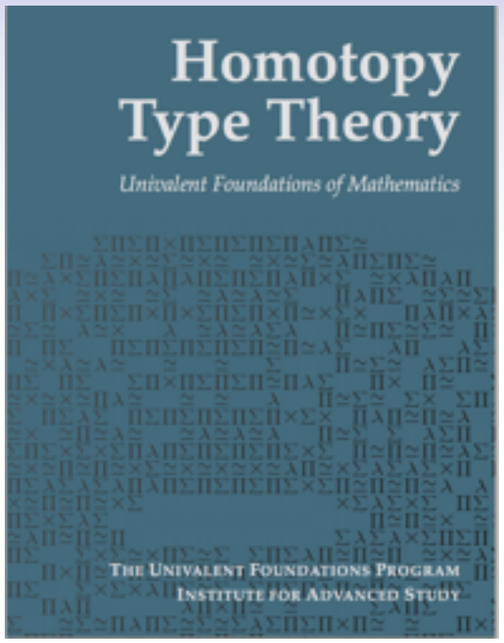
1. Quotient types
2. Spaces
3. Programming applications

Conclusion

Homotopy Type Theory



Reading list



1. The HoTT Book, homotopytypetheory.org

2. HoTT in Coq:
   github.com/hott/hott
   github.com/UniMath/UniMath

3. Homotopy theory in Agda:
   Fundamental group of the circle [LICS'13]
   $\pi_n(S^n) = \mathbb{Z}$ [CPP'13]
   Eilenberg-MacLane spaces [3:30pm today LICS]
   github.com/dlicata335/
   github.com/hott/hott-agda

4. Homotopical Patch Theory [ICFP'14]