Univalence from a Computer Science Point-of-View

Dan Licata
Wesleyan University

Martin-Löf type theory

[70s-80s]

The Coq Proof Assistant

PRL Project "Proof/Program Refinement Logic"

Agda

Agda is a dependently typed functional programming language.

LEAN

THEOREM PROVER

Idris

A Language with Dependent Types

Proofs are programs

```
data nat =
       zero
     | suc (n : nat)
```

cubicaltt
[Cohen,Coquand,
Huber,Mörtberg]

```
double : nat -> nat = split
  zero -> zero
  suc n -> suc (suc (double n))
```

cubicaltt [Cohen, Coquand, Huber, Mörtberg]

```
data nat =
      zero
    | suc (n : nat)

double : nat -> nat = split
  zero -> zero
  suc n -> suc (suc (double n))

even (n : nat) : U =
  (k : nat) * Path nat n (double k)

odd (n : nat) : U =
  (k : nat) * Path nat n (suc (double k))
```

cubicaltt
[Cohen, Coquand, Huber, Mörtberg]

```
data nat =
      zero
    | suc (n : nat)

double : nat -> nat = split
  zero -> zero
  suc n -> suc (suc (double n))

even (n : nat) : U =
  (k : nat) * Path nat n (double k)

odd (n : nat) : U =
  (k : nat) * Path nat n (suc (double k))
```

↑

"exists k : nat such that n = 2k+1"

Theorem: every natural number is even or odd
Proof: induction on n.

Base case: 0 is even

Inductive case: Suppose n is even or n is odd.
To show: n+1 is even or n+1 is odd.
Case where n is even (n=2k):
  n+1 = 2k+1 is odd.
Case where n is odd (n=2k+1):
  n+1 = 2k+2 = 2(k+1) is even.

“for all n : nat, n is even or n is odd”

```
evenodd : (n : nat) -> or (even n) (odd n) = split
  zero -> inl (zero, <_> zero)
  suc n -> step (evenodd n) where
    step : or (even n) (odd n) -> or (even (suc n)) (odd (suc n)) =
      split
       inl e ->
           -- if n is even (n=2k), then n+1 is odd (=2k+1)
           inr (e.1 ,
                -- n = 2k, so n+1 = 2k+1
                <x> suc (e.2 @ x))
       inr o ->
           -- if n is odd (=2k+1), then n+1 is even (= 2(k+1))
           inl (suc o.1,
                -- n = 2k+1, so n+1 = 2k+2 = 2(k+1)
                <x> (suc (o.2 @ x)))
```

“for all n : nat, n is even or n is odd”

```
evenodd : (n : nat) -> or (even n) (odd n) = split
  zero -> inl (zero, <_> zero)
  suc n -> step (evenodd n) where
    step : or (even n) (odd n) -> or (even (suc n)) (odd (suc n)) =
      split
       inl e ->
           -- if n is even (n=2k), then n+1 is odd (=2k+1)
           inr (e.1 ,
                -- n = 2k, so n+1 = 2k+1
                <x> suc (e.2 @ x))
       inr o ->
           -- if n is odd (=2k+1), then n+1 is even (= 2(k+1))
           inl (suc o.1,
                -- n = 2k+1, so n+1 = 2k+2 = 2(k+1)
                <x> (suc (o.2 @ x)))
```

What program is this?

evenodd.ctt

“for all” is function   “or” is coproduct

```
evenodd : (n : nat) -> or (even n) (odd n) = split
  zero -> inl (zero, <_> zero)
  suc n -> step (evenodd n) where
    step : or (even n) (odd n) -> or (even (suc n)) (odd (suc n)) =
      split
       inl e ->
           -- if n is even (n=2k), then n+1 is odd (=2k+1)
           inr (e.1 ,
                -- n = 2k, so n+1 = 2k+1
                <x> suc (e.2 @ x))
       inr o ->
           -- if n is odd (=2k+1), then n+1 is even (= 2(k+1))
           inl (suc o.1,
                -- n = 2k+1, so n+1 = 2k+2 = 2(k+1)
                <x> (suc (o.2 @ x)))
```

What program is this?

coproduct injection is parity

```
evenodd : (n : nat) -> or (even n) (odd n) = split
  zero -> inl (zero, <_> zero)
  suc n -> step (evenodd n) where
    step : or (even n) (odd n) -> or (even (suc n)) (odd (suc n)) =
      split
       inl e ->
           -- if n is even (n=2k), then n+1 is odd (=2k+1)
           inr (e.1 ,
                -- n = 2k, so n+1 = 2k+1
                <x> suc (e.2 @ x))
       inr o ->
           -- if n is odd (=2k+1), then n+1 is even (= 2(k+1))
           inl (suc o.1,
                -- n = 2k+1, so n+1 = 2k+2 = 2(k+1)
                <x> (suc (o.2 @ x)))
```

What program is this?

floor(n/2)

```
evenodd : (n : nat) -> or (even n) (odd n) = split
  zero -> inl (zero, <_> zero)
  suc n -> step (evenodd n) where
    step : or (even n) (odd n) -> or (even (suc n)) (odd (suc n)) =
      split
       inl e ->
           -- if n is even (n=2k), then n+1 is odd (=2k+1)
           inr (e.1 ,
                -- n = 2k, so n+1 = 2k+1
                <x> suc (e.2 @ x))
       inr o ->
           -- if n is odd (=2k+1), then n+1 is even (= 2(k+1))
           inl (suc o.1,
                -- n = 2k+1, so n+1 = 2k+2 = 2(k+1)
                <x> (suc (o.2 @ x)))
```

What program is this?

proof that n = 2*floor(n/2)[+1]

```
evenodd : (n : nat) -> or (even n) (odd n) = split
  zero -> inl (zero, <_> zero)
  suc n -> step (evenodd n) where
    step : or (even n) (odd n) -> or (even (suc n)) (odd (suc n)) =
      split
       inl e ->
           -- if n is even (n=2k), then n+1 is odd (=2k+1)
           inr (e.1 ,
                -- n = 2k, so n+1 = 2k+1
                <x> suc (e.2 @ x))
       inr o ->
           -- if n is odd (=2k+1), then n+1 is even (= 2(k+1))
           inl (suc o.1,
                -- n = 2k+1, so n+1 = 2k+2 = 2(k+1)
                <x> (suc (o.2 @ x)))
```

What program is this?

proof that n = 2*floor(n/2)[+1]

```
evenodd : (n : nat) -> or (even n) (odd n) = split
  zero -> inl (zero, <_> zero)
  suc n -> step (evenodd n) where
    step : or (even n) (odd n) -> or (even (suc n)) (odd (suc n)) =
      split
       inl e ->
           -- if n is even (n=2k), then n+1 is odd (=2k+1)
           inr (e.1 ,
                -- n = 2k, so n+1 = 2k+1
                <x> suc (e.2 @ x))
       inr o ->
           -- if n is odd (=2k+1), then n+1 is even (= 2(k+1))
           inl (suc o.1,
                -- n = 2k+1, so n+1 = 2k+2 = 2(k+1)
                <x> (suc (o.2 @ x)))
```

(every element of Path nat k k *is reflexivity/identity)*

Computation

elimination reduces on introduction

- function applied to argument reduces to body of definition
- projection of a pair reduces to component
- case distinction for coproduct reduces on injection
- recursion on nat reduces on zero and suc(n)

Computation

elimination reduces on introduction

- function applied to argument reduces
  to body of definition

- projection of a pair reduces to component

- case distinction for coproduct reduces on injection

- recursion on nat reduces on zero and suc(n)

“definitional” rather than “typal” equalities/paths

Computation

elimination reduces on introduction

- function applied to argument reduces
  to body of definition
- projection of a pair reduces to component
- case distinction for coproduct reduces on injection
- recursion on nat reduces on zero and suc(n)

“definitional” rather than “typal” equalities/paths

equality

Computation

elimination reduces on introduction

- function applied to argument reduces to body of definition
- projection of a pair reduces to component
- case distinction for coproduct reduces on injection
- recursion on nat reduces on zero and suc(n)

“definitional” rather than “typal” equalities/paths

equality **1-simplex**

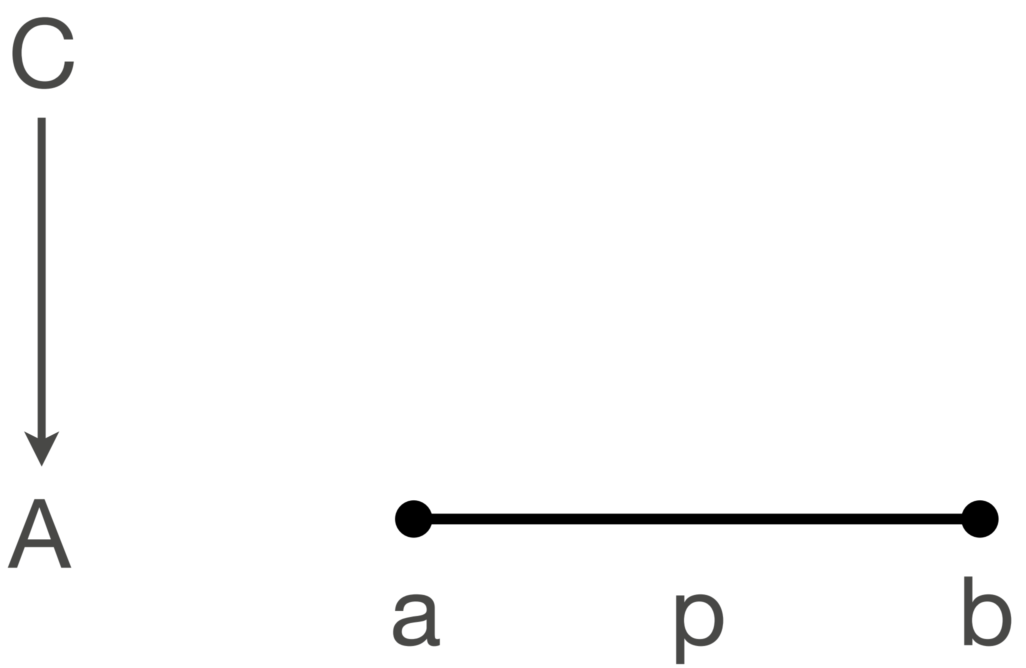
Computation

elimination reduces on introduction

$C : A \to \text{Type}$

Computation

elimination reduces on introduction

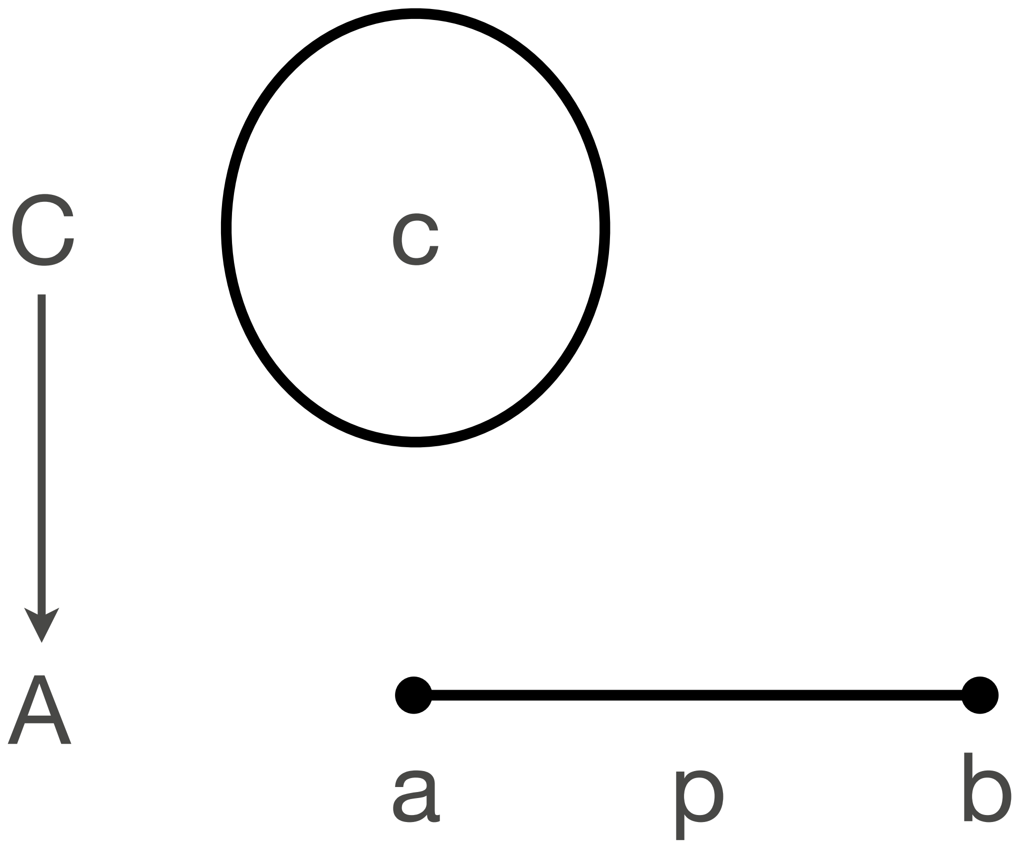


C : A → Type

p : Path A a b

Computation

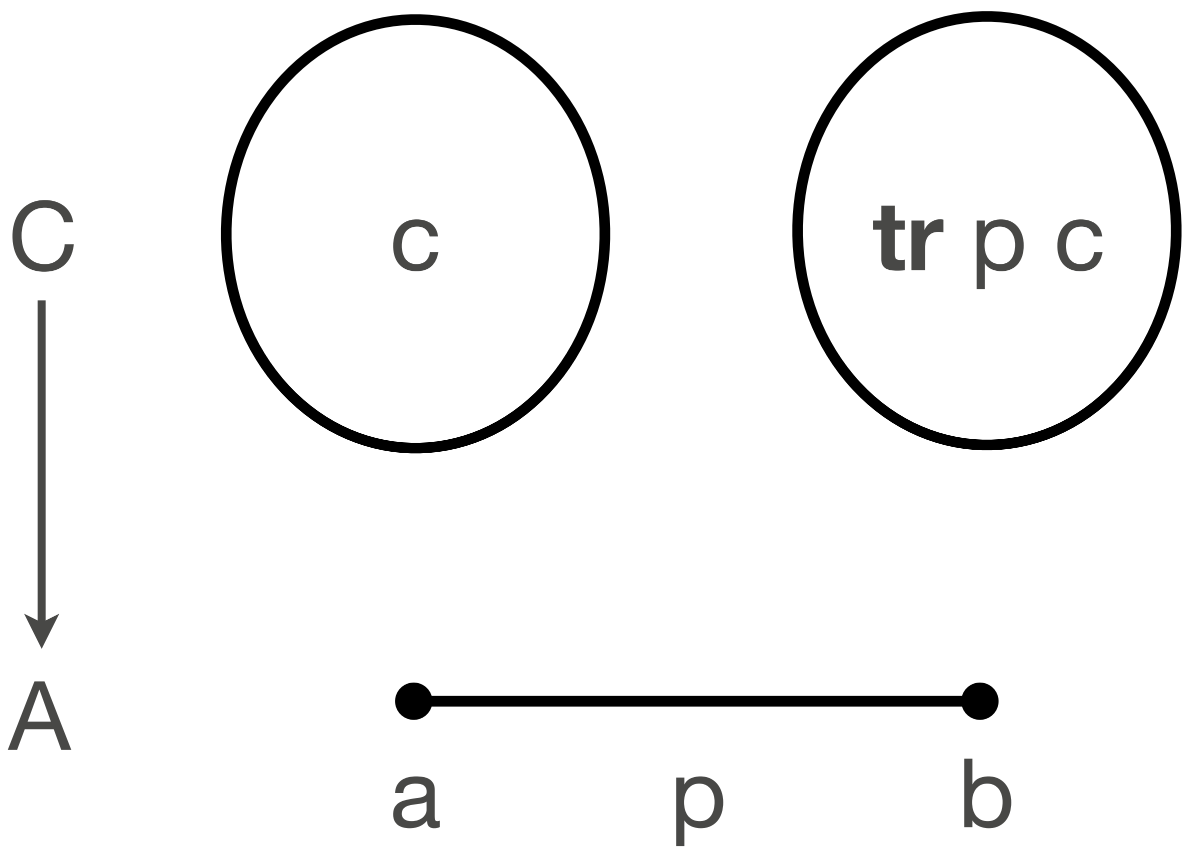
elimination reduces on introduction



C : A → Type
p : Path A a b
c : C(a)

Computation

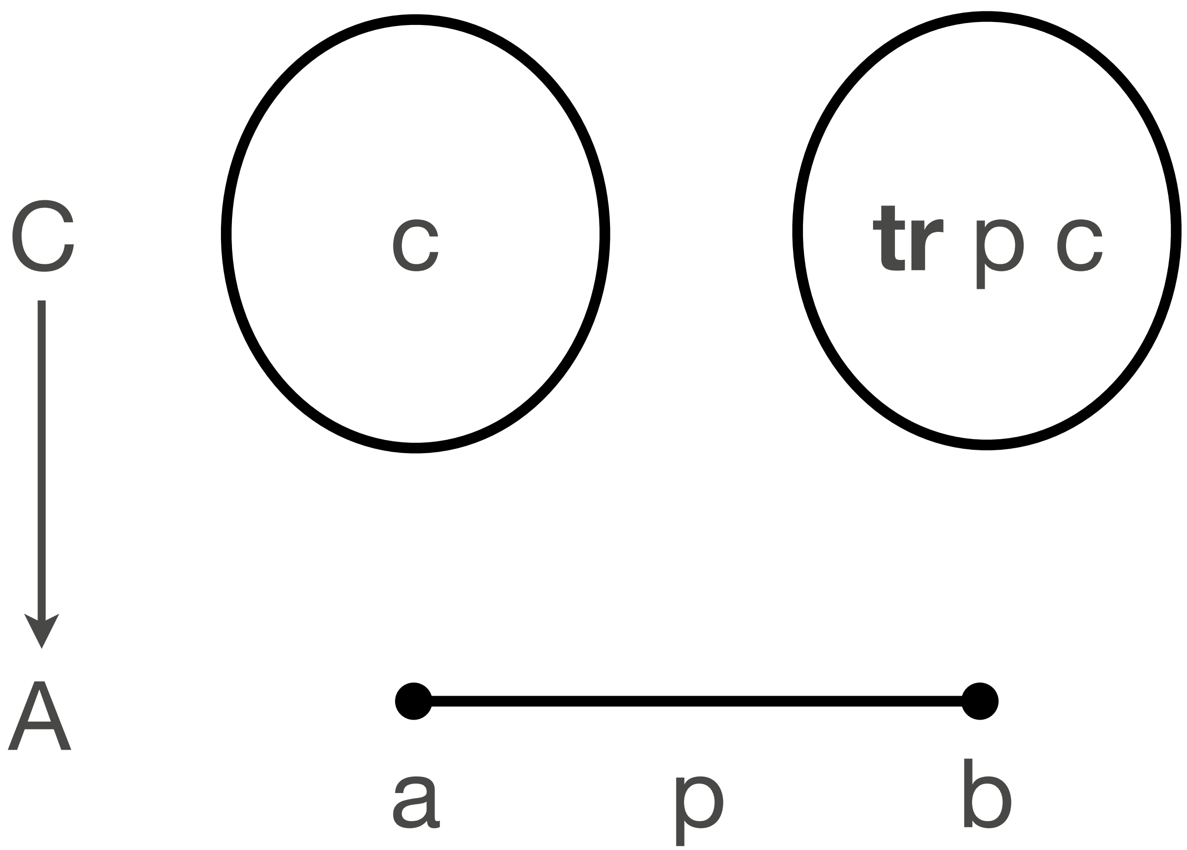
elimination reduces on introduction



C : A → Type
p : Path A a b
c : C(a)
then
transport C p c : C(b)

Computation

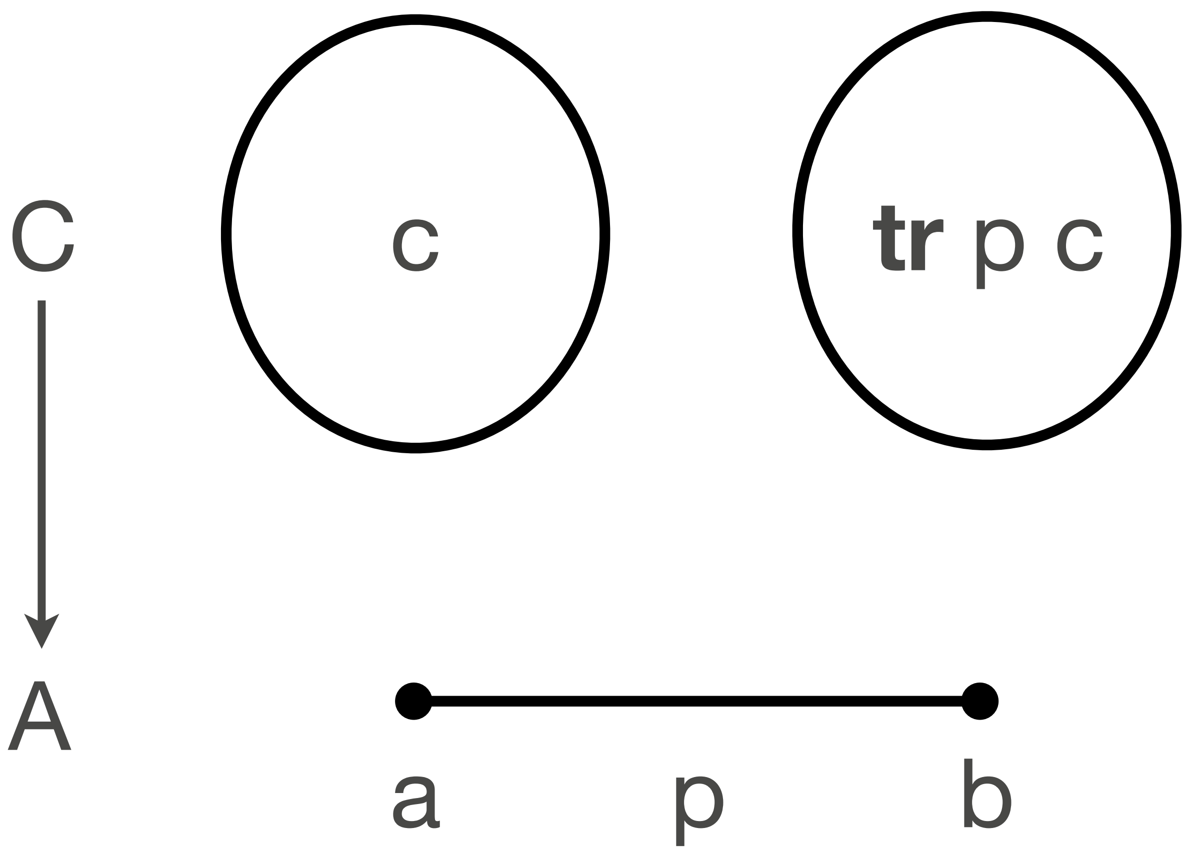
elimination reduces on introduction



C : A → Type
p : Path A a b
c : C(a)
then
transport C p c : C(b)
and reduces to c
when p is identity Path A a a

Computation

elimination reduces on introduction



C : A → Type
p : Path A a b
c : C(a)
then
transport C p c : C(b)
and reduces to c
when p is identity Path A a a

original "intended model" of MLTT: every "path" is identity

Canonicity theorem

Constructive proof of:

For all (closed) `t:nat` in MLTT,
there exists a numeral `k` with
`t` definitionally equal to `k`

Univalence Axiom

$(A,B : U) \to \text{Equiv } A\ B \xrightarrow{\sim} \text{Path } U\ A\ B$

Univalence Axiom

$(A, B : U) \to \text{Equiv } A\ B \xrightarrow{\sim} \text{Path } U\ A\ B$

axioms break canonicity

Univalence Axiom

$$(A,B : U) \to \text{Equiv } A\ B \xrightarrow{\sim} \text{Path } U\ A\ B$$

axioms break canonicity

Central question for computation with univalence:
what does it mean to **transport** along such a path?

Voevodsky's homotopy canonicity conjecture

Some of the key univalent concepts (cont.)

9. Unlike many other axioms (e.g. the axiom of excluded middle), the univalence axiom is expected "to have computational content". In other words decidable normalization should be extendable in a certain sense to terms which involve the univalence axiom. For example there is the following precise:

   Conjecture 1. There exists a terminating algorithm which for any term expression t of type [nat] (natural numbers) constructed using the univalence axiom returns a term expression t' of type [nat] which does not use univalence axiom and a term expression of the identity type [Id nat t t'] which may use the univalence axiom.

[talk in Götenborg, 2011]

Voevodsky's homotopy canonicity conjecture

Constructive proof of:

For all (closed) `t:nat` in MLTT+univalence,
there exists a numeral `k` with
a `Path nat t k` (potentially using univalence)

Voevodsky's homotopy canonicity conjecture

Constructive proof of:

For all (closed) `t:nat` in MLTT+univalence,
there exists a numeral k with
a `Path nat t k` (potentially using univalence)

computation valid in all models

Voevodsky's homotopy canonicity conjecture

Constructive proof of:

For all (closed) `t:nat` in MLTT+univalence,
there exists a numeral k with
a `Path nat t k` (potentially using univalence)

computation valid in all models

ua already implies how ua "computes"

Progress

Progress

- Models of MLTT+ua in a constructive metatheory (procedure for running programs implicit in proof!)

Progress

- Models of MLTT+ua in a constructive metatheory (procedure for running programs implicit in proof!)
- Models of MLTT+ua based on programs and operational semantics

Progress

- Models of MLTT+ua in a constructive metatheory (procedure for running programs implicit in proof!)
- Models of MLTT+ua based on programs and operational semantics
- New type theories including MLTT+ua that satisfy (definitional) canonicity

Progress

- Models of MLTT+ua in a constructive metatheory (procedure for running programs implicit in proof!)
- Models of MLTT+ua based on programs and operational semantics
- New type theories including MLTT+ua that satisfy (definitional) canonicity
- Several new experimental proof assistants

Progress

- Models of MLTT+ua in a constructive metatheory (procedure for running programs implicit in proof!)
- Models of MLTT+ua based on programs and operational semantics
- New type theories including MLTT+ua that satisfy (definitional) canonicity
- Several new experimental proof assistants

maybe new type theories can be interpreted in same kinds of models?

Progress

- Models of MLTT+ua in a constructive metatheory (procedure for running programs implicit in proof!)
- Models of MLTT+ua based on programs and operational semantics
- New type theories including MLTT+ua that satisfy (definitional) canonicity
- Several new experimental proof assistants

maybe new type theories can be interpreted in same kinds of models?

definitional equalities are easier to use

Constructive Cubical Models

Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]

Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]
- de Morgan cube category, composition→filling, cofibration syntax [Cohen,Coquand,Huber,Mörtberg,'14-'15]

Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]
- de Morgan cube category, composition→filling, cofibration syntax [Cohen,Coquand,Huber,Mörtberg,'14-'15]
- cartesian cube category, homogenization, diagonal cofibs [Angiuli,Favonia,Harper,Wilson;+Brunerie,Coquand,L.,'14-'17]

Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]
- de Morgan cube category, composition→filling, cofibration syntax [Cohen,Coquand,Huber,Mörtberg,'14-'15]
- cartesian cube category, homogenization, diagonal cofibs [Angiuli,Favonia,Harper,Wilson;+Brunerie,Coquand,L.,'14-'17]
- canonicity/operational models [Huber,'16; Angiuli,Favonia,Harper,Wilson,'16-'17]

Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]
- de Morgan cube category, composition→filling, cofibration syntax [Cohen,Coquand,Huber,Mörtberg,'14-'15]
- cartesian cube category, homogenization, diagonal cofibs [Angiuli,Favonia,Harper,Wilson;+Brunerie,Coquand,L.,'14-'17]
- canonicity/operational models [Huber,'16; Angiuli,Favonia,Harper,Wilson,'16-'17]
- higher inductive types: syntax, fibrancy/operational semantics [Isaev'14,Coquand,Huber,Mörtberg,'14-'17;Cavallo,Harper,'17]

Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]
- de Morgan cube category, composition→filling, cofibration syntax [Cohen,Coquand,Huber,Mörtberg,'14-'15]
- cartesian cube category, homogenization, diagonal cofibs [Angiuli,Favonia,Harper,Wilson;+Brunerie,Coquand,L.,'14-'17]
- canonicity/operational models [Huber,'16; Angiuli,Favonia,Harper,Wilson,'16-'17]
- higher inductive types: syntax, fibrancy/operational semantics [Isaev'14,Coquand,Huber,Mörtberg,'14-'17;Cavallo,Harper,'17]
- type theory with non-fibrant types and exact equality as in Voevodsky's HTS [AFH'17]

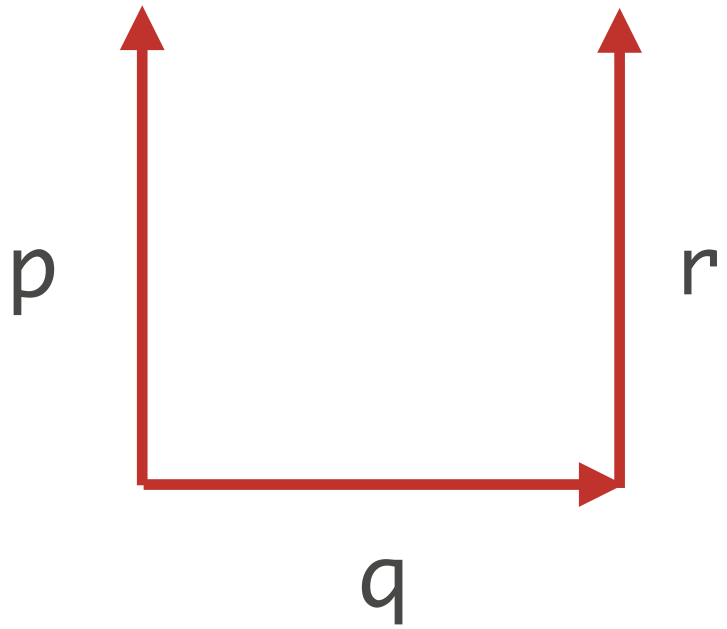
Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]
- de Morgan cube category, composition→filling, cofibration syntax [Cohen,Coquand,Huber,Mörtberg,'14-'15]
- cartesian cube category, homogenization, diagonal cofibs [Angiuli,Favonia,Harper,Wilson;+Brunerie,Coquand,L.,'14-'17]
- canonicity/operational models [Huber,'16; Angiuli,Favonia,Harper,Wilson,'16-'17]
- higher inductive types: syntax, fibrancy/operational semantics [Isaev'14,Coquand,Huber,Mörtberg,'14-'17;Cavallo,Harper,'17]
- type theory with non-fibrant types and exact equality as in Voevodsky's HTS [AFH'17]
- internal language presentations (fibrancy as type) [Coquand;Orton-Pitts'16;Spitters+'16] + universes [Sattler;LOPS'18]

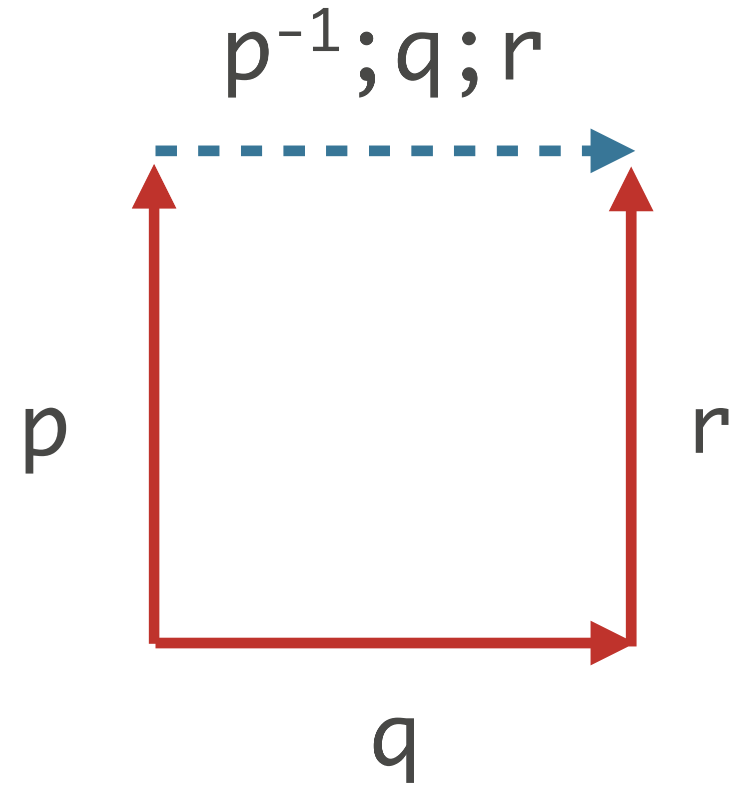
Constructive Cubical Models

- monoidal cube category, uniformity [Bezem,Coquand,Huber'13]
- de Morgan cube category, composition→filling, cofibration syntax [Cohen,Coquand,Huber,Mörtberg,'14-'15]
- cartesian cube category, homogenization, diagonal cofibs [Angiuli,Favonia,Harper,Wilson;+Brunerie,Coquand,L.,'14-'17]
- canonicity/operational models [Huber,'16; Angiuli,Favonia,Harper,Wilson,'16-'17]
- higher inductive types: syntax, fibrancy/operational semantics [Isaev'14,Coquand,Huber,Mörtberg,'14-'17;Cavallo,Harper,'17]
- type theory with non-fibrant types and exact equality as in Voevodsky's HTS [AFH'17]
- internal language presentations (fibrancy as type) [Coquand;Orton-Pitts'16;Spitters+'16] + universes [Sattler;LOPS'18]
- more mathematical analyses [Awodey'13-,Gambino,Sattler'15-'17]

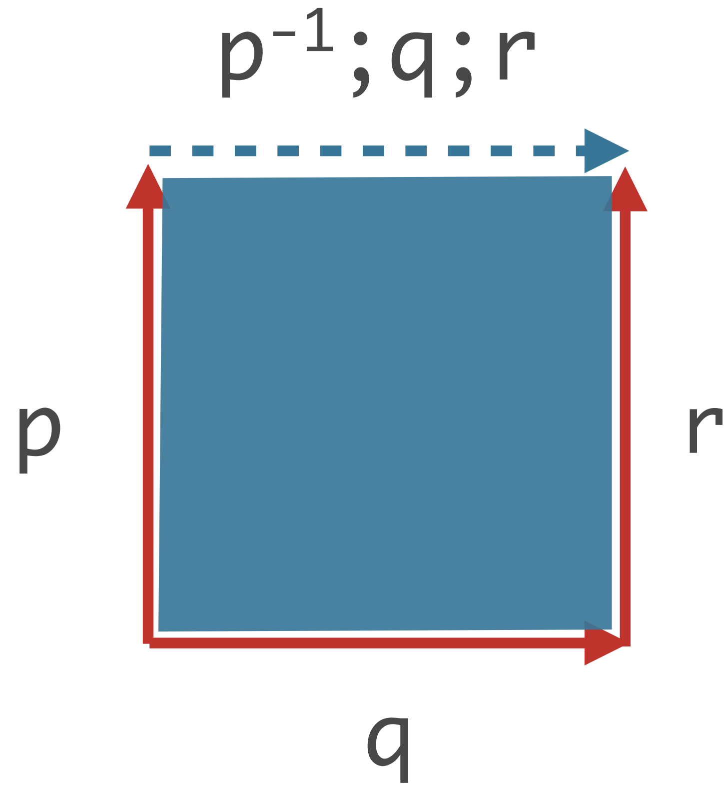
Kan filling



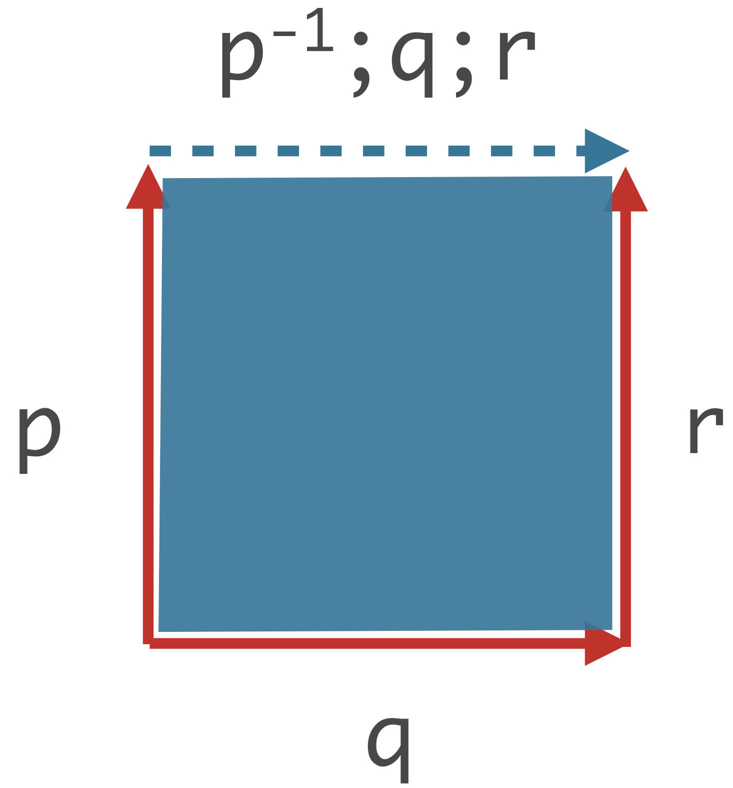
Kan filling

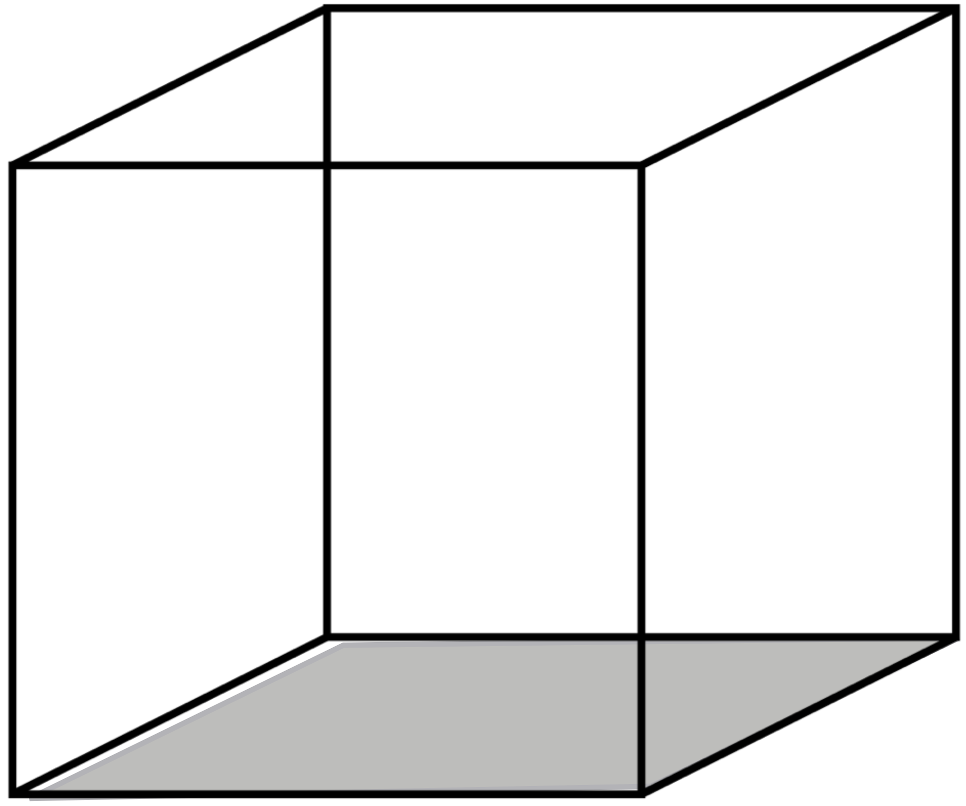


Kan filling

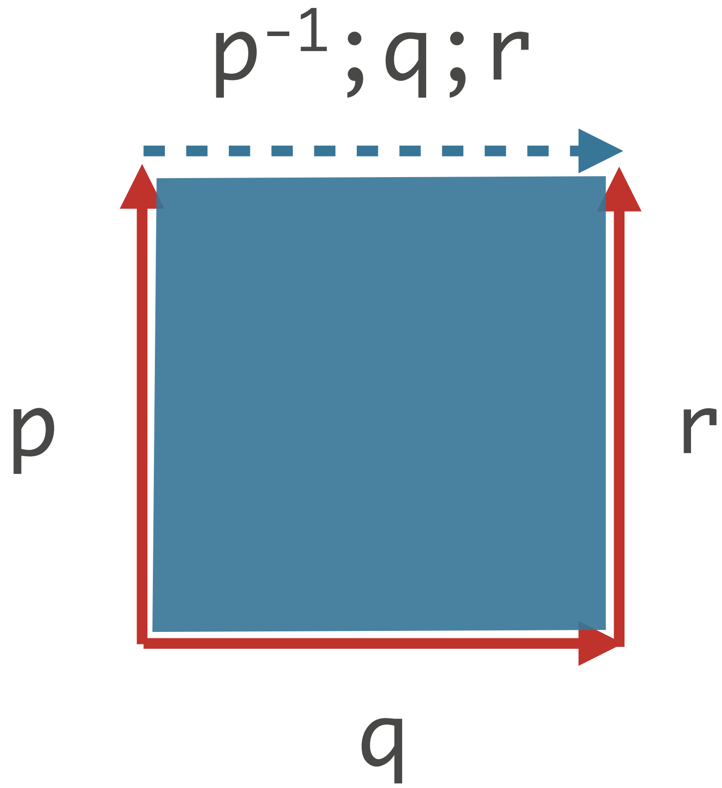


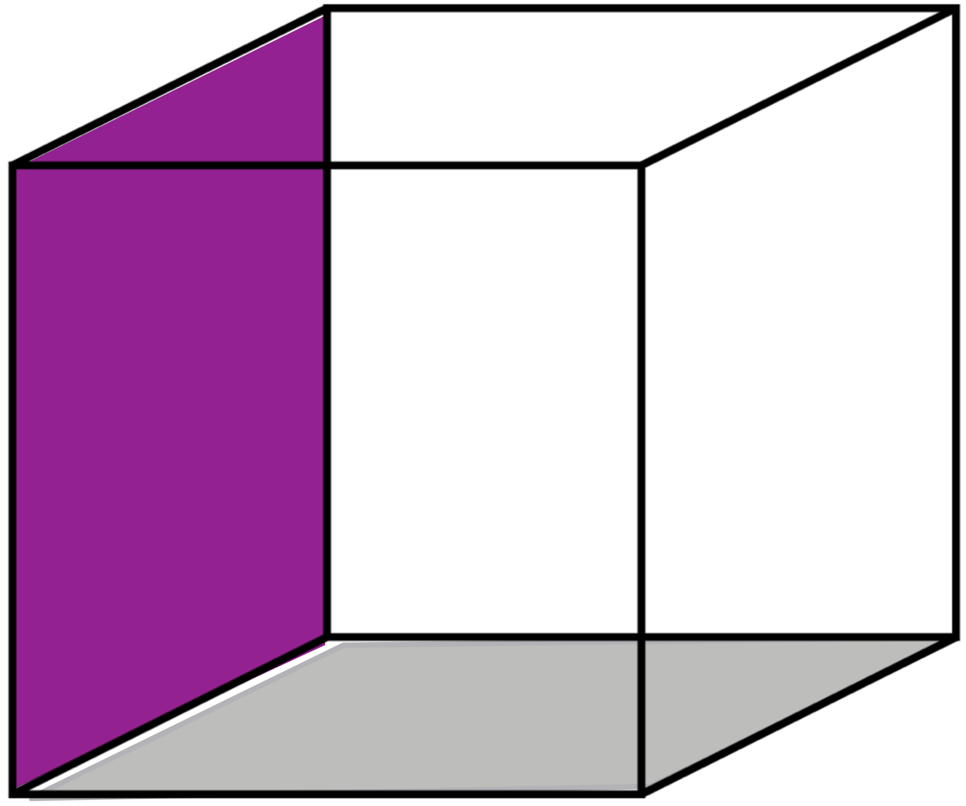
Kan filling



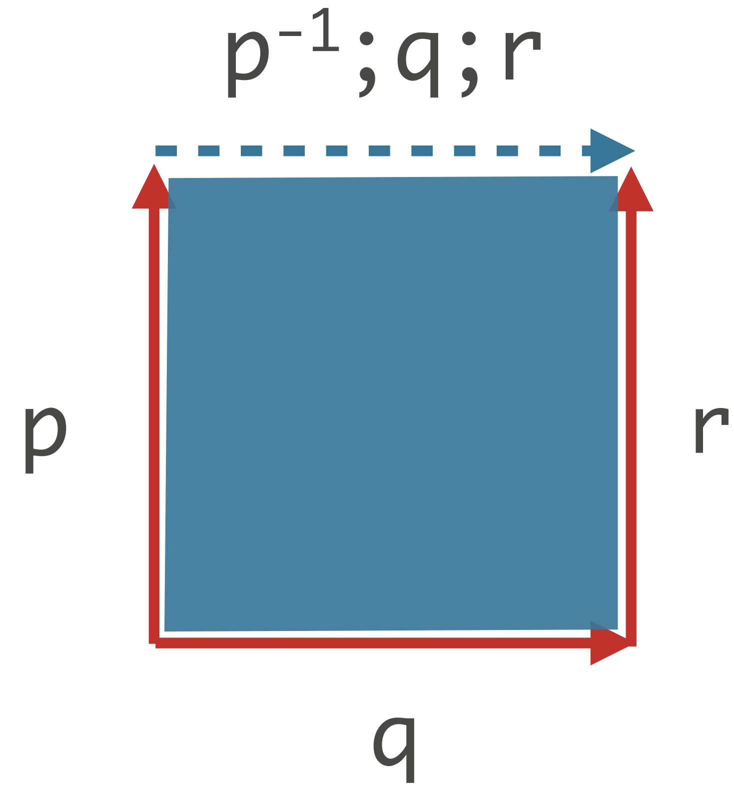


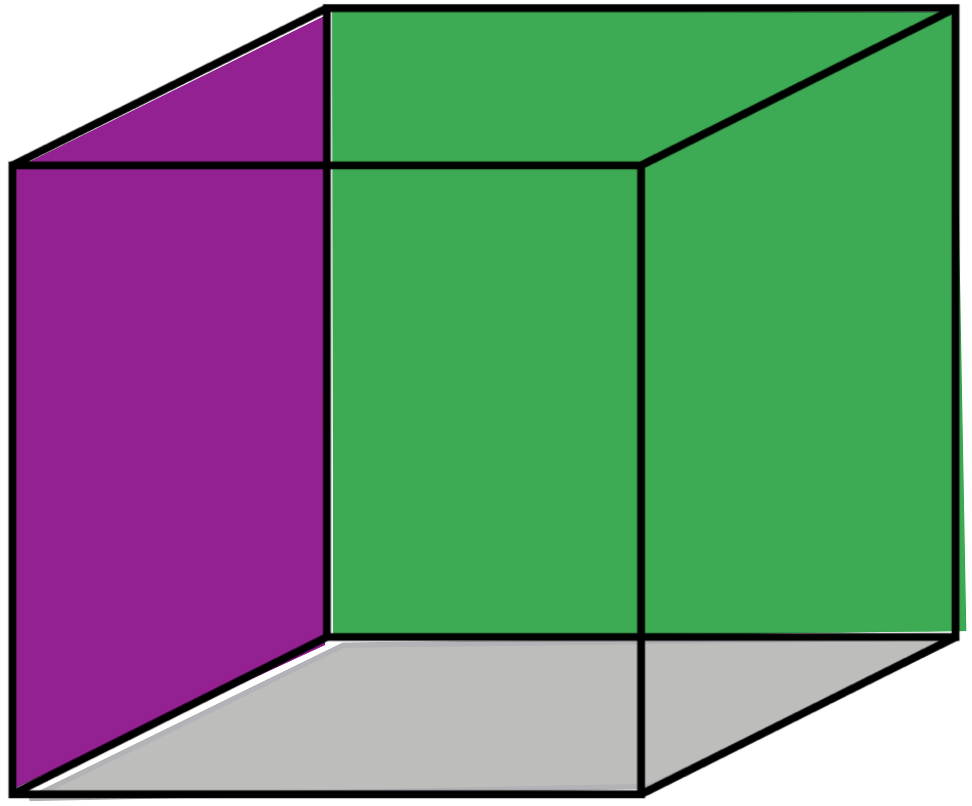
Kan filling



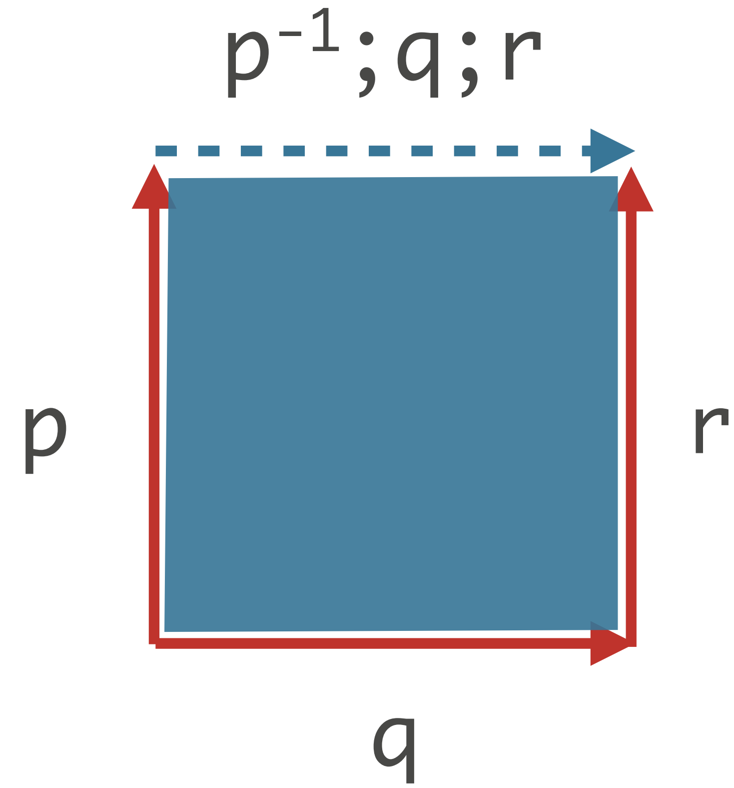


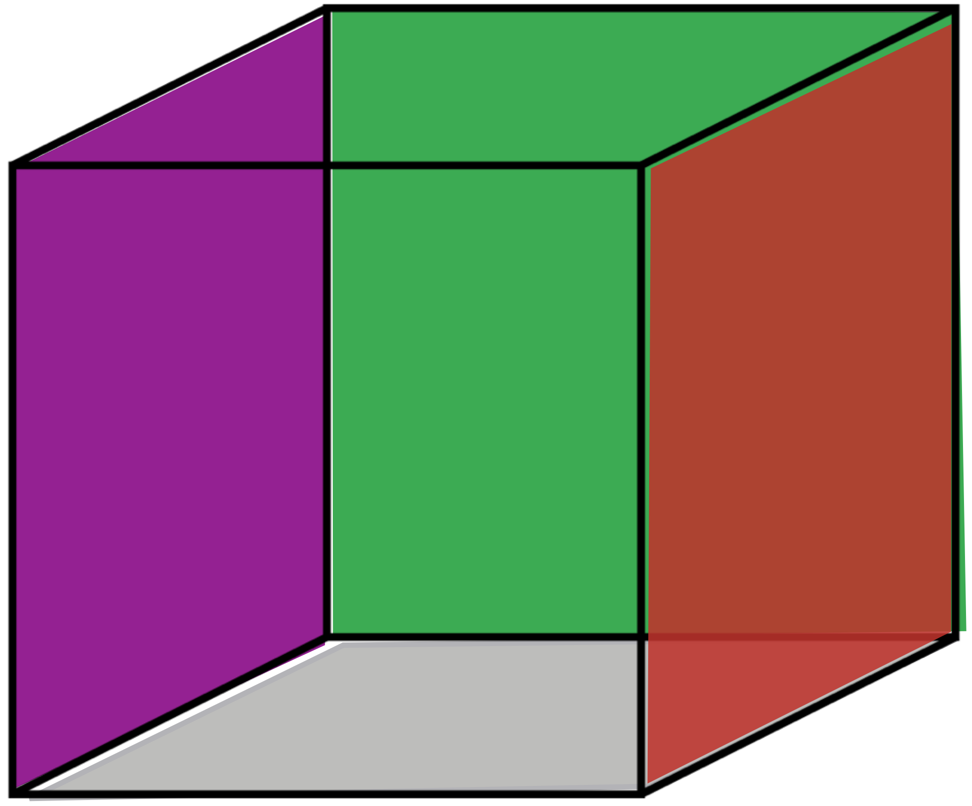
Kan filling



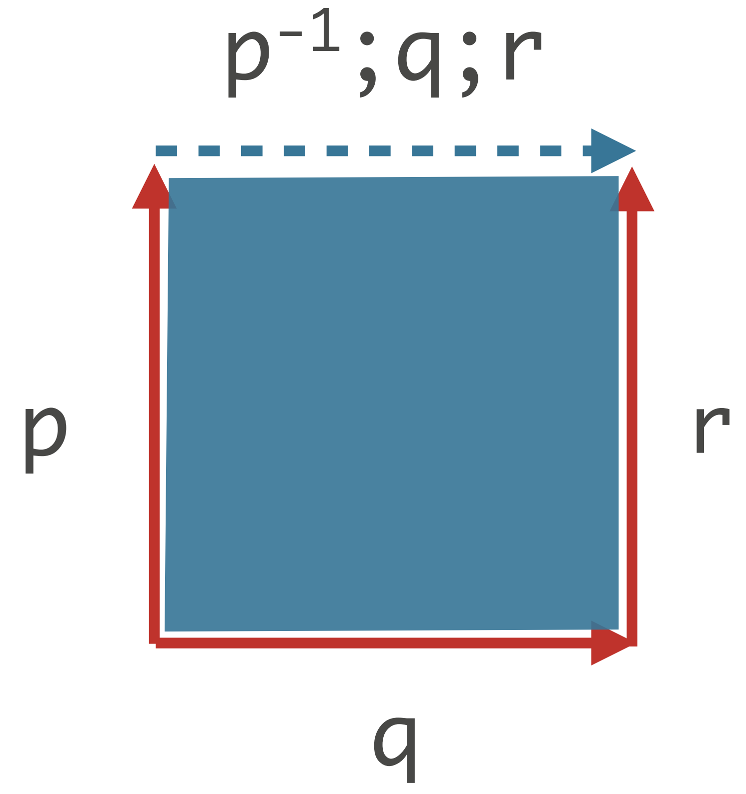


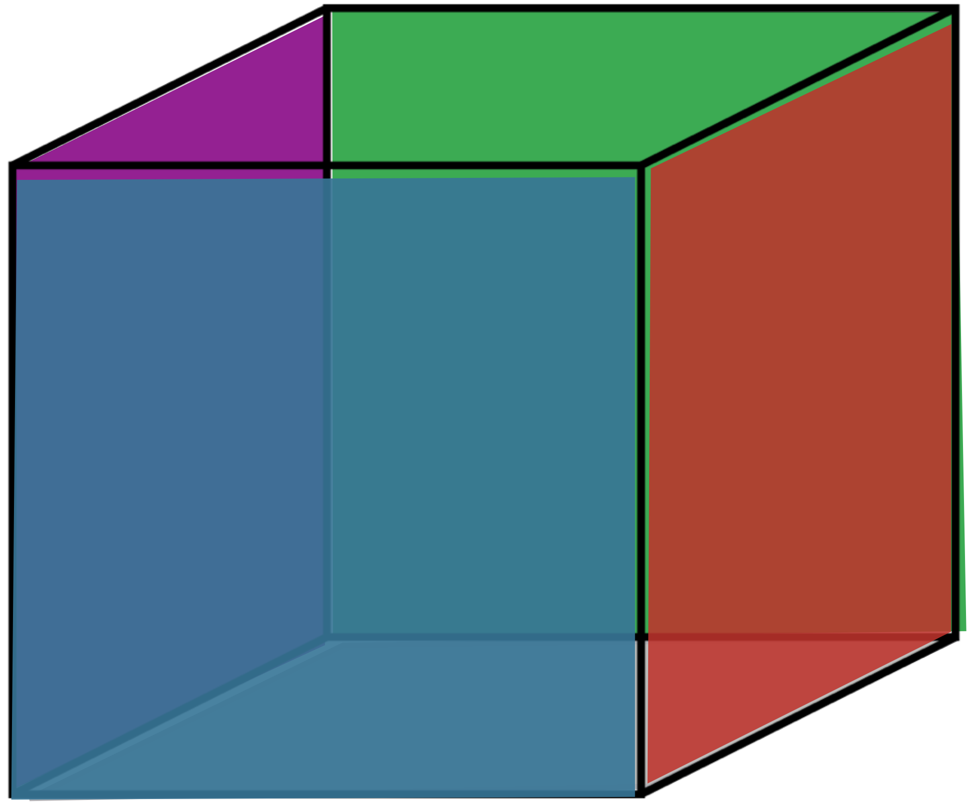
Kan filling



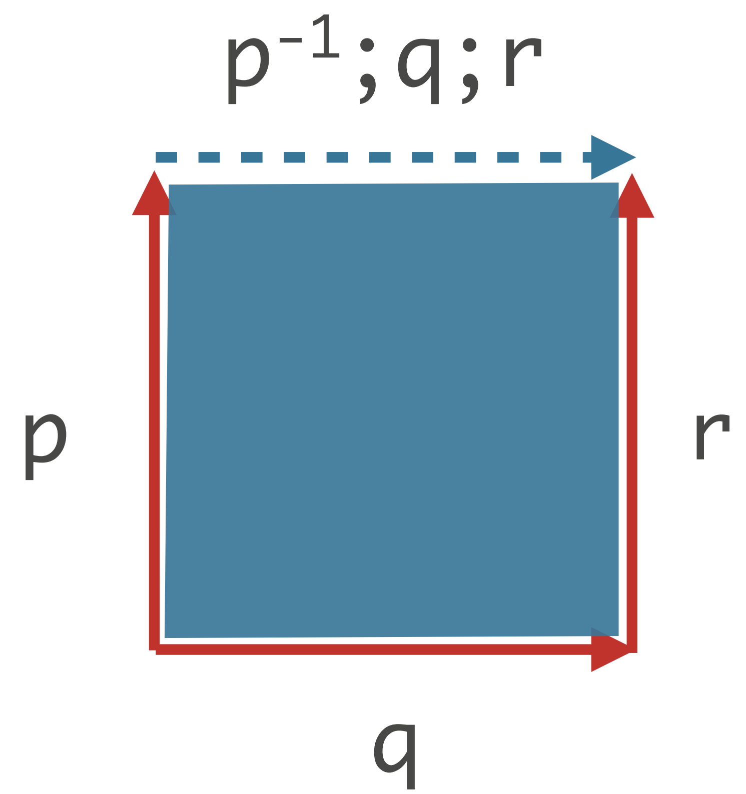


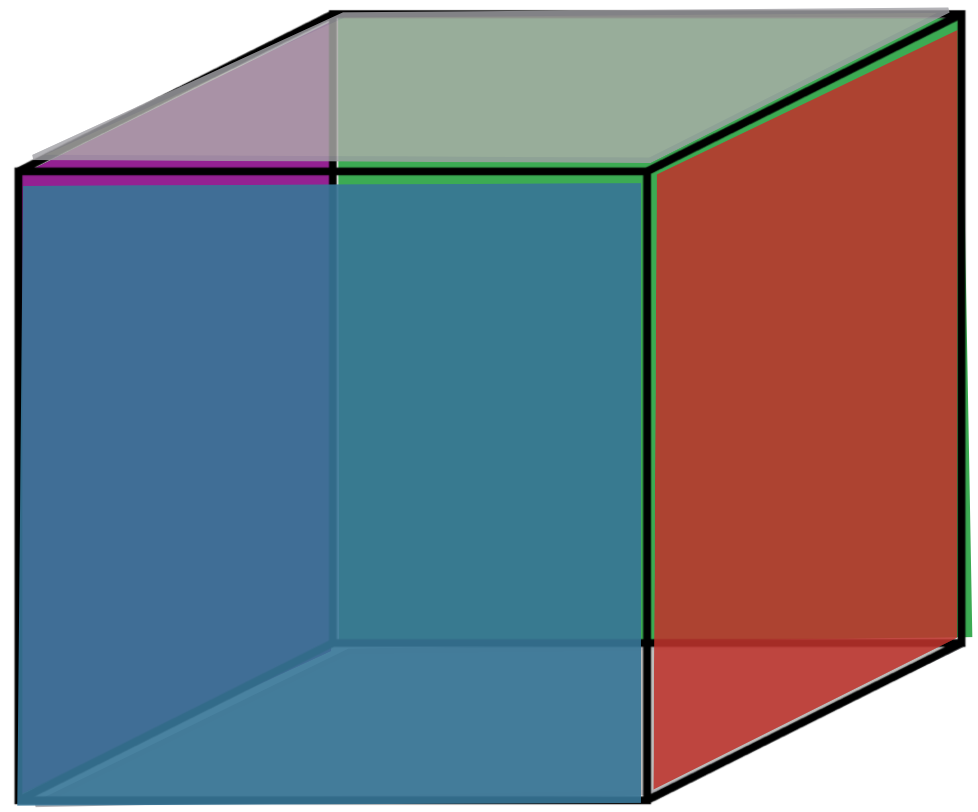
Kan filling





Kan filling





Main Ideas

Main Ideas

- $\mathbf{Sets}^{\mathbb{C}^{op}}$ for $\mathbb{C}$ free semicartesian category on $0,1 : * \to \mathbb{I}$; free cartesian category; …with connections/reversals

Main Ideas

- $Sets^{\mathbb{C}^{op}}$ for $\mathbb{C}$ free semicartesian category on $0,1 : * \to \mathbb{I}$; free cartesian category; …with connections/reversals
- fibration: algebraic/specified solutions to filling problems

Main Ideas

- $Sets^{\mathbb{C}^{op}}$ for $\mathbb{C}$ free semicartesian category on $0,1 : * \to \mathbb{I}$; free cartesian category; …with connections/reversals
- fibration: algebraic/specified solutions to filling problems
- algorithms for filling in ∏, ∑, Path, universe, univalence

Main Ideas

- $\mathbf{Sets}^{\mathbb{C}^{op}}$ for $\mathbb{C}$ free semicartesian category on $0,1 : * \to \mathbb{I}$; free cartesian category; …with connections/reversals
- fibration: algebraic/specified solutions to filling problems
- algorithms for filling in $\prod$, $\sum$, Path, universe, univalence
- definition of fibration chosen carefully — stable under change of base (uniformity), (trivial) cofibrations — in harmony with choice of cube category

Relation to sSet?

- known methods use **P** A := A(- ⊗ $\mathbb{I}$) or $A^{\mathbf{y}\mathbb{I}}$ and its right adjoint to define universes and filling in them
- unclear if any "type theoretic model structures" are Quillen-equiv to sSet/Top; some are not [Sattler]

Recommender System

https://www.uwo.ca/math/faculty/kapulkin/seminars/hottest.html

Last spring: Coquand
Angiuli

October 11: Favonia

https://www.cs.uoregon.edu/research/summerschool/summer18/topics.php

Harper

Computation with univalence in…

Computation with univalence in…

definitions of $\mathbb{Z}$

Computation with univalence in…

definitions of $\mathbb{Z}$

fundamental groups of $\mathbb{S}^1$ and $\mathbb{T}$

Computation with univalence in…

definitions of $\mathbb{Z}$

fundamental groups of $\mathbb{S}^1$ and $\mathbb{T}$

calculation of $\pi_4(\mathbb{S}^3)$

Running the equivalence principle

ℤ in type theory (1)

`nat + nat`

negative **non-negative**

-2	-1	0	1	2
●	●	●	●	●
`inl 1`	`inl 0`	`inr 0`	`inr 1`	`inr 2`

$\mathbb{Z}$ in type theory (2)

`nat` $+_{(0,0)}$ `nat`

non-positive **non-negative**

-2	-1	0	1	2
●	●	●	●	●
`inl 2`	`inl 1`	`inr 0`	`inr 1`	`inr 2`
		`inl 0`		

$\mathbb{Z}$ in type theory (3)

(nat × nat) / (a+b’ $=_{nat}$ a’+b)

-1	0	1
●	●	●
(0,1)	(0,0)	(1,0)
(1,2)	(1,1)	(2,1)

$\mathbb{Z}$ in type theory (4)

```
free (set-level) group on one generator
```

```
     -1            0            1
      ●            ●            ●
pred(zero)       zero      suc(zero)
```

```
pred(succ(zero))
succ(pred(zero))
```

$\mathbb{Z}$ in type theory (5)

loops in S^1

-1	0	1	2
●	●	●	●
$\texttt{loop}^{-1}$	id	loop	loop;loop

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

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

addition (1)

```
addZ : Z -> Z -> Z = split
  inl neg_a -> split@(Z -> Z) with
                 inl neg_b -> inl(suc(add neg_a neg_b))
                 inr nonneg_b -> sub nonneg_b (suc neg_a)
  inr nonneg_a -> split@(Z -> Z) with
                   inl neg_b -> sub nonneg_a (suc neg_b)
                   inr nonneg_b -> inr(add nonneg_a nonneg_b)
```

addition (1)

```
addZ : Z -> Z -> Z = split
  inl neg_a -> split@(Z -> Z) with
                  inl neg_b -> inl(suc(add neg_a neg_b))
                  inr nonneg_b -> sub nonneg_b (suc neg_a)
  inr nonneg_a -> split@(Z -> Z) with
                    inl neg_b -> sub nonneg_a (suc neg_b)
                    inr nonneg_b -> inr(add nonneg_a nonneg_b)
```

-1-a + -1-b = -2-(a+b)

inl(a) + inl(b) = inl(1+a+b)

addition (1)

```
addZ : Z -> Z -> Z = split
  inl neg_a -> split@(Z -> Z) with
                  inl neg_b -> inl(suc(add neg_a neg_b))
                  inr nonneg_b -> sub nonneg_b (suc neg_a)
  inr nonneg_a -> split@(Z -> Z) with
                    inl neg_b -> sub nonneg_a (suc neg_b)
                    inr nonneg_b -> inr(add nonneg_a nonneg_b)
```

$$-1-a + b = (b - (1+a))$$

$$sub : nat \times nat \to Z$$

addition (1)

```
addZ : Z -> Z -> Z = split
  inl neg_a -> split@(Z -> Z) with
                 inl neg_b -> inl(suc(add neg_a neg_b))
                 inr nonneg_b -> sub nonneg_b (suc neg_a)
  inr nonneg_a -> split@(Z -> Z) with
                   inl neg_b -> sub nonneg_a (suc neg_b)
                   inr nonneg_b -> inr(add nonneg_a nonneg_b)
```

$$-1-a + b = (b - (1+a))$$

$$\mathrm{sub} : \mathrm{nat} \times \mathrm{nat} \to Z$$

addition (3)

```
add ((a,b),(a',b')) = (a+a',b+b')
```

plus proof that respects quotient

assoc:

$$((a_1,b_1)+(a_2,b_2))+(a_3,b_3)$$
$$= ((a_1+a_2)+a_3,(b_1+b_2)+b_3)$$
$$= (a_1+(a_2+a_3),b_1+(b_2+b_3))$$
$$= (a_1,b_1)+((a_2,b_2)+(a_3,b_3))$$

Equivalence of (1) and (3)

```
from : Zd -> Z = split
  diff a b -> sub a b
  quot a b a' b' q @ x -> q @ x
  setZ i j p q @ x y ->
    ZSet (from i) (from j) (<x> from (p @ x))
         (<x> from (q @ x)) @ x @ y

to : Z -> Zd = split
  inl n -> diff zero (suc n)
  inr n -> diff n zero
```

plus proof mutually inverse

Using univalence

```
ZisZd : Path U Z Zd =
  isoPath Z Zd to from fromto tofrom
```

Therefore:
any construction on types
that can be defined for `Zd`
can be transferred to `Z`**, and vice versa**

Group structure

```
data GroupStr (X : U) =
  groupstr (op : X -> X -> X)
           (unit : X)
           (inv : X -> X)
           (unitl : (x : X) -> Path X (op unit x) x)
           (unitr : (x : X) -> Path X (op x unit) x)
           (assoc : (x y z : X) ->
                    Path X (op (op x y) z) (op x (op y z)))
           (invl  : (x : X) -> Path X (op (inv x) x) unit)
           (invr  : (x : X) -> Path X (op x (inv x)) unit)
```

Without univalence

`Given e : A ≃ B`

Without univalence

```
Given e : A ≃ B

GroupStr : U → U
```

Without univalence

`Given e : A ≃ B`

`GroupStr : U → U`

define `GroupStr A ≃ GroupStr B`

Without univalence

Given $e : A \simeq B$

$\text{GroupStr} : U \to U$

define $\text{GroupStr}\ A \simeq \text{GroupStr}\ B$

e.g. $b_1 \odot_B b_2 = e(e^{-1}(b_1) \odot_A e^{-1}(b_2))$

Without univalence

`Given e : A ≃ B`

`GroupStr : U → U`

define `GroupStr A ≃ GroupStr B`

e.g. $b_1 \odot_B b_2 = e(e^{-1}(b_1) \odot_A e^{-1}(b_2))$

No definable construction on types differentiates equivalent types

Using univalence

```
Z≃Zd : Path U Z Zd
```
univalence

Using univalence

```
Z≃Zd : Path U Z Zd
```
univalence

```
GroupStr : U → U
```

Using univalence

`Z≃Zd : Path U Z Zd` **univalence**

`GroupStr : U → U`

define `GroupStr Zd`

Using univalence

`Z≃Zd : Path U Z Zd` **univalence**

`GroupStr : U → U`

define `GroupStr Zd`

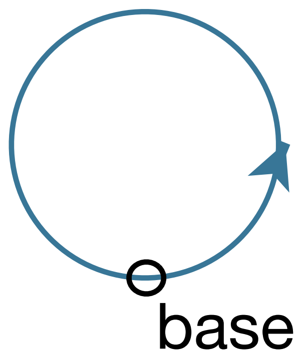
mechanically derive `GroupStr Z`
by transporting along the equivalence

intdiff.ctt

Higher inductive types and synthetic homotopy theory

Circle

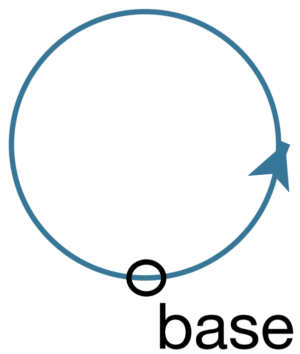
Circle S^1 is a **higher inductive type** generated by



Circle

Circle $\mathbb{S}^1$ is a **higher inductive type** generated by

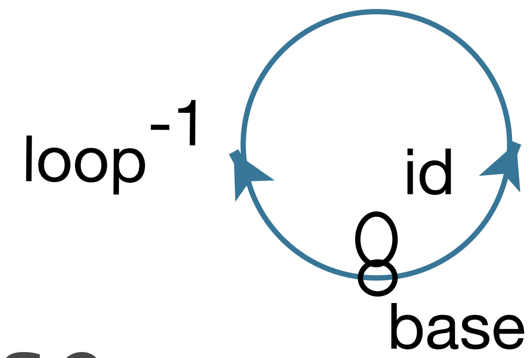
```
base : 𝕊¹
loop : Path 𝕊¹ base base
```



Circle

Circle $\mathbb{S}^1$ is a **higher inductive type** generated by

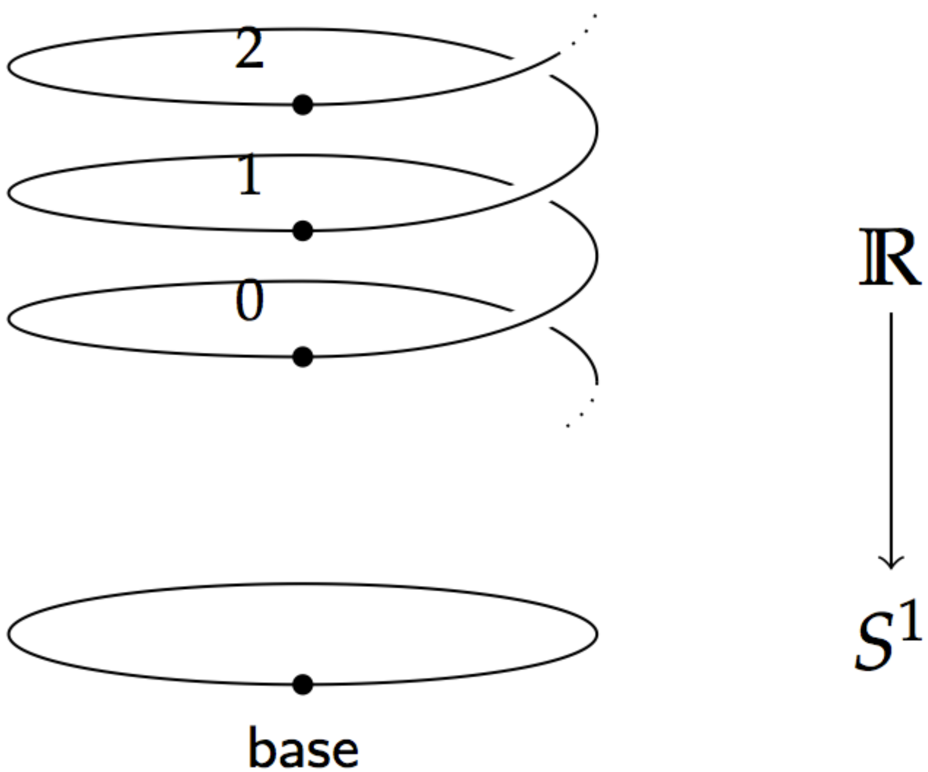
```
base : 𝕊¹
loop : Path 𝕊¹ base base
```



Free type (∞-groupoid/uniform Kan cubical set):

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

Universal Cover



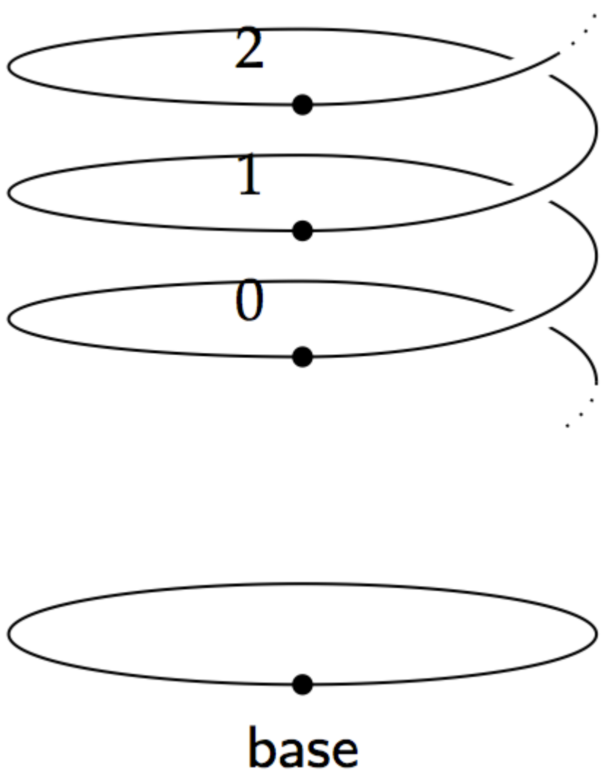
`wind : Ω(S¹) → ℤ`

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

lifting `loop` adds 1

lifting `loop`$^{-1}$ subtracts 1

Universal Cover



$$\mathbb{R} \downarrow S^1$$

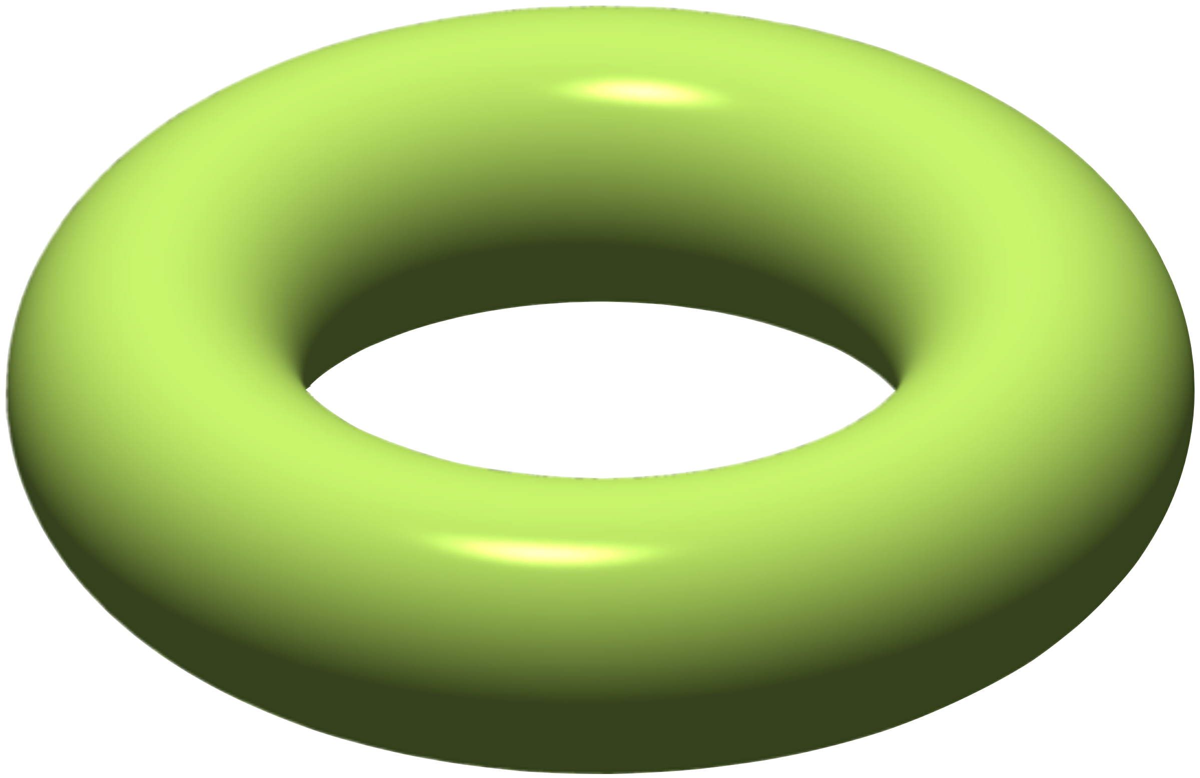
```
Helix : S¹ → U
Helix(base) := ℤ
Helix(loop) :=
  ua(x ⟼ x+1 : ℤ ≃ ℤ)
```

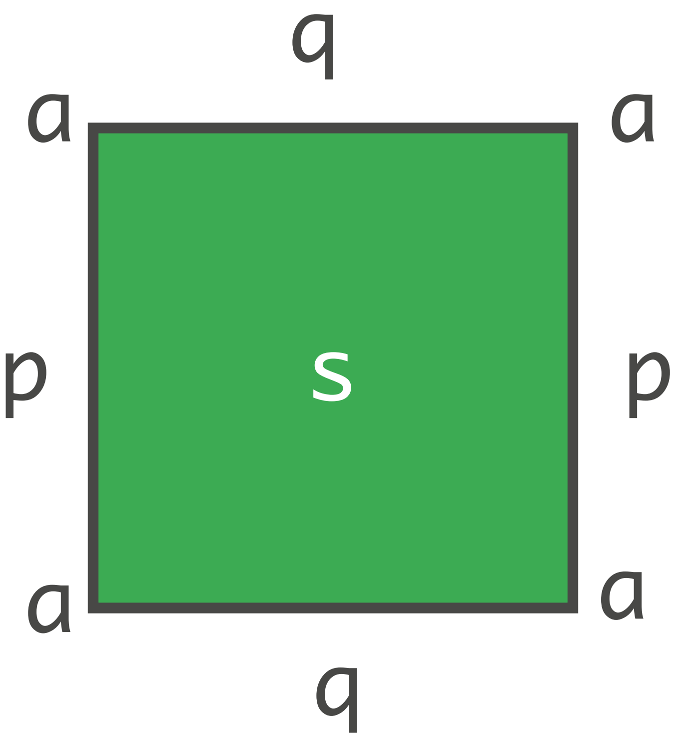
lifting `loop` adds 1

lifting `loop`$^{-1}$ subtracts 1

circletalk.ctt

Torus






```
a   : Torus
p,q : Path a a
s   : Square q q p p
```

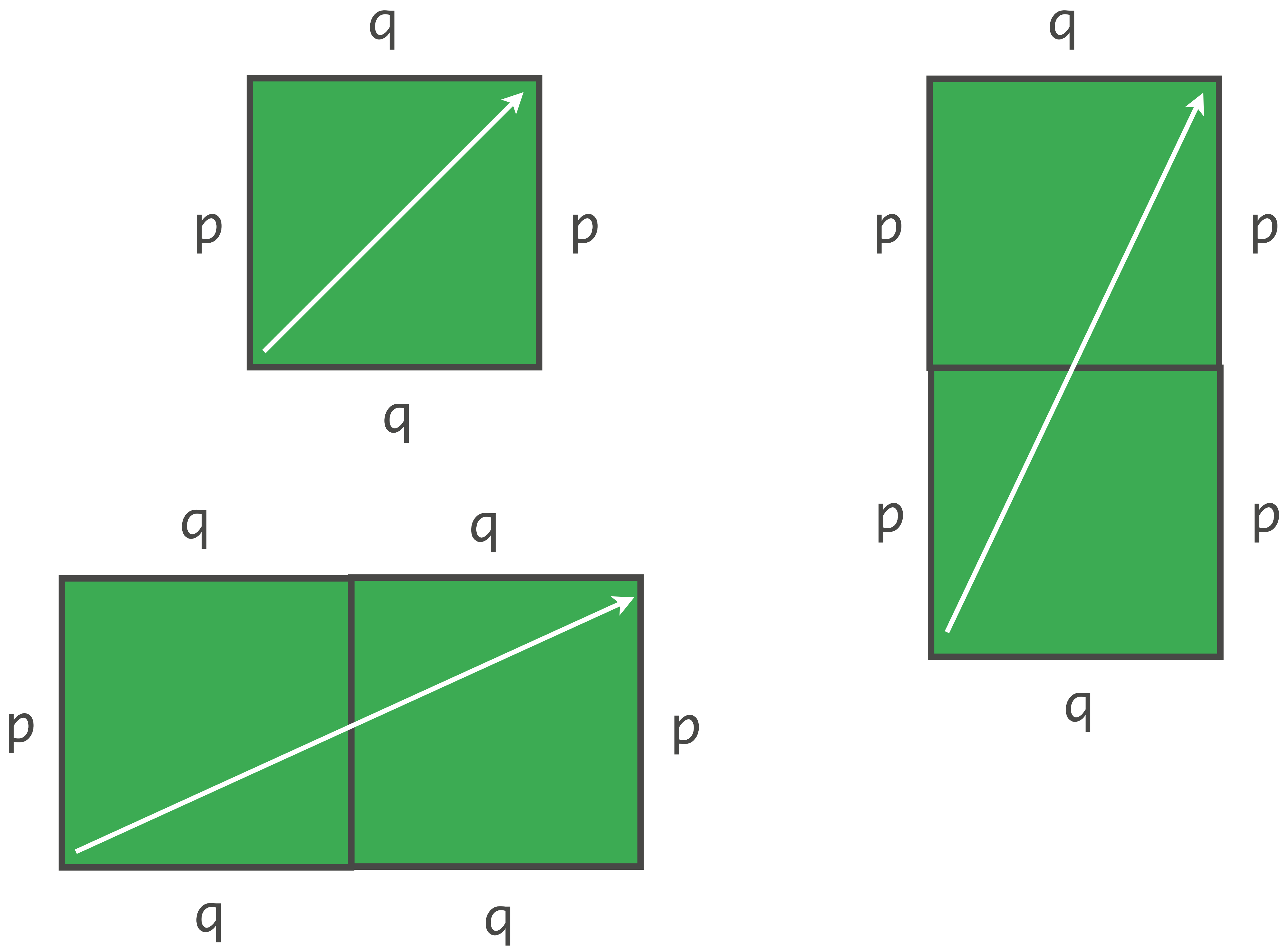
$$\mathbb{T} \simeq \mathbb{S}^1 \times \mathbb{S}^1$$

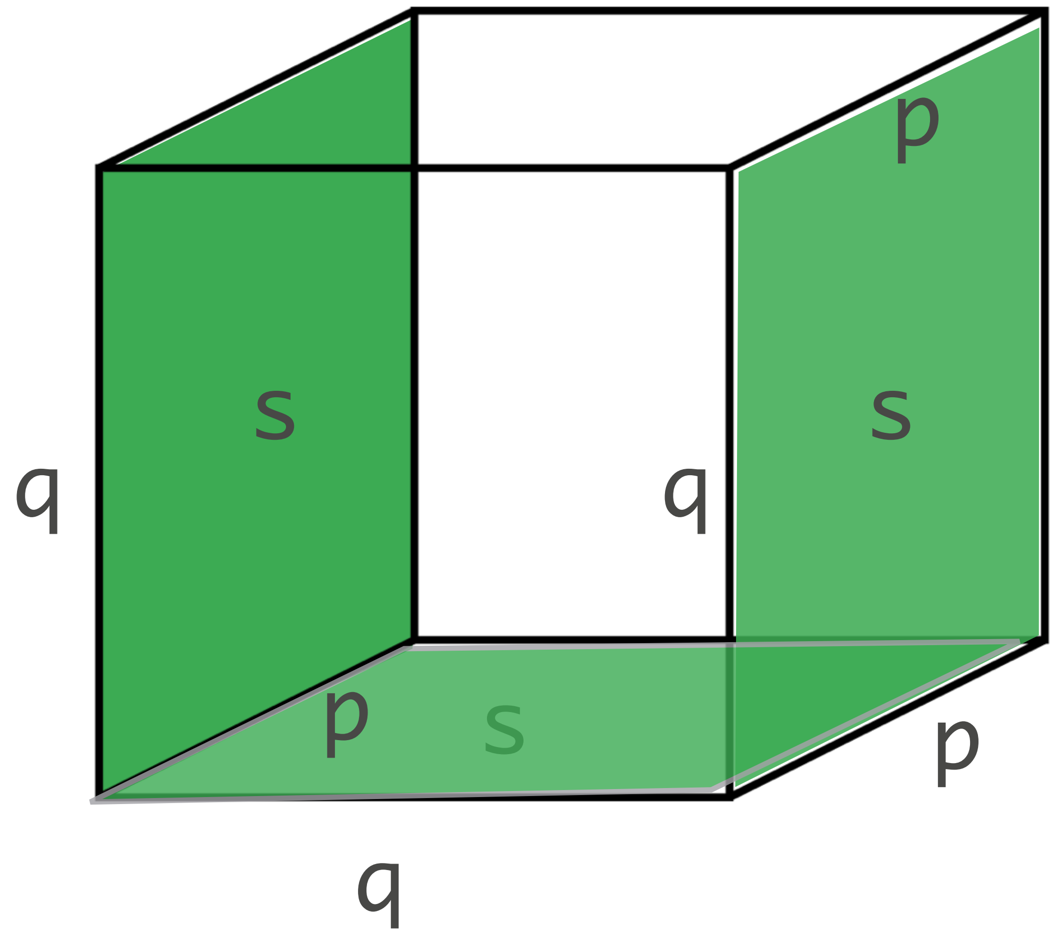
```
t2c : Torus -> and S1 S1 = split
  a -> (base,base)
  p @ y -> (loop @ y, base)
  q @ x -> (base, loop @ x)
  s @ x y -> (loop @ y, loop @ x)
```

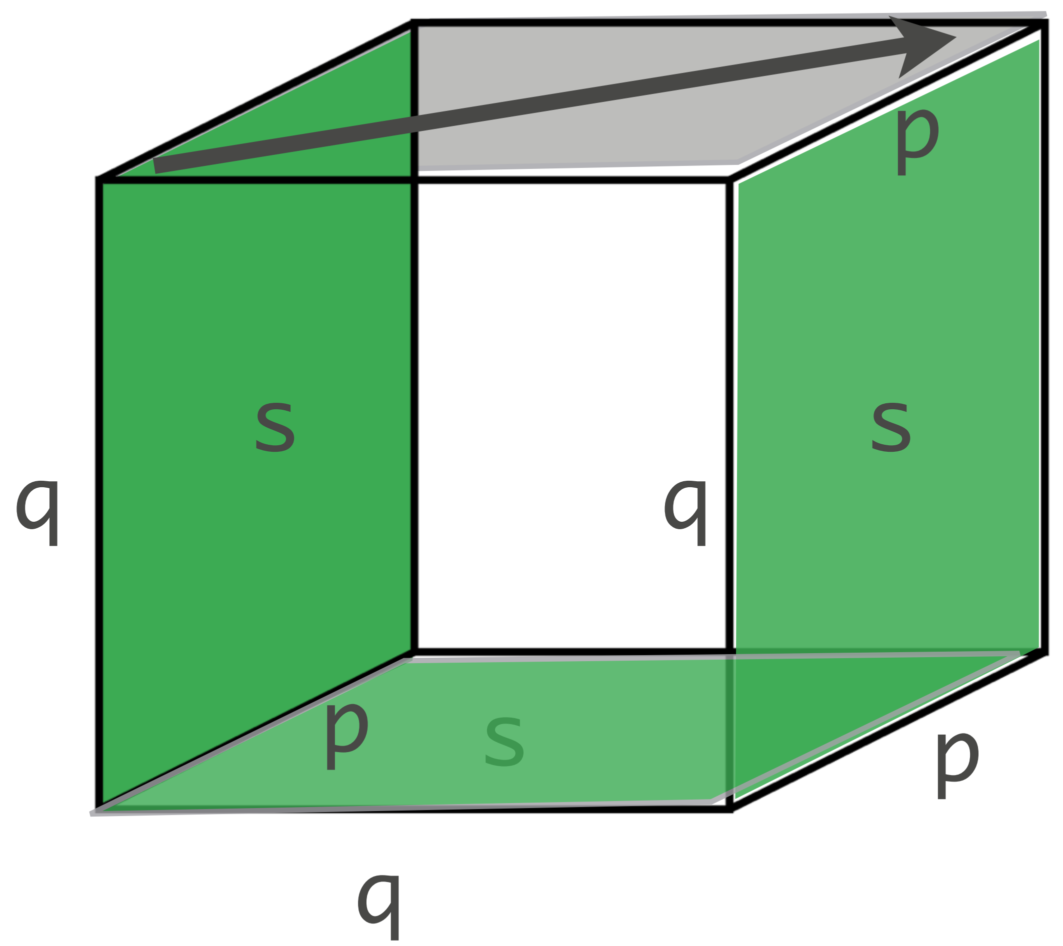
free type: suffices to specify images of generators

$$\Omega(\mathbb{T}) \to \Omega(\mathbb{S}^1 \times \mathbb{S}^1) \to \Omega(\mathbb{S}^1) \times \Omega(\mathbb{S}^1)$$

```
count (t : OmegaT) : (and Z Z) =
  (winding (<x> (t2c (t@x)).1) ,
   winding (<x> (t2c (t@x)).2))
```

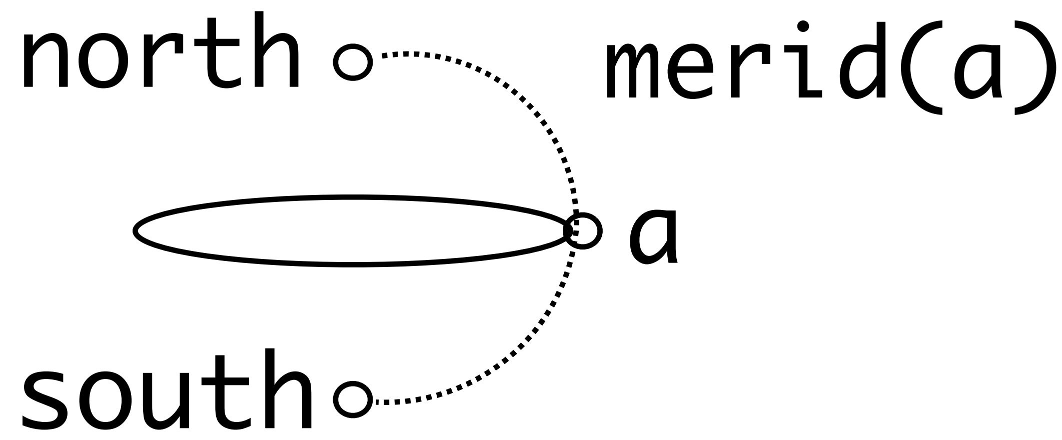

q
p
p
q
q
p
p
p
p
q
q
q
p
p
q
q


p
s
q
s
q
p
s
p
q


p
s
s
q
q
p
s
p
q

torustalk.ctt

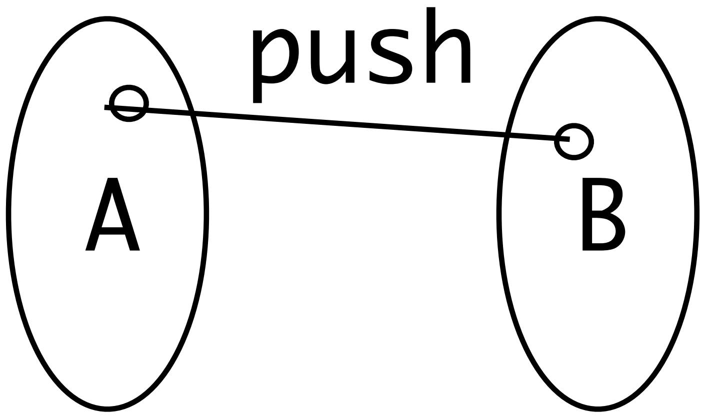
Suspension




```
data susp (A : U) = north
                  | south
                  | merid (a : A)
                    <i> [ (i=0) -> north
                        , (i=1) -> south ]

-- n-spheres
sphere : nat -> U = split
  zero  -> bool
  suc n -> susp (sphere n)
```

Join




```
data join (A B : U) = inl (a : A)
                    | inr (b : B)
                    | push (a : A) (b : B)
                       <i> [ (i = 0) -> inl a
                           , (i = 1) -> inr b ]
```

Synth homotopy theory

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

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

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

(Co)homology

Mayer-Vietoris

[Brunerie, Cavallo, Favonia, Finster, Licata, Lumsdaine, Sojakova, Shulman]

Synth homotopy theory

Serre Spectral Sequence [Avigad, Awodey, Buchholtz, van Doorn, Newstead, Rijke, Shulman]

Cellular Cohomology [Favonia, Buchholtz]

Higher Groups [Buchholtz, van Doorn, Rijke]

Cayley-Dickson, Quaternionic Hopf [Buchholtz, Rijke]

Real projective spaces [Buchholtz, Rijke]

Free Higher Groups [Kraus, Altenkirch]

Brunerie's number

Constructive proof in type theory of:

There exists a k such that $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/k\mathbb{Z}$

$$\Omega^3\mathbb{S}^3 \xrightarrow{\Omega^3 e} \Omega^3(\mathbb{S}^1 * \mathbb{S}^1) \xrightarrow{\Omega^3 \alpha} \Omega^3\mathbb{S}^2 \longrightarrow \mathbb{Z}$$

Brunerie's number

Constructive proof in type theory of:

There exists a k such that $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/k\mathbb{Z}$

generator of $\pi_3(\mathbb{S}^3)$

$$\Omega^3\mathbb{S}^3 \xrightarrow{\Omega^3 e} \Omega^3(\mathbb{S}^1 * \mathbb{S}^1) \xrightarrow{\Omega^3 \alpha} \Omega^3\mathbb{S}^2 \longrightarrow \mathbb{Z}$$

Brunerie's number

Constructive proof in type theory of:

There exists a k such that $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/k\mathbb{Z}$

generator of $\pi_3(\mathbb{S}^3)$

$$\Omega^3\mathbb{S}^3 \xrightarrow{\Omega^3 e} \Omega^3(\mathbb{S}^1 * \mathbb{S}^1) \xrightarrow{\Omega^3 \alpha} \Omega^3\mathbb{S}^2 \longrightarrow \mathbb{Z}$$

view $\mathbb{S}^3$ as join

Brunerie's number

Constructive proof in type theory of:

There exists a k such that $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/k\mathbb{Z}$

generator of $\pi_3(\mathbb{S}^3)$

$$\Omega^3\mathbb{S}^3 \xrightarrow{\Omega^3 e} \Omega^3(\mathbb{S}^1 * \mathbb{S}^1) \xrightarrow{\Omega^3 \alpha} \Omega^3\mathbb{S}^2 \longrightarrow \mathbb{Z}$$

view $\mathbb{S}^3$ as join

main map

Brunerie's number

Constructive proof in type theory of:

There exists a k such that $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/k\mathbb{Z}$

generator of $\pi_3(\mathbb{S}^3)$

$$\Omega^3\mathbb{S}^3 \xrightarrow{\Omega^3 e} \Omega^3(\mathbb{S}^1 * \mathbb{S}^1) \xrightarrow{\Omega^3 \alpha} \Omega^3\mathbb{S}^2 \longrightarrow \mathbb{Z}$$

view $\mathbb{S}^3$ as join

main map

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

Proof Assistants

cubicaltt [Cohen,Coquand,Huber,Mörtberg]

cubical Agda [Vezzosi]

redtt [Angiuli,Cavallo,Favonia,Harper,Mörtberg,Sterling]

yacctt [Angiuli,Mörtberg]

redprl [Angiuli,Cavallo,Favonia,Harper,Sterling]

- different cube categories, filling operations
- optimized definitions of filling operations
- term representations, evaluation strategies, def. equality
- non-fibrant types and exact equalities

CS Applications

- guarded recursion [Birkedal,Bizjak,Clouston,Spitters,Vezzosi]
- relational parametricity [Bernardy,Coquand,Moulin; Nuyts,Vezzosi,Devriese]
- effects in computational cubical type theory? [Angiuli,Cavallo,Favonia,Harper,Sterling,Wilson]
- transporting along functions in directed type theories? [Riehl,Shulman;Riehl,Sattler;L.,Weaver]

Questions

- homotopy theories of cubical sets models? [Sattler;Kapulkin,Voevodsky'18]
- interpret cubical type theory in other models?
- homotopy canonicity for MLTT+ua?
- Path U A B definitionally equal to Equiv A B? [Altenkrich,Kaposi]
- regularity? $A^{\mathbb{I}}$ + transport id def. equal to id [Awodey]

Thanks!