

Programming and Proving in Homotopy Type Theory

Daniel R. Licata

**Carnegie Mellon University &
Institute for Advanced Study**



Kepler Conjecture (1611)

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



Hales' proof (1998)

- * Reduces Kepler Conjecture to proving that a function has a lower bound on 5,000 different configurations of spheres
- * This requires solving 100,000 linear programming problems
- * 1998 submission:
 - 300 pages of math
 - + **50,000 LOC (revised 2006: 15,000 LOC)**

Proofs can be hard to check

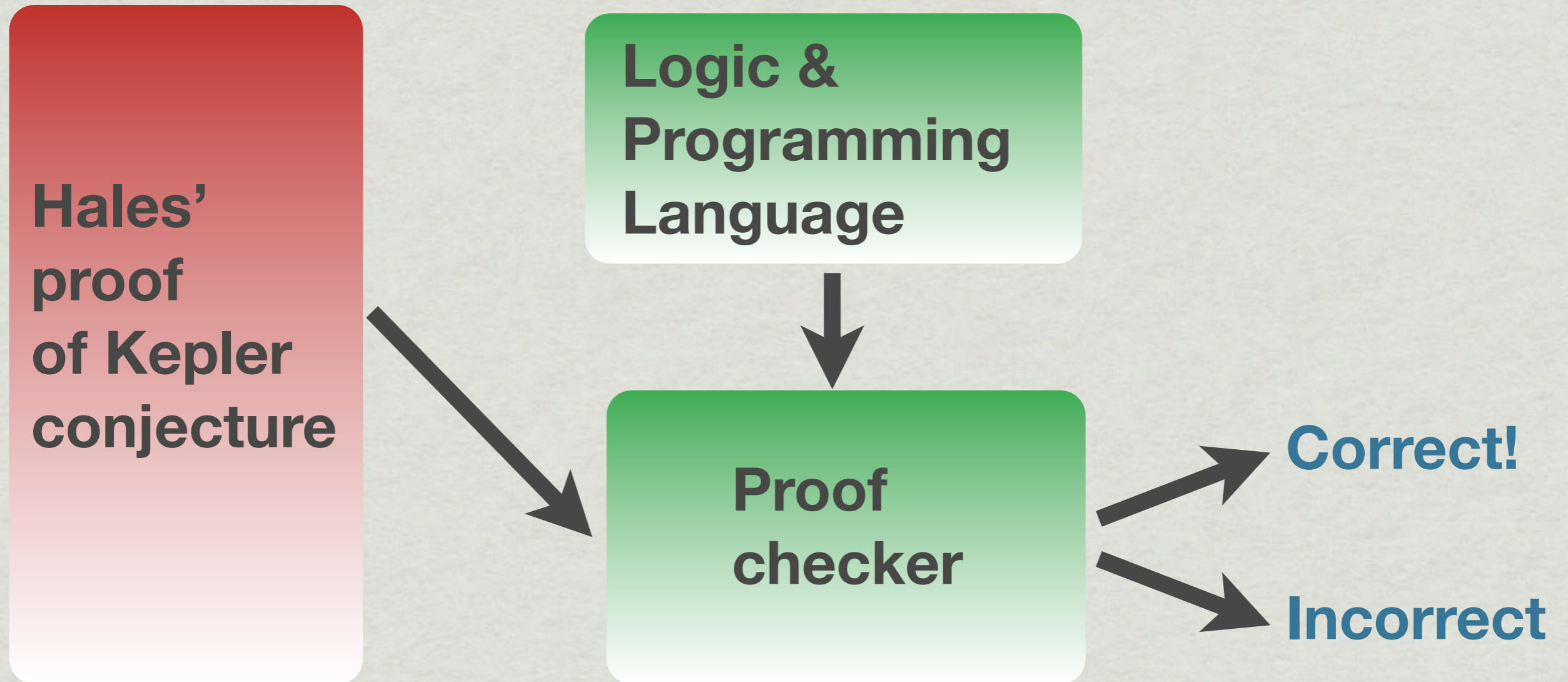
In 2003, after 4 years' work,
12 referees had checked lots of lemmas,
but gave up on verifying the proof

Proofs can be hard to check

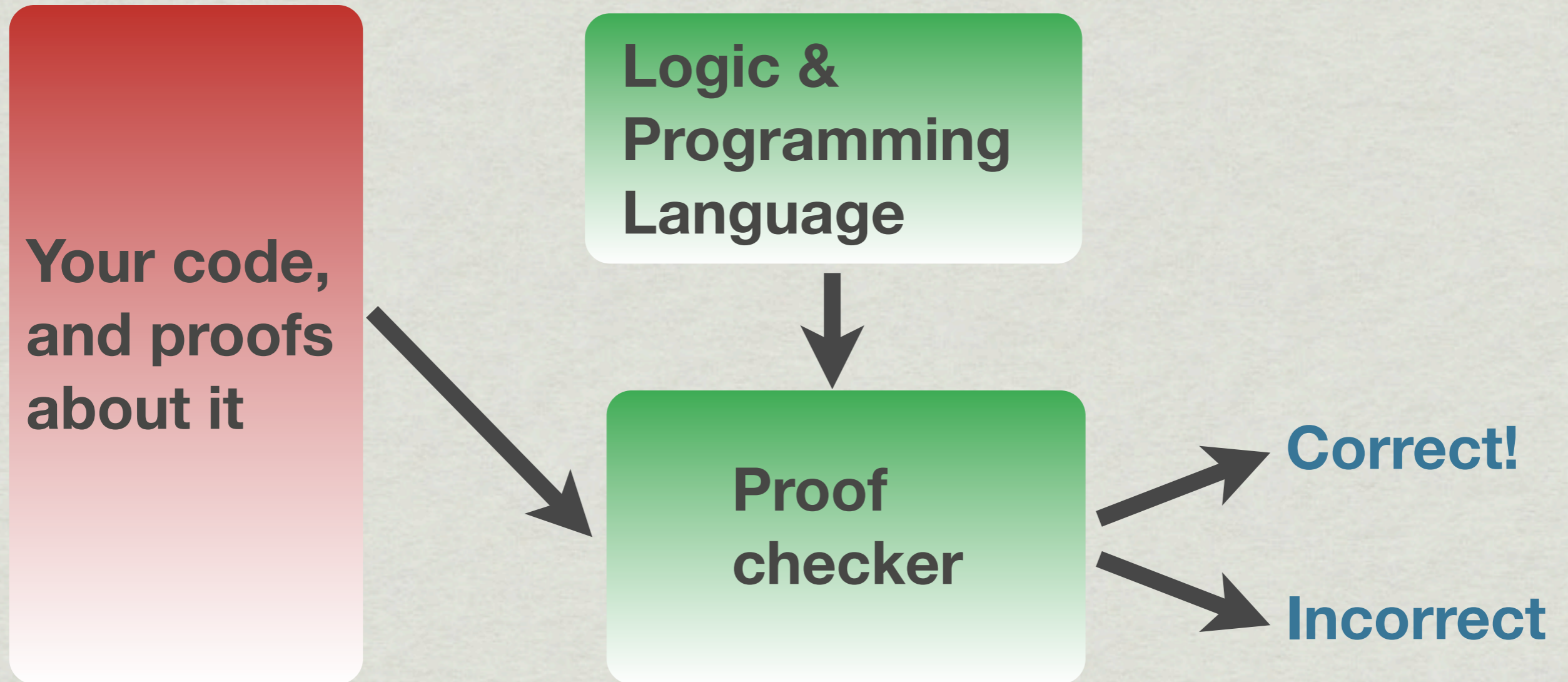
In 2003, after 4 years' work,
12 referees had checked lots of lemmas,
but gave up on verifying the proof

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

Computer-checked math



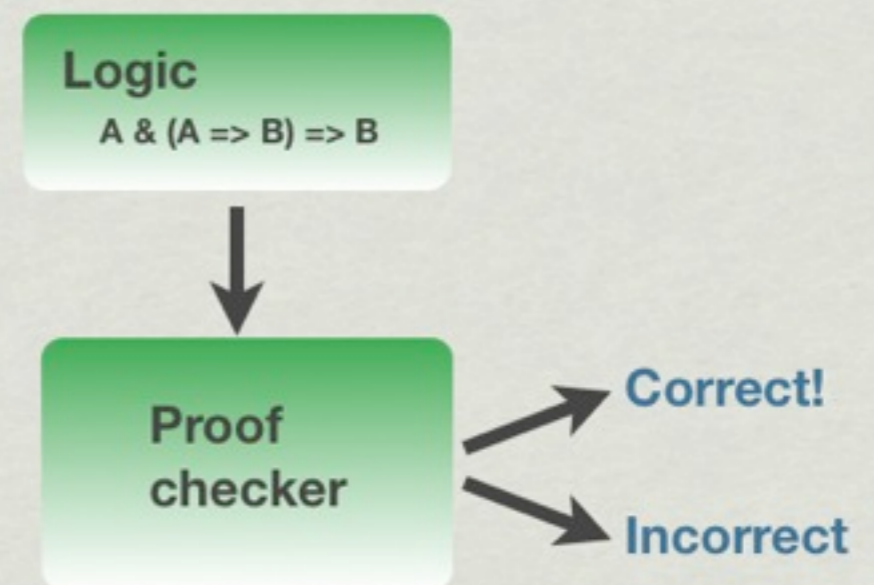
Computer-checked software



Computer-assisted proofs

Proof assistant

- Interactive proof editor
- Automated proofs
- Libraries



Computer-assisted proofs

- * are much easier to believe:
computer does the journal reviewing
- * can use computational methods
and still be fully rigorous
- * broaden access:
computer as gifted&talented teacher
- * are easier to write?

Kepler proof (85% done)

Informal

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

Computer-checked

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

Kepler proof (85% done)

Informal

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

Computer-checked

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

Kepler proof (85% done)

Informal

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

Computer-checked

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

Kepler proof (85% done)

Informal

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

Computer-checked

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

We have some work to do!

Now's the time

Recent successes:

- * Kepler conjecture [2013?, HOL Light]
- * Four-color theorem [2005, Coq]
- * Feit-Thompson theorem [2012, Coq]
- * Correctness of a C compiler [2006, Coq]
- * Correctness of Standard ML [2009, Twelf]

Mathematicians are interested!

- * Year-long program at IAS hosted by Voevodsky

Making better proof assistants

PL: languages for expressing mathematics

SE: managing large codebases

Compilers + distributed computing: speed

Machine learning: automated proof search

HCI: usable by working mathematicians

Graphics: visualization

Making better proof assistants

PL: languages for expressing mathematics

SE: managing large codebases

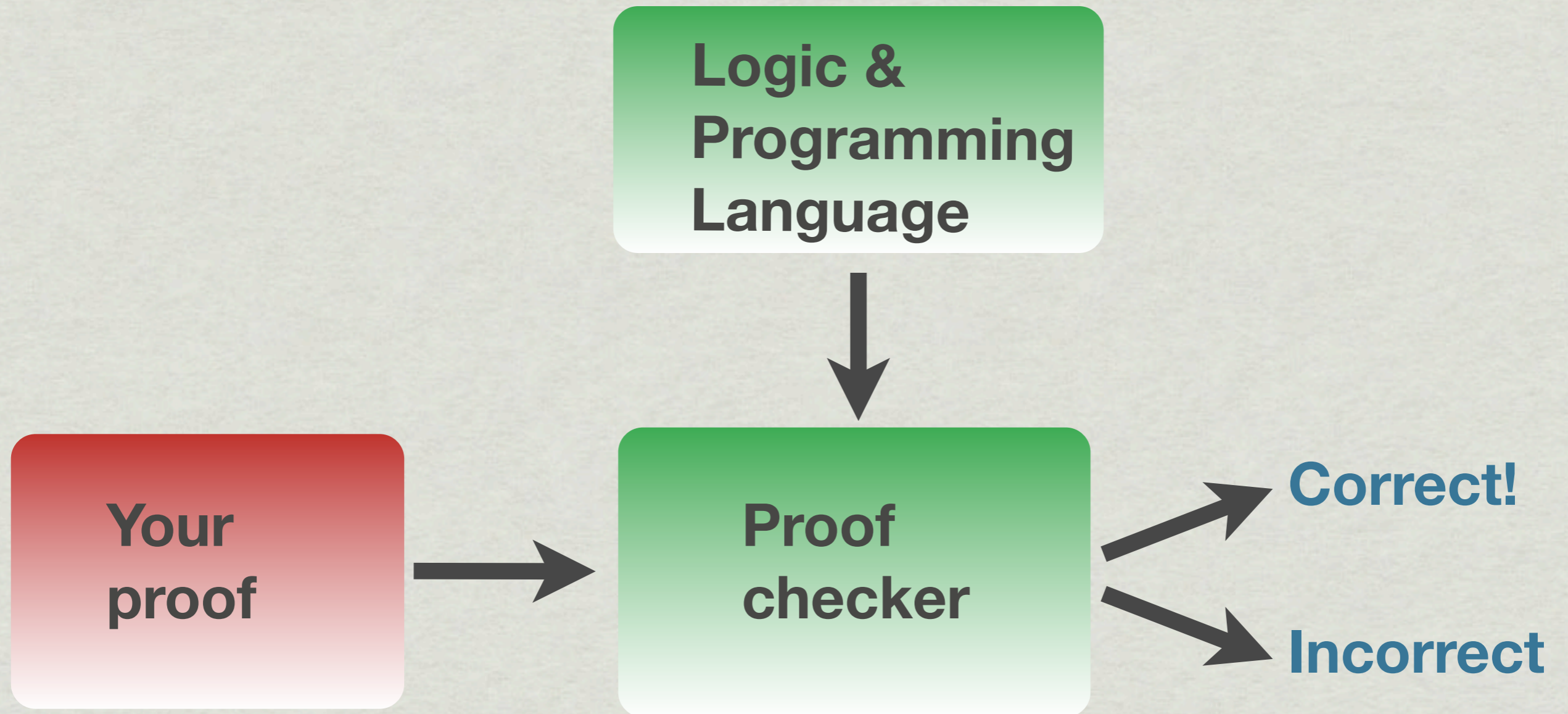
Compilers + distributed computing: speed

Machine learning: automated proof search

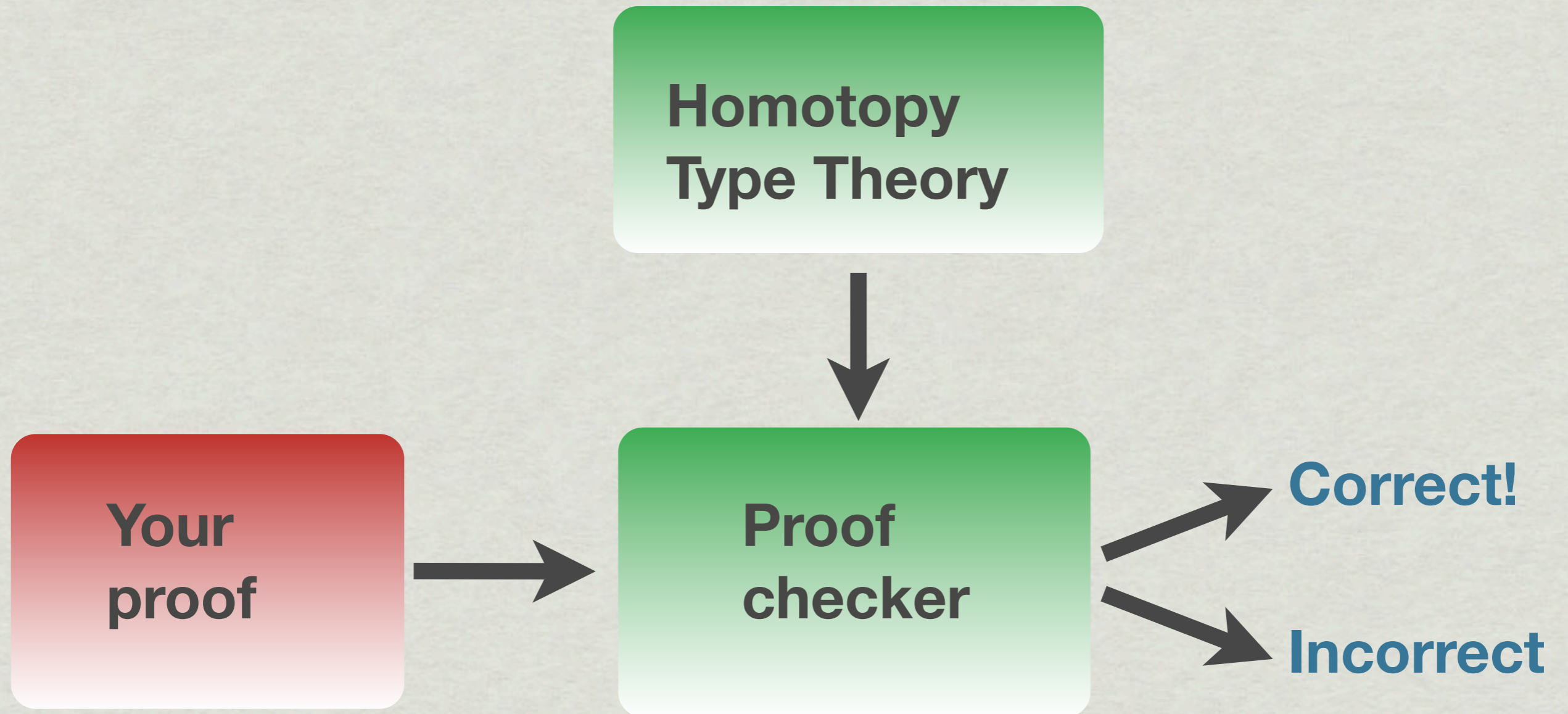
HCI: usable by “working mathematicians”

Graphics: visualization

Homotopy Type Theory



Homotopy Type Theory



Type Theory

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

- * Functional programming language

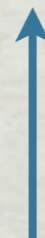
```
insertsort : list<int> → list<int>  
mergesort  : list<int> → list<int>
```

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

Propositions as Types

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

$\forall x. \text{mergesort}(x) = \text{insertsort}(x)$



type of proofs of program equality

Propositions as Types

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

`proof : $\forall x. \text{mergesort}(x) = \text{insertsort}(x)$`


type of proofs of program equality

Propositions as Types


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

`proof : $\forall x. \text{mergesort}(x) = \text{insertsort}(x)$`

`proof x = case x of`

`[] => reflexivity`

`(x :: xs) => ...`



***proof by case analysis represented
by a function defined by cases***

Type are sets?

Traditional view:

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{prog1} \rangle = \langle \text{prog2} \rangle$

set theory

$x \in S$

$x = y$

Type are sets?

Traditional view:

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{prog1} \rangle = \langle \text{prog2} \rangle$

set theory

$x \in S$

$x = y$

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

Type are sets?

Traditional view:

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog1} \rangle = \langle \text{prog2} \rangle$

set theory

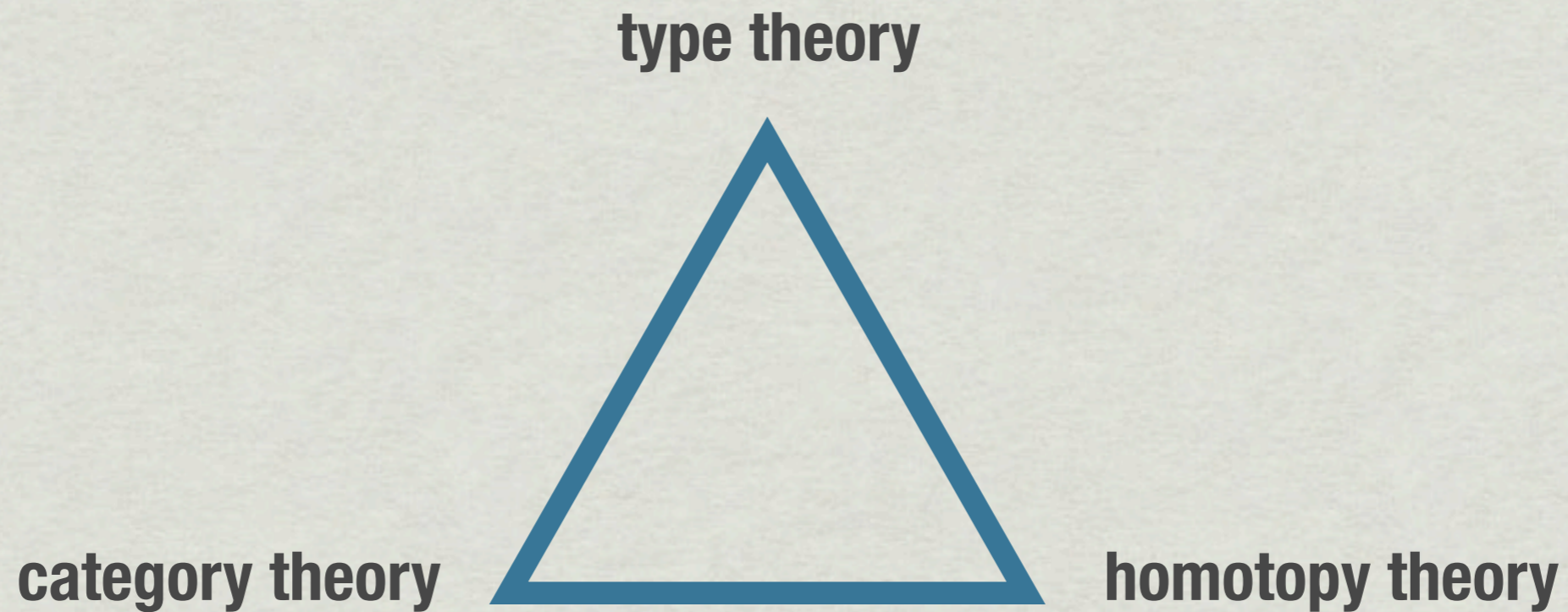
$x \in S$

$x = y$

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

In (intensional) type theory, an equation has
a non-trivial $\langle \text{proof} \rangle$

Homotopy Type Theory



Types are ∞ -groupoids

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

Types are ∞ -groupoids

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

set theory

$x \in S$

$x = y$

Types are ∞ -groupoids

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

$\langle 2\text{-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$

set theory

$x \in S$

$x = y$

Types are ∞ -groupoids

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

$\langle 2\text{-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$

$\langle 3\text{-proof} \rangle : \langle 2\text{-proof}_1 \rangle = \langle 2\text{-proof}_2 \rangle$

set theory

$x \in S$

$x = y$

Types are ∞ -groupoids

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

$\langle 2\text{-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$

$\langle 3\text{-proof} \rangle : \langle 2\text{-proof}_1 \rangle = \langle 2\text{-proof}_2 \rangle$

\vdots

set theory

$x \in S$

$x = y$

Types are ∞ -groupoids

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

$\langle 2\text{-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$

$\langle 3\text{-proof} \rangle : \langle 2\text{-proof}_1 \rangle = \langle 2\text{-proof}_2 \rangle$

\vdots

set theory

$x \in S$

$x = y$

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

Types are ∞ -groupoids

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

$\langle 2\text{-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$

$\langle 3\text{-proof} \rangle : \langle 2\text{-proof}_1 \rangle = \langle 2\text{-proof}_2 \rangle$

\vdots

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

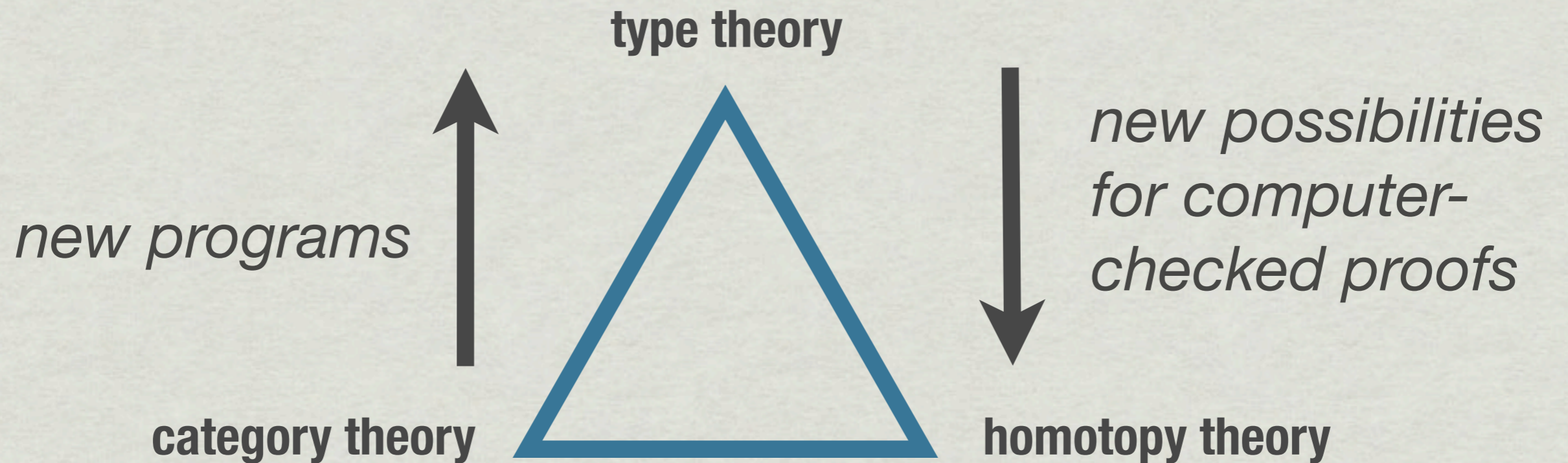
set theory

$x \in S$

$x = y$

∞ -groupoid:
each level has a
group structure,
and they interact

Homotopy Type Theory



I am developing
a computational theory of ∞ -groupoids
and applying it to
computer-checked math and software

Results

1. I have developed computer-checked proofs of theorems in homotopy theory [LICS'13]
2. I have discovered how to run programs in Homotopy Type Theory, for the special case of 2-dimensional type theory [POPL'12]
3. I have applied these new concepts to computer-checked software [thesis + MFPS'11]

Outline

1. Computer-checked homotopy theory
2. Computer-checked software

Outline

- 1. Computer-checked homotopy theory**
2. Computer-checked software

Homotopy Theory

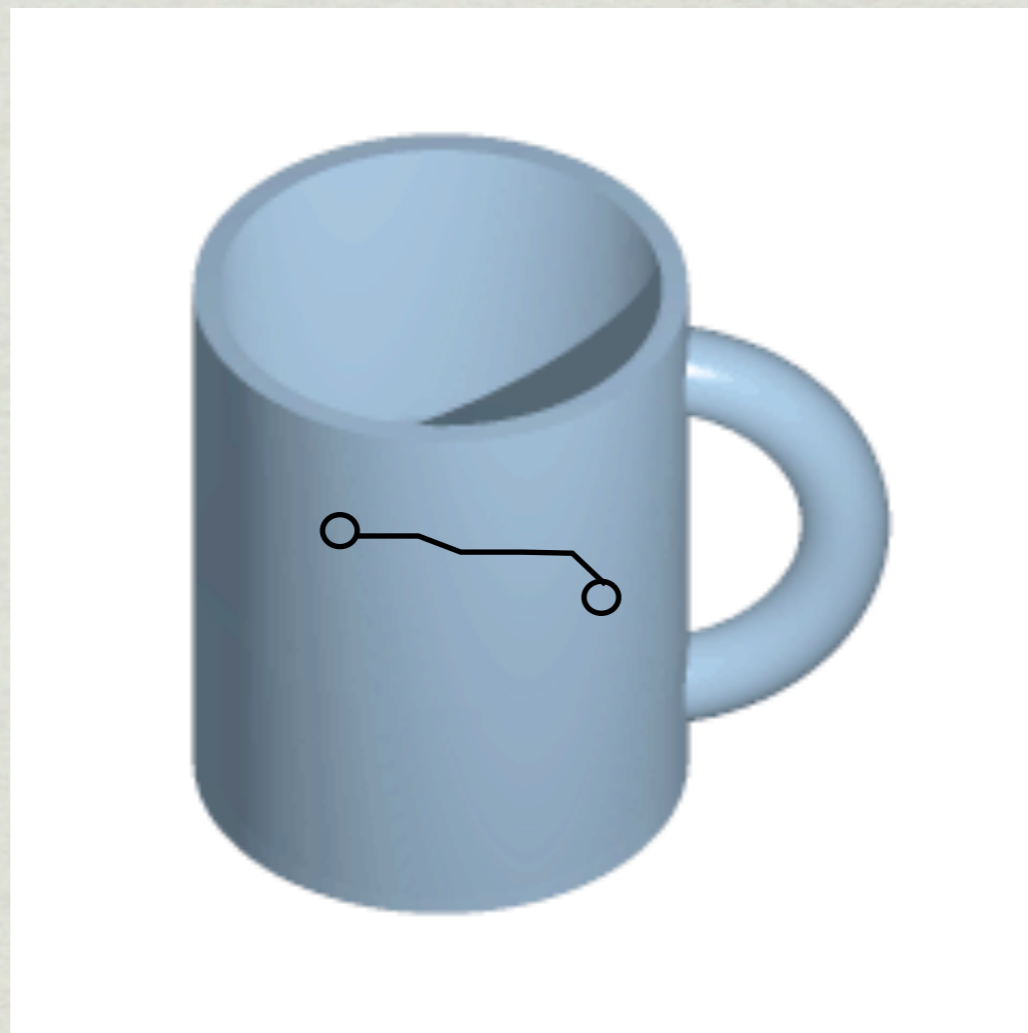
A branch of topology,
the study of spaces and continuous deformations



[image from wikipedia]

Homotopy Theory

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



[image from wikipedia]

Synthetic vs Analytic

Synthetic geometry (Euclid)

POSTULATES.

I.

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

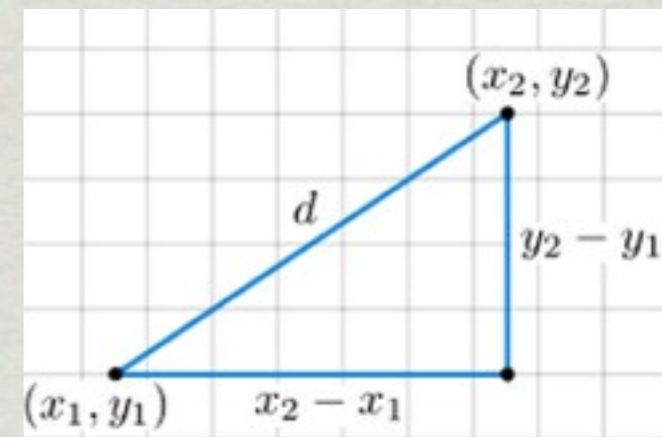
II.

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

III.

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

Analytic geometry (Descartes)



[image from wikipedia]

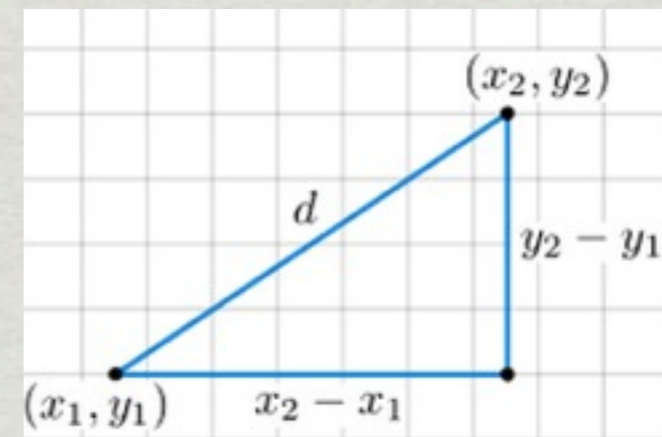
Synthetic vs Analytic

Synthetic geometry (Euclid)

POSTULATES.

- I.
LET it be granted that a straight line may be drawn from any one point to any other point.
- II.
That a terminated straight line may be produced to any length in a straight line.
- III.
And that a circle may be described from any centre, at any distance from that centre.

Analytic geometry (Descartes)



Classical homotopy theory is analytic:

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

[image from wikipedia]

Synthetic homotopy theory

homotopy theory

space

points

paths

homotopies

⋮

type theory

$\langle \text{type} \rangle$

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

$\langle 2\text{-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$

⋮

Synthetic homotopy theory

homotopy theory

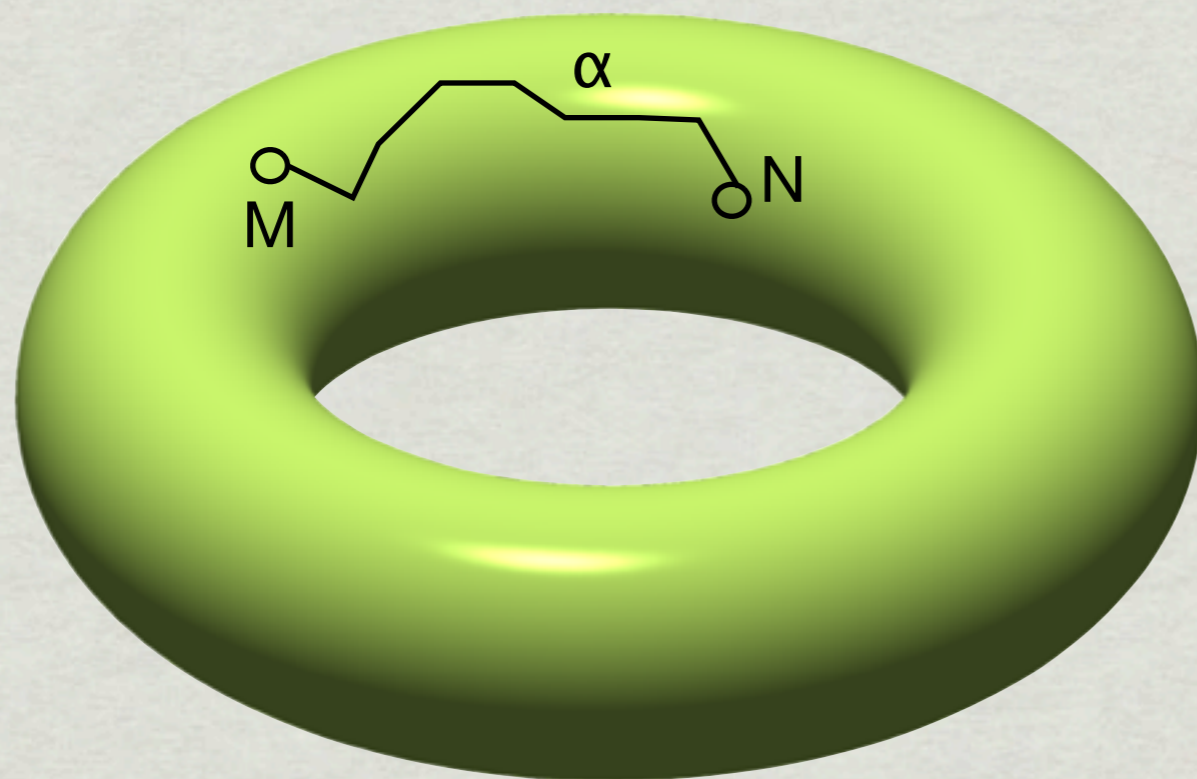
space
points
paths
homotopies
⋮

type theory

<type>
<program> : <type>
<proof> : <prog₁> = <prog₂>
<2-proof> : <proof₁> = <proof₂>
⋮

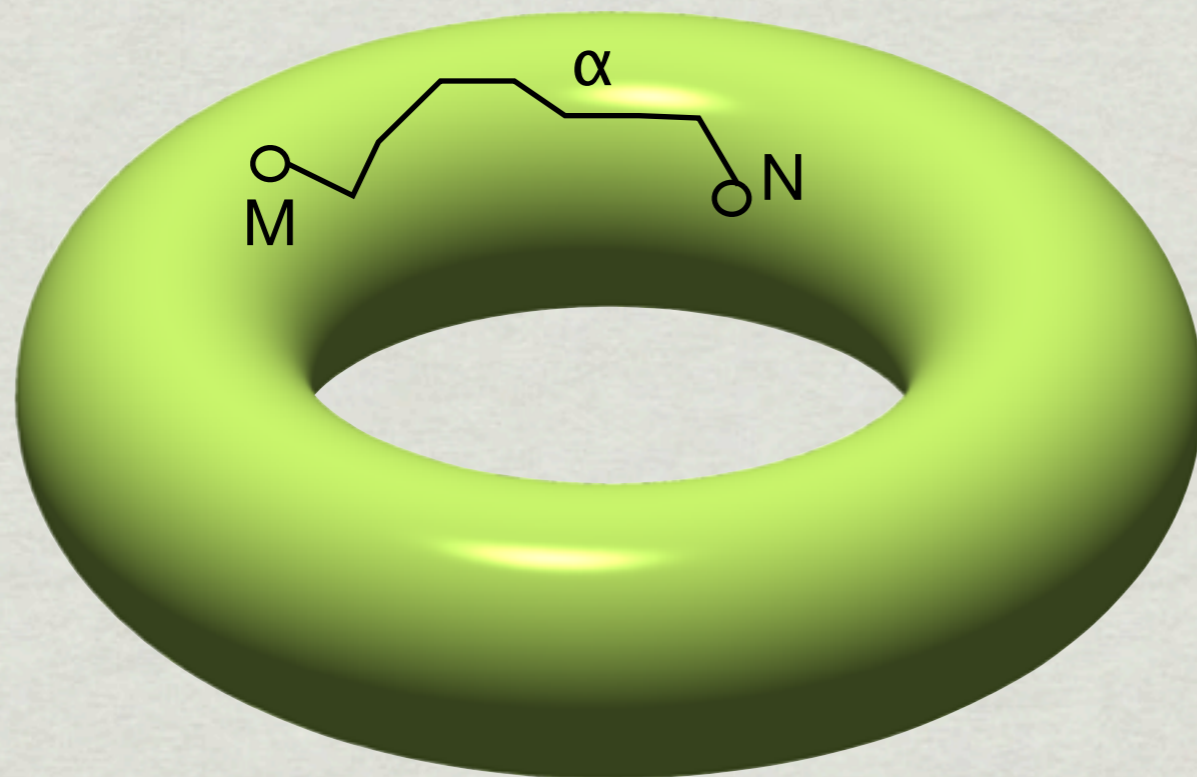
*A path is **not** a set of points; it is a primitive notion*

Spaces as types



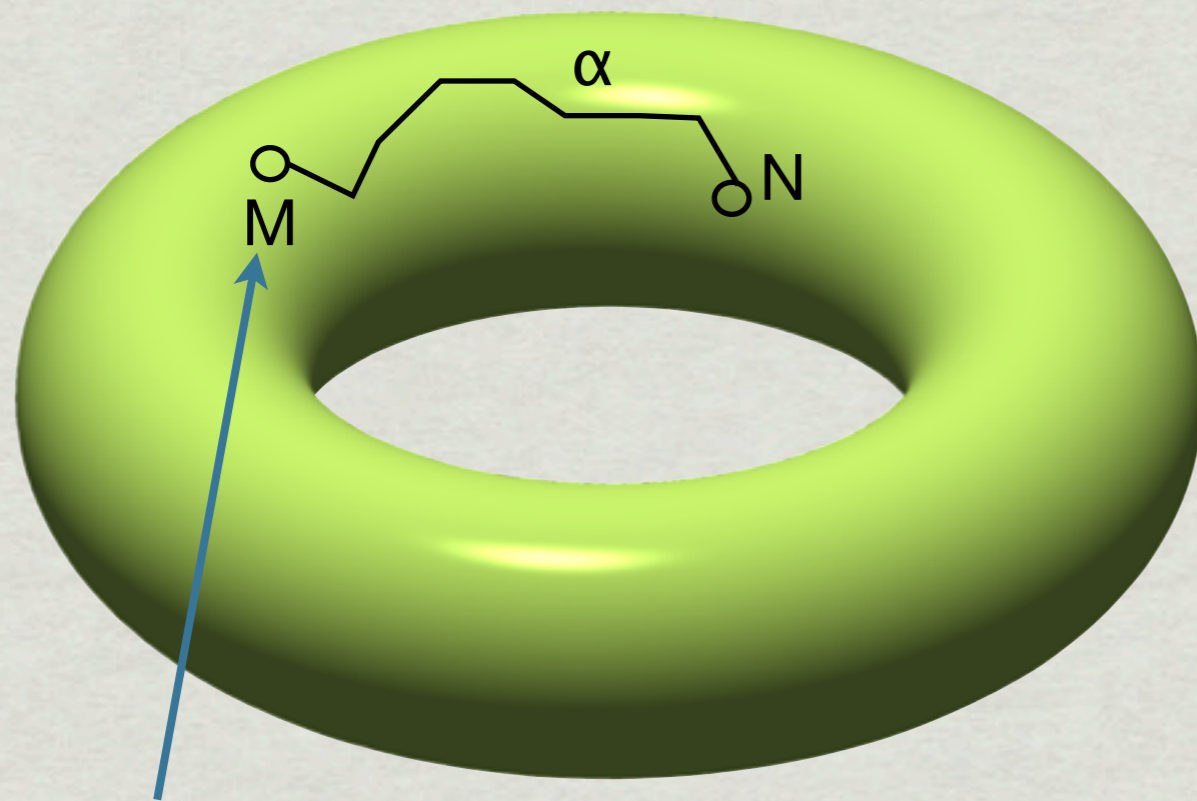
Spaces as types

a space is a type A



Spaces as types

a space is a type A

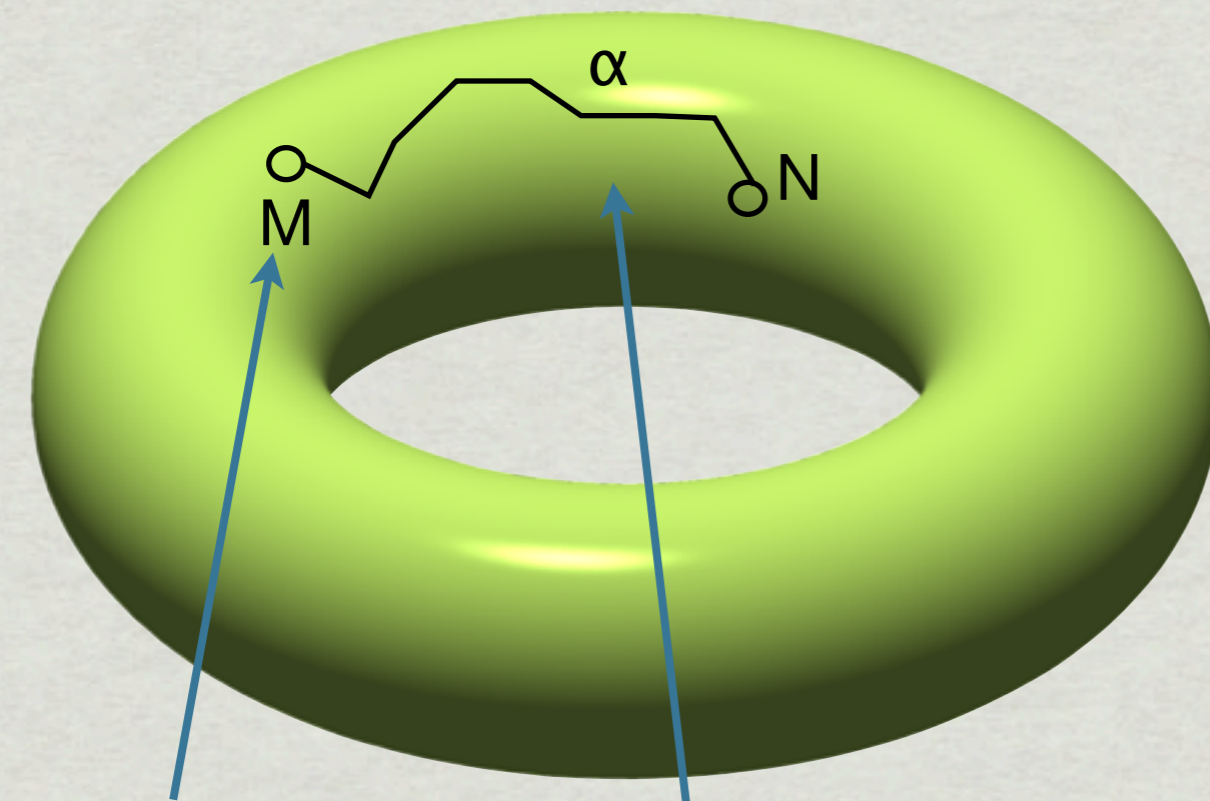


**points are
programs**

$M : A$

Spaces as types

a space is a type A



**points are
programs**

$M : A$

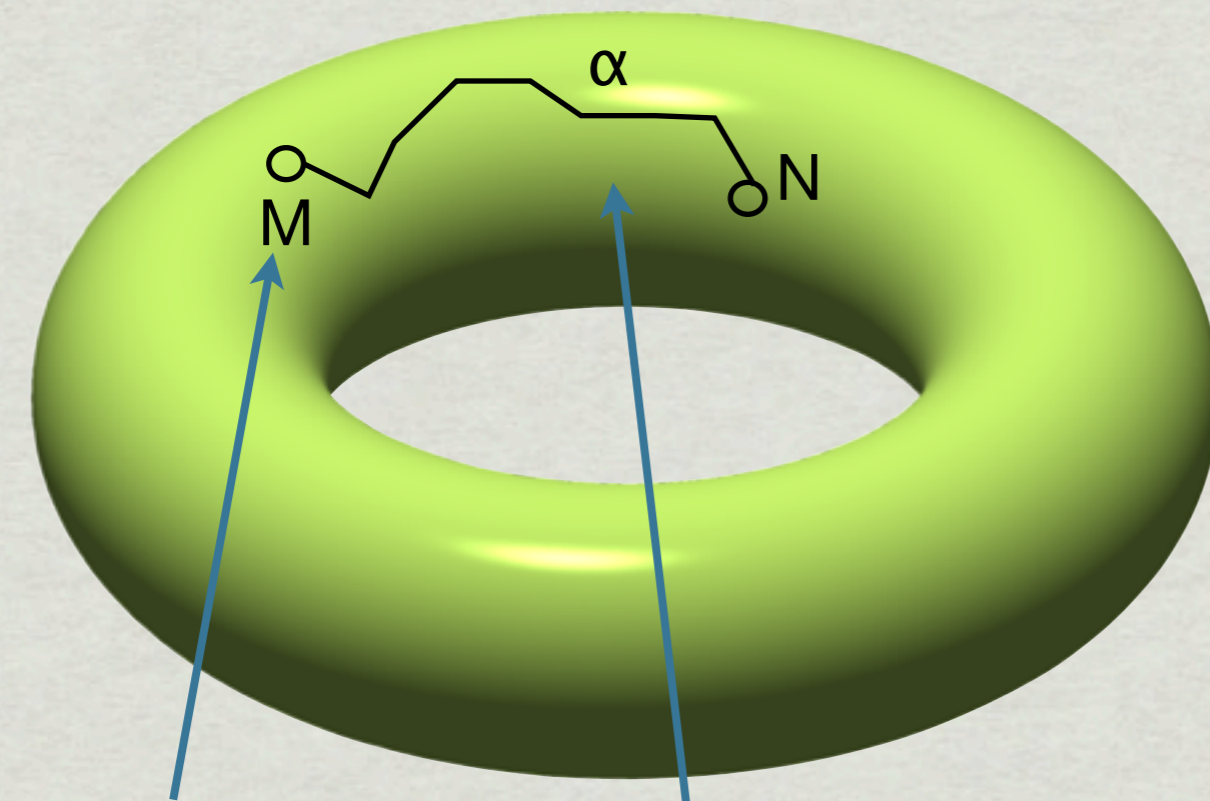
**paths are
*proofs of equality***

$\alpha : M =_A N$

Spaces as types

a space is a type A

path operations



points are
programs

$M : A$

paths are
proofs of equality

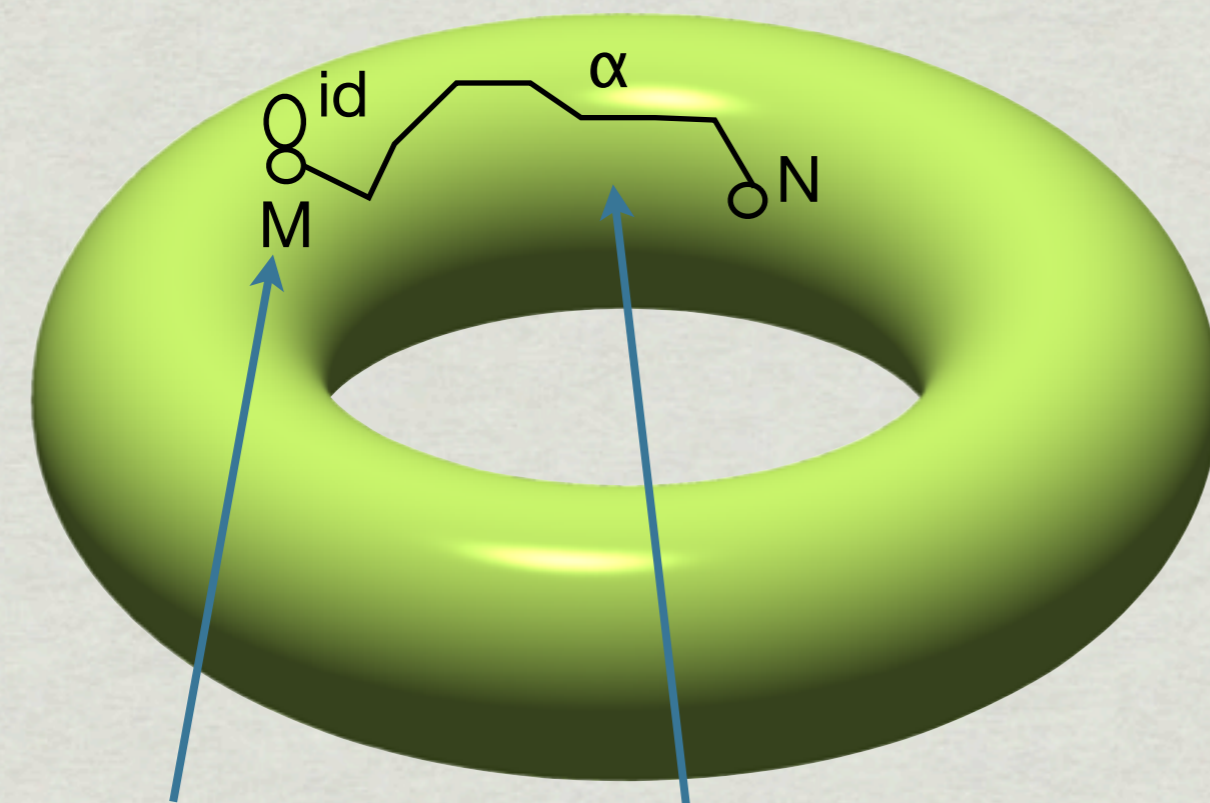
$\alpha : M =_A N$

Spaces as types

a space is a type A

path operations

$\text{id} : M = M \text{ (refl)}$



points are
programs

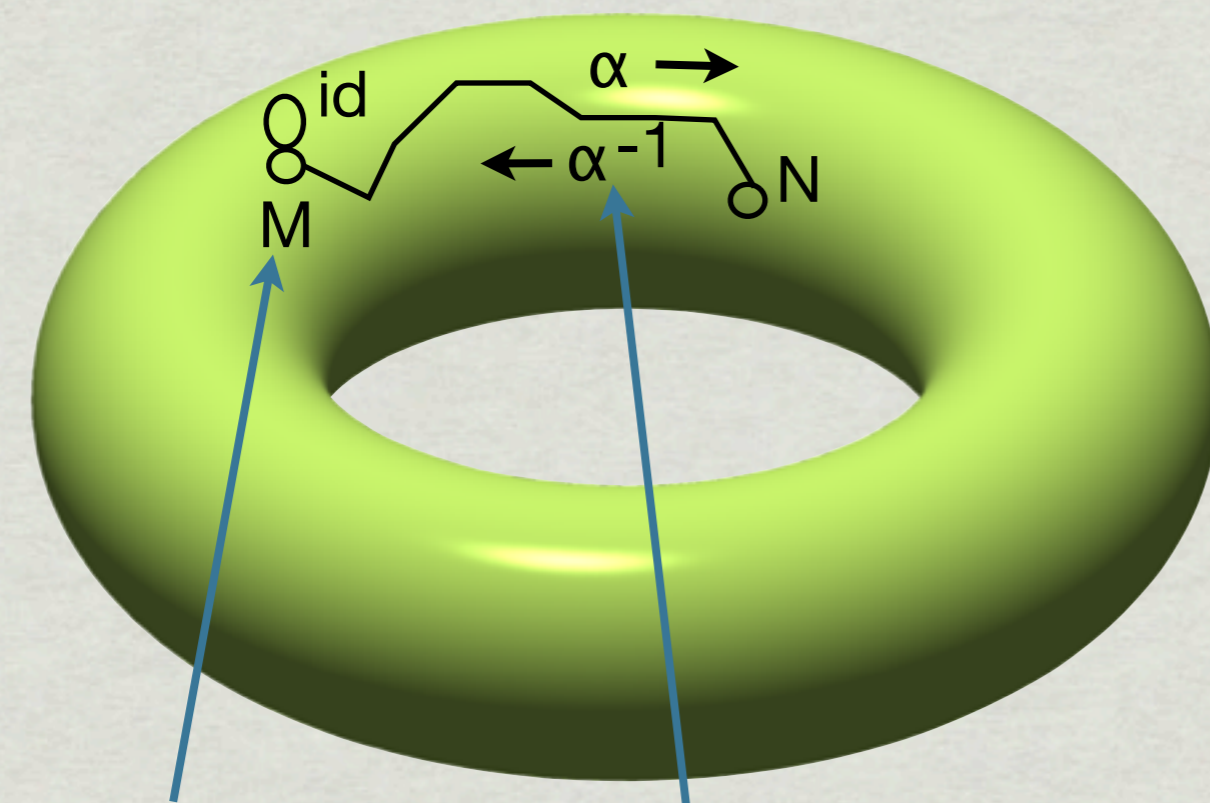
$M : A$

paths are
proofs of equality

$\alpha : M =_A N$

Spaces as types

a space is a type A



points are
programs

$M : A$

paths are
proofs of equality

$\alpha : M =_A N$

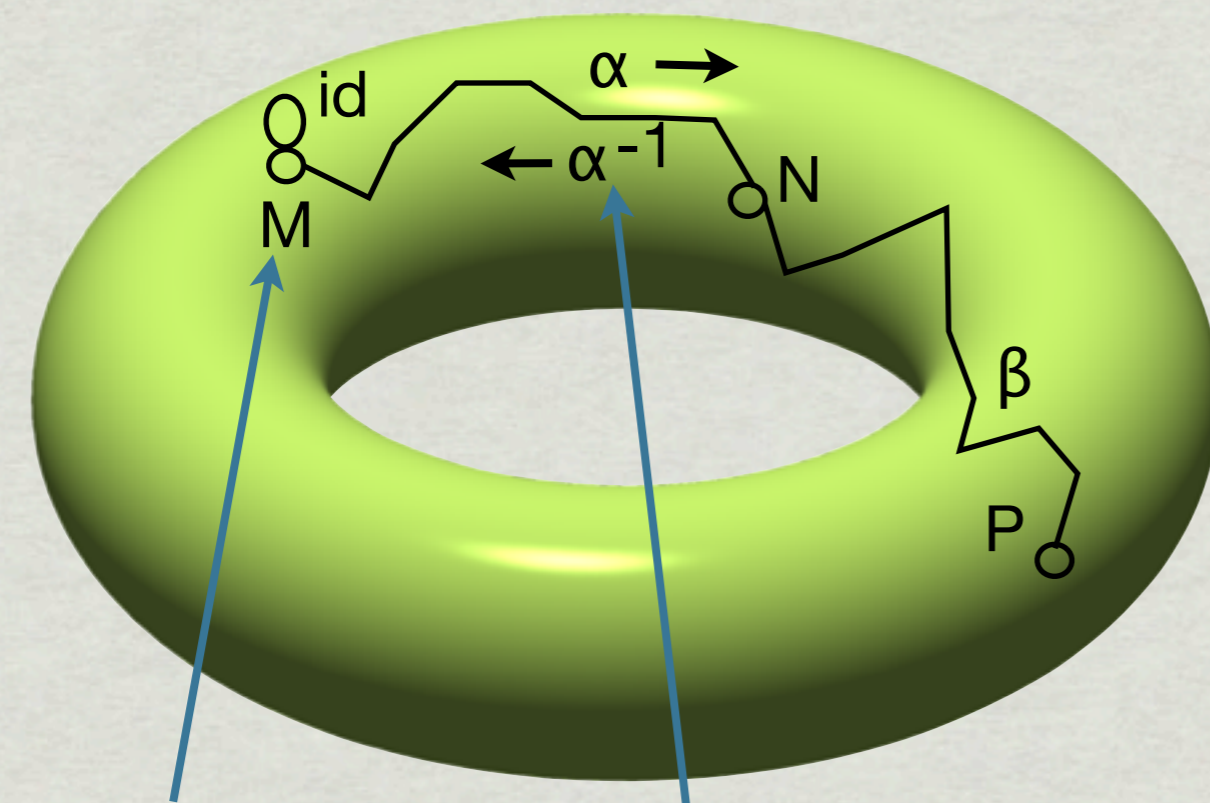
path operations

$\text{id} : M = M \text{ (refl)}$

$\alpha^{-1} : N = M \text{ (sym)}$

Spaces as types

a space is a type A



points are
programs

$M : A$

paths are
proofs of equality

$\alpha : M =_A N$

path operations

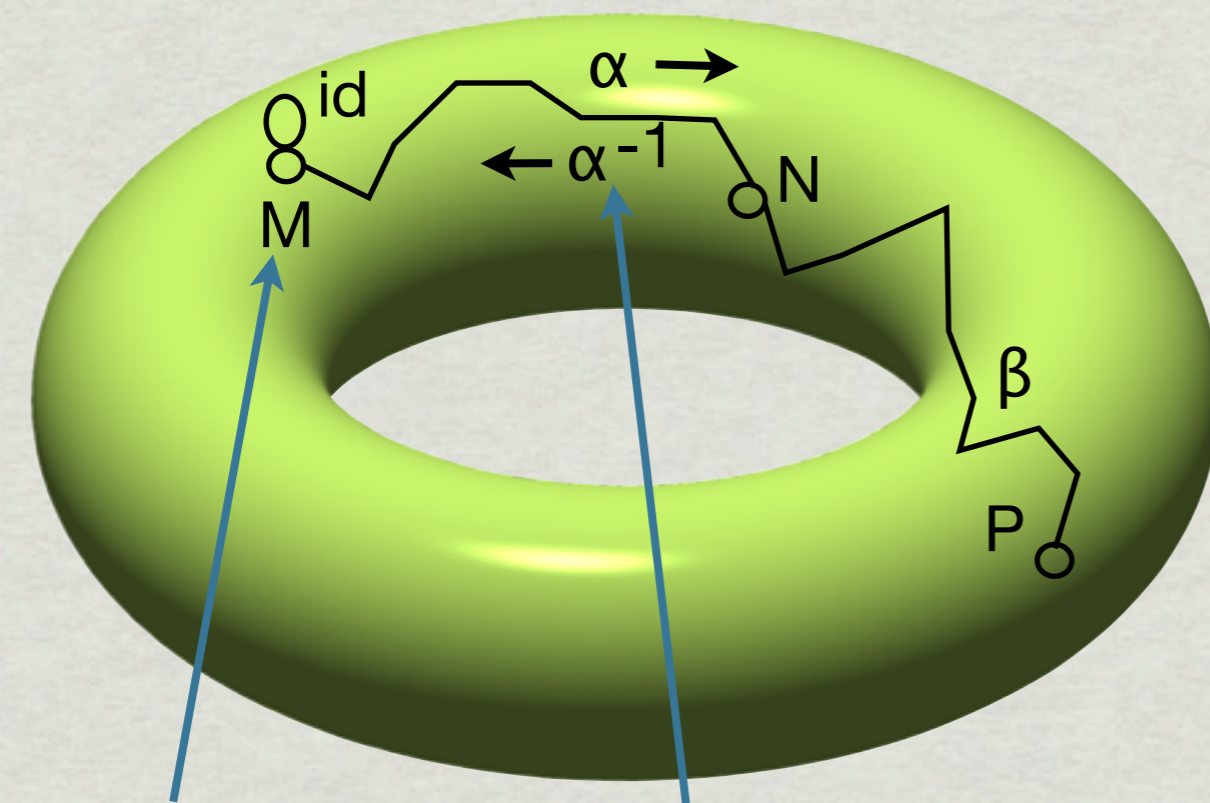
$\text{id} : M = M \text{ (refl)}$

$\alpha^{-1} : N = M \text{ (sym)}$

$\beta \circ \alpha : M = P \text{ (trans)}$

Spaces as types

a space is a type A



points are
programs
 $M : A$

paths are
proofs of equality
 $\alpha : M =_A N$

path operations

$\text{id} : M = M \text{ (refl)}$

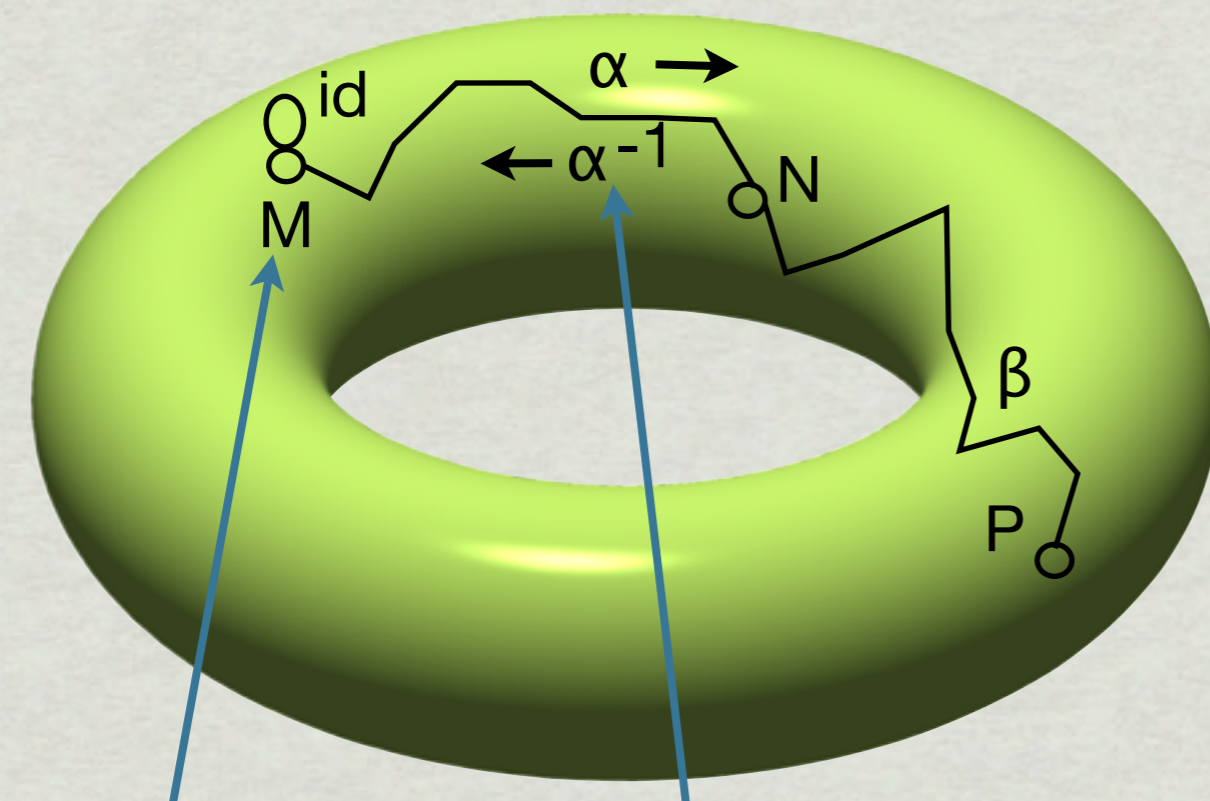
$\alpha^{-1} : N = M \text{ (sym)}$

$\beta \circ \alpha : M = P \text{ (trans)}$

Fundamental group:
group of loops

Spaces as types

a space is a type A



points are
programs

$M : A$

paths are
proofs of equality

$\alpha : M =_A N$

path operations

$\text{id} : M = M \text{ (refl)}$

$\alpha^{-1} : N = M \text{ (sym)}$

$\beta \circ \alpha : M = P \text{ (trans)}$

Fundamental group:

group of loops
modulo homotopy

Homotopy

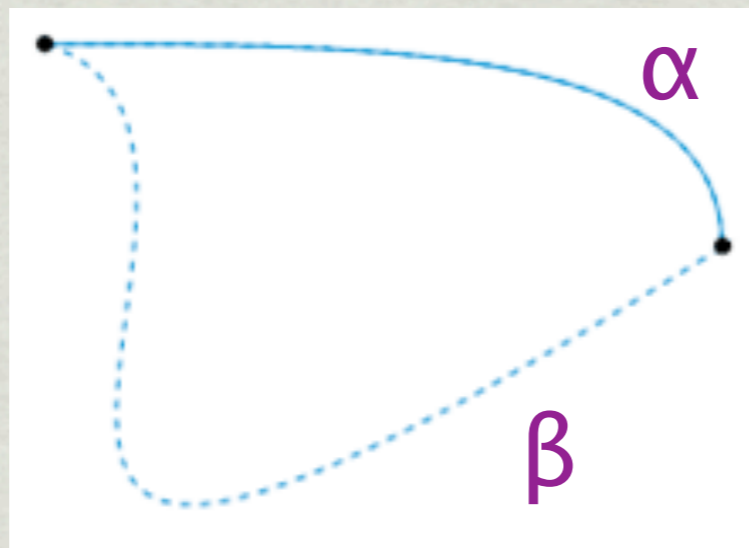
Deformation of one path into another

α

β

Homotopy

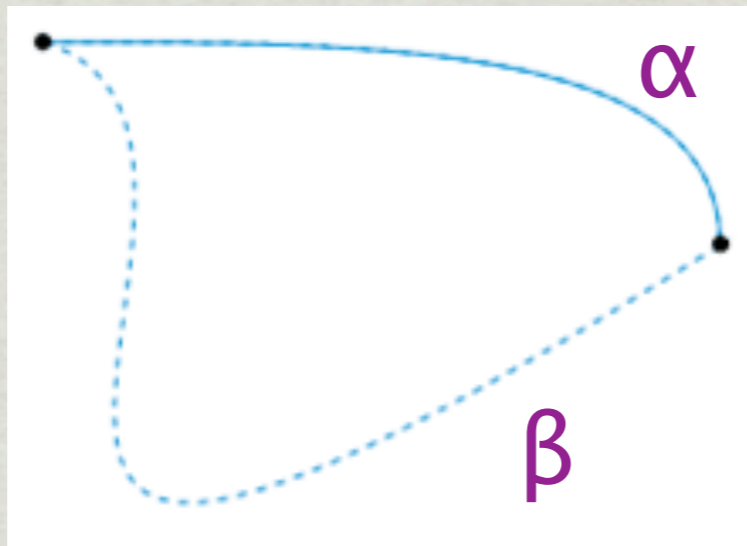
Deformation of one path into another



[image from wikipedia]

Homotopy

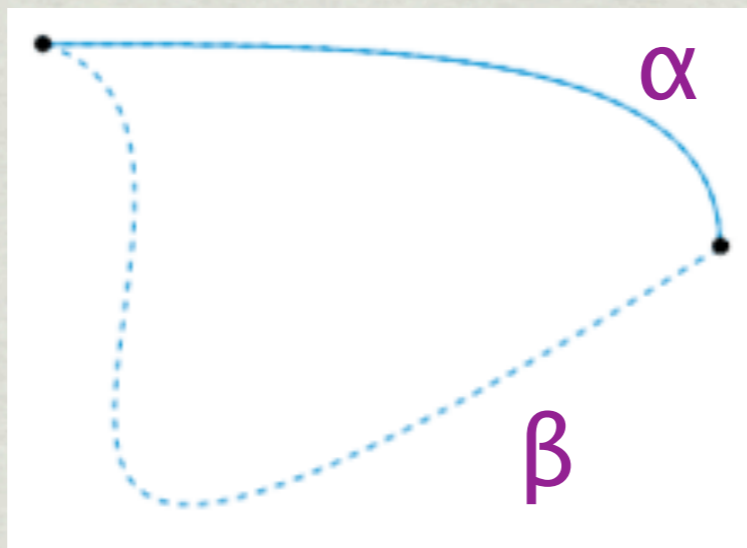
Deformation of one path into another



[image from wikipedia]

Homotopy

Deformation of one path into another

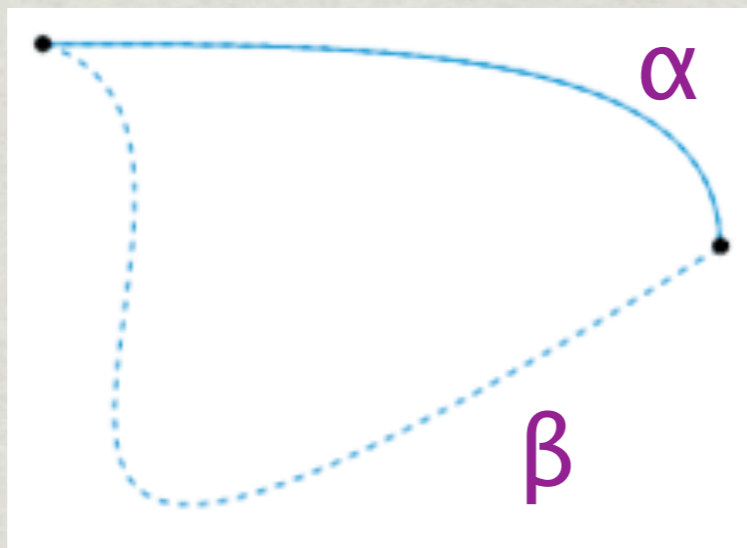


= 2-dimensional *path between paths*

[image from wikipedia]

Homotopy

Deformation of one path into another



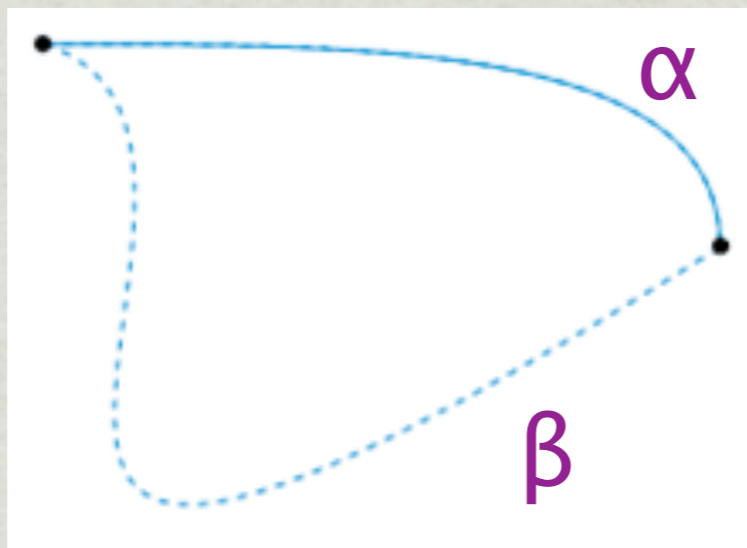
$\langle 2\text{-proof} \rangle : \alpha = \beta$

= 2-dimensional *path between paths*

[image from wikipedia]

Homotopy

Deformation of one path into another



<2-proof> : $\alpha = \beta$

= 2-dimensional *path between paths*

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

[image from wikipedia]

We can do homotopy theory
by writing functional programs

Functions on sets

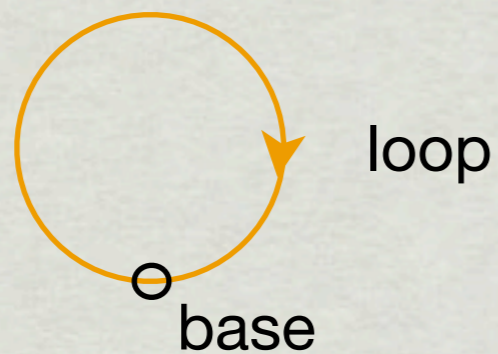
Function on a set gives the image of each element:

$$\begin{aligned}\text{not} &: \text{Bool} \rightarrow \text{Bool} \\ \text{not}(\text{true}) &= \text{false} \\ \text{not}(\text{false}) &= \text{true}\end{aligned}$$

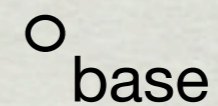
Functions on spaces

Function on a space gives the image of each point

Circle



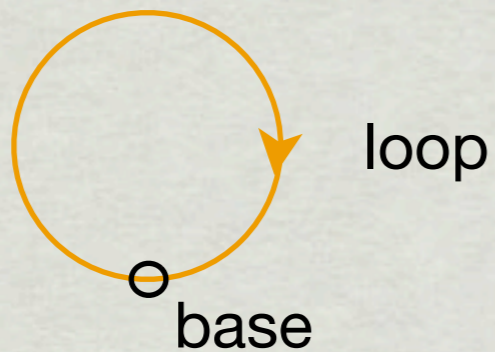
Circle



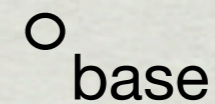
Functions on spaces

Function on a space gives the image of each point

Circle



Circle

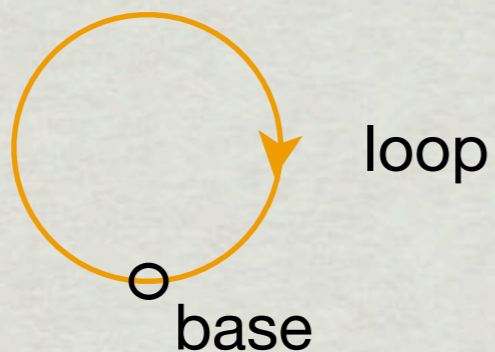


and each path!

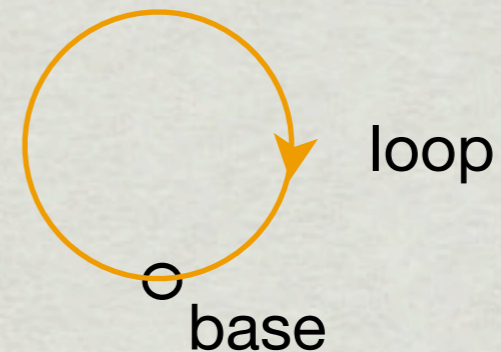
Functions on spaces

Function on a space gives the image of each point

Circle



Circle

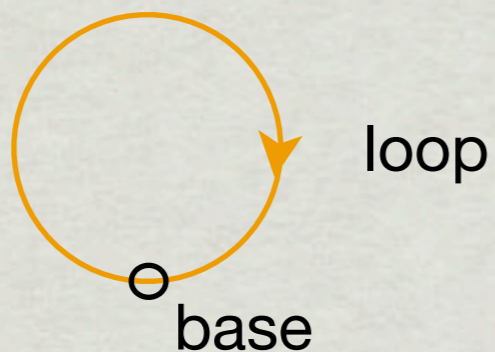


and each path!

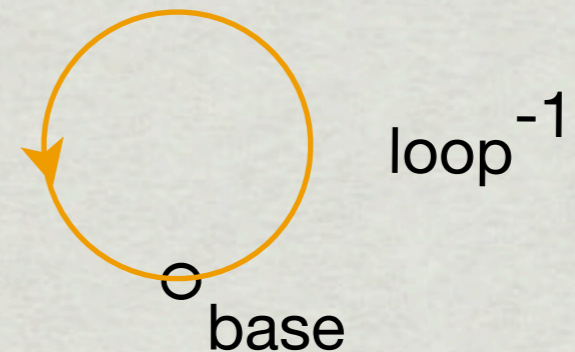
Functions on spaces

Function on a space gives the image of each point

Circle



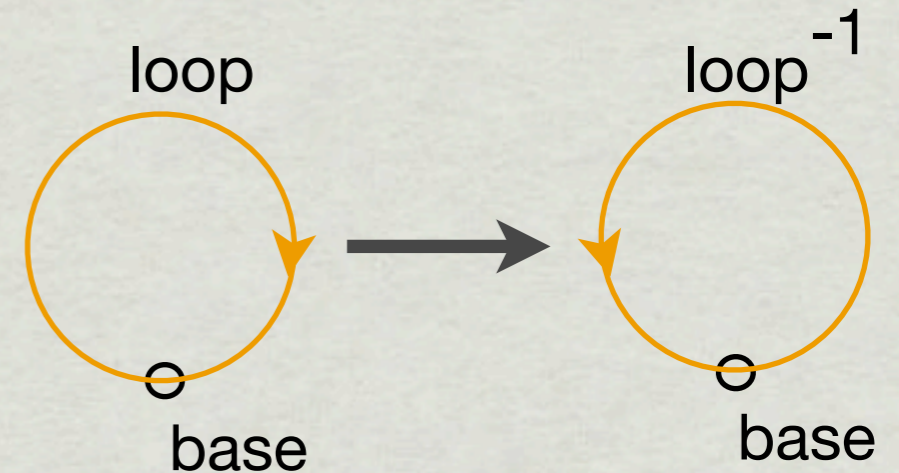
Circle



and each path!

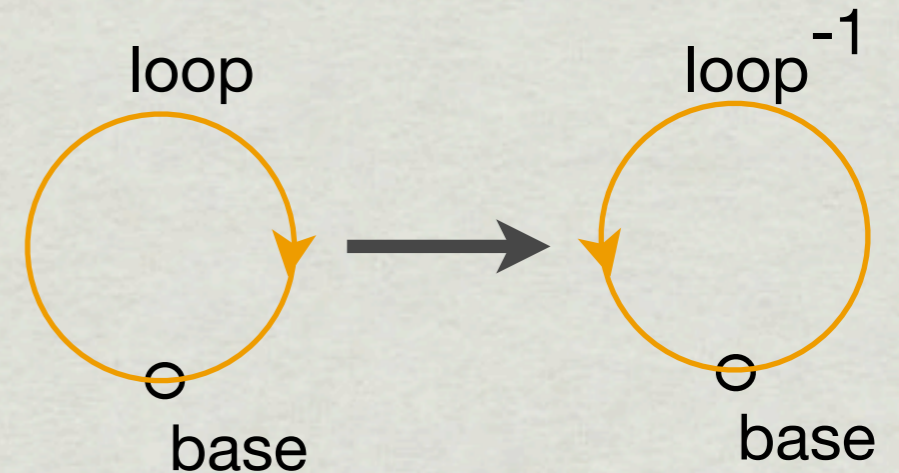
Circle Recursion

`reverse : Circle → Circle`
`reverse(base) = base`
`reverse(loop) = loop-1`



Circle Recursion

$\text{reverse} : \text{Circle} \rightarrow \text{Circle}$
 $\text{reverse}(\text{base}) = \text{base}$
 $\text{reverse}(\text{loop}) = \text{loop}^{-1}$



This specifies the image for **all paths** because

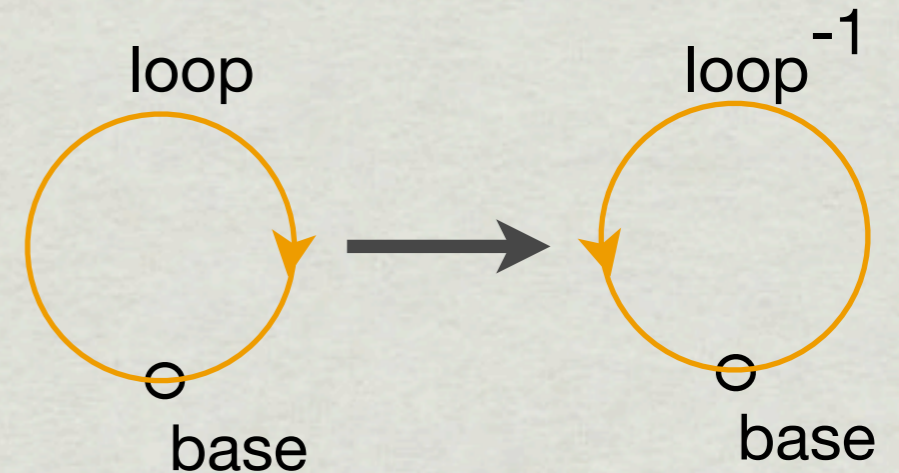
1. circle is **inductively generated** by `Loop`: all paths are built from `Loop` by identity, inverse, composition
2. all functions are homomorphisms

Homomorphism

$\text{reverse} : \text{Circle} \rightarrow \text{Circle}$
 $\text{reverse}(\text{base}) = \text{base}$
 $\text{reverse}(\text{loop}) = \text{loop}^{-1}$

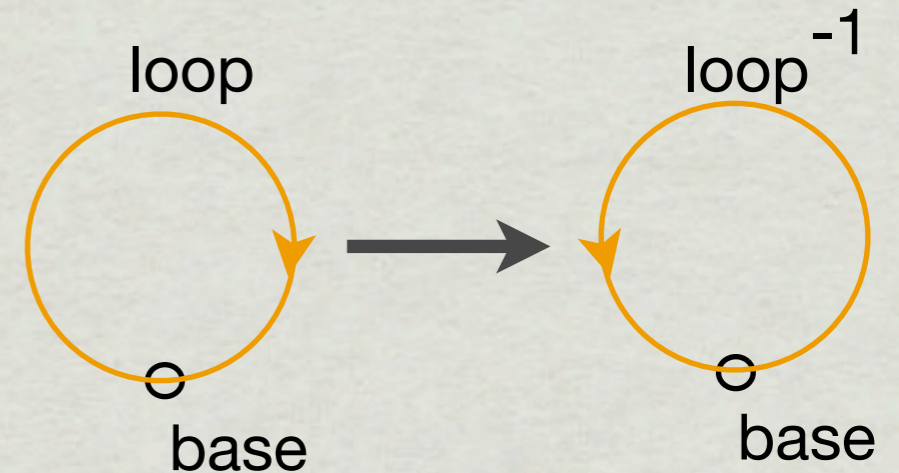
Computation steps:

$\text{reverse}(\text{loop } \circ \text{ loop})$



Homomorphism

$\text{reverse} : \text{Circle} \rightarrow \text{Circle}$
 $\text{reverse}(\text{base}) = \text{base}$
 $\text{reverse}(\text{loop}) = \text{loop}^{-1}$

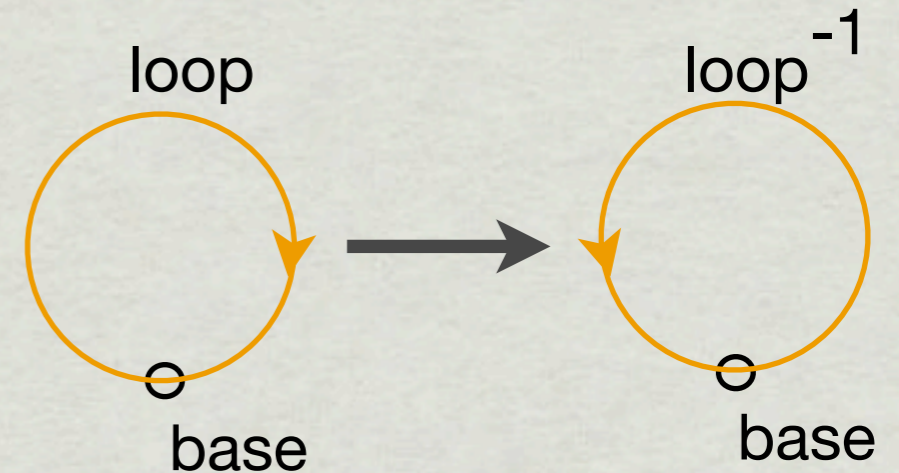


Computation steps:

$\text{reverse}(\text{loop} \circ \text{loop})$
 $= (\text{reverse loop}) \circ (\text{reverse loop})$ **homomorphism**

Homomorphism

$\text{reverse} : \text{Circle} \rightarrow \text{Circle}$
 $\text{reverse}(\text{base}) = \text{base}$
 $\text{reverse}(\text{loop}) = \text{loop}^{-1}$

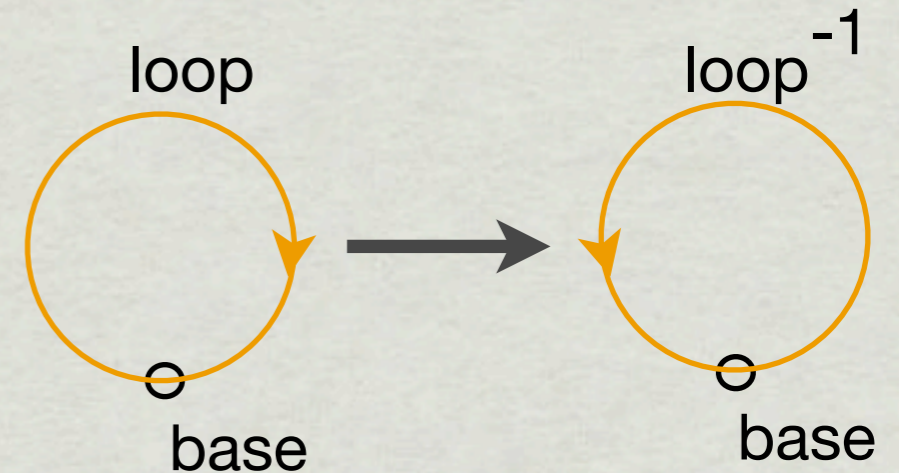


Computation steps:

$\text{reverse}(\text{loop} \circ \text{loop})$
 $= (\text{reverse loop}) \circ (\text{reverse loop})$ **homomorphism**
 $= \text{loop}^{-1} \circ \text{loop}^{-1}$ **definition**

Homomorphism

$\text{reverse} : \text{Circle} \rightarrow \text{Circle}$
 $\text{reverse}(\text{base}) = \text{base}$
 $\text{reverse}(\text{loop}) = \text{loop}^{-1}$

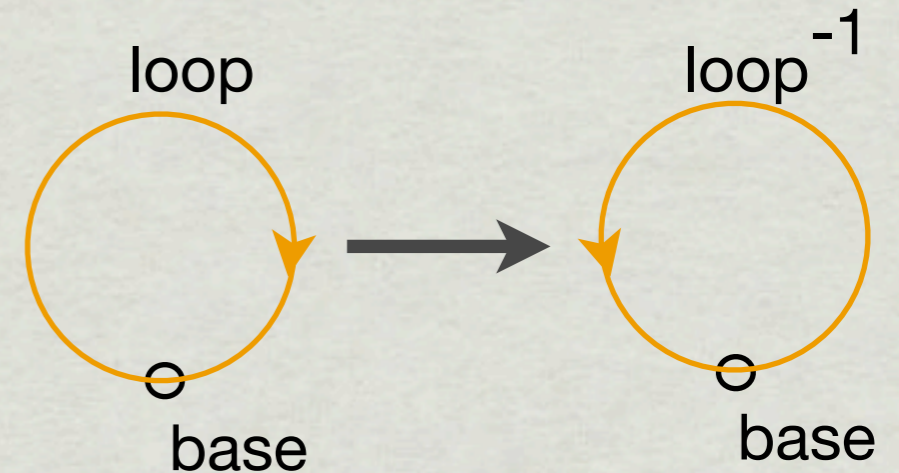


Computation steps:

$\text{reverse}(\text{loop} \circ \text{loop})$
 $= (\text{reverse loop}) \circ (\text{reverse loop})$ **homomorphism**
 $= \text{loop}^{-1} \circ \text{loop}^{-1}$ **definition**
 $= (\text{loop} \circ \text{loop})^{-1}$ **group laws**

Homomorphism

$\text{reverse} : \text{Circle} \rightarrow \text{Circle}$
 $\text{reverse}(\text{base}) = \text{base}$
 $\text{reverse}(\text{loop}) = \text{loop}^{-1}$

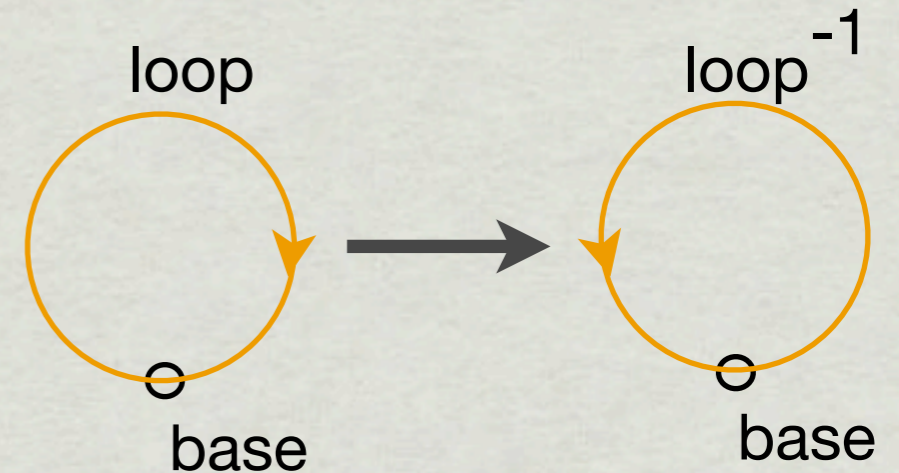


Computation steps:

$\text{reverse}(\text{loop} \circ \text{loop})$
 $= (\text{reverse loop}) \circ (\text{reverse loop})$ **homomorphism**
 $= \text{loop}^{-1} \circ \text{loop}^{-1}$ **definition**
 $= (\text{loop} \circ \text{loop})^{-1}$ **group laws**

Circle induction

$\text{reverse} : \text{Circle} \rightarrow \text{Circle}$
 $\text{reverse}(\text{base}) = \text{base}$
 $\text{reverse}(\text{loop}) = \text{loop}^{-1}$



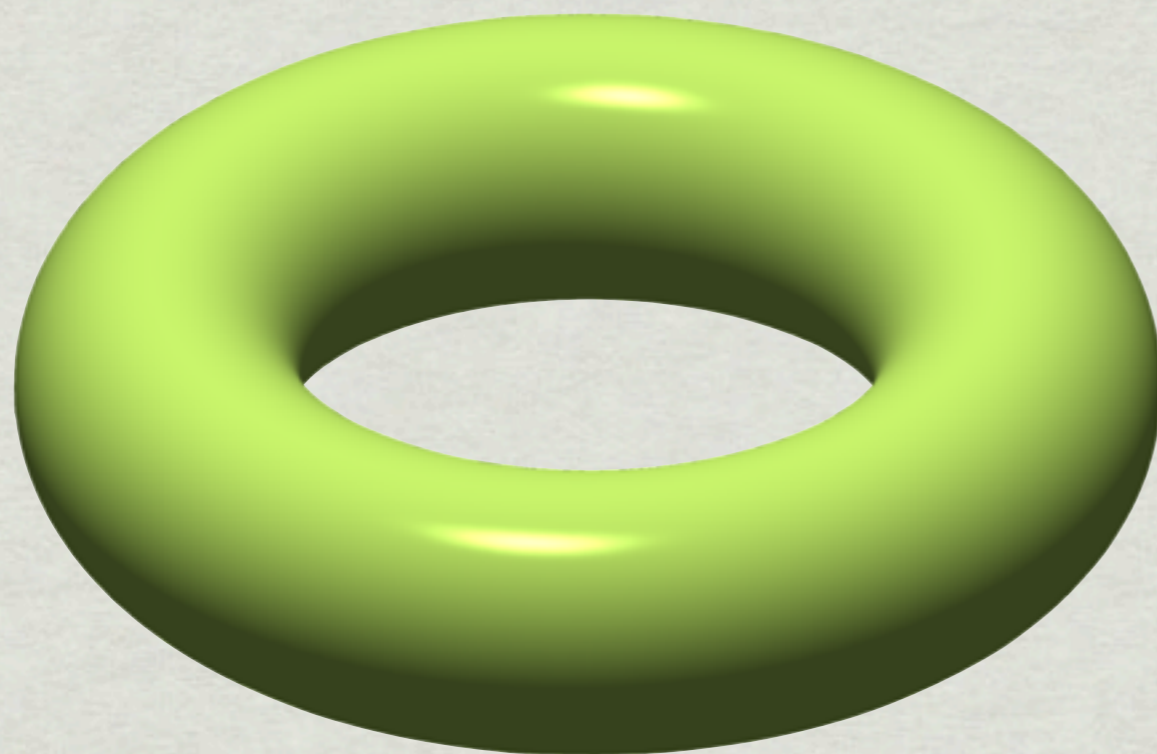
Theorem: $\forall p. \text{reverse}(p) = p^{-1}$

Proof: uses *circle induction*:

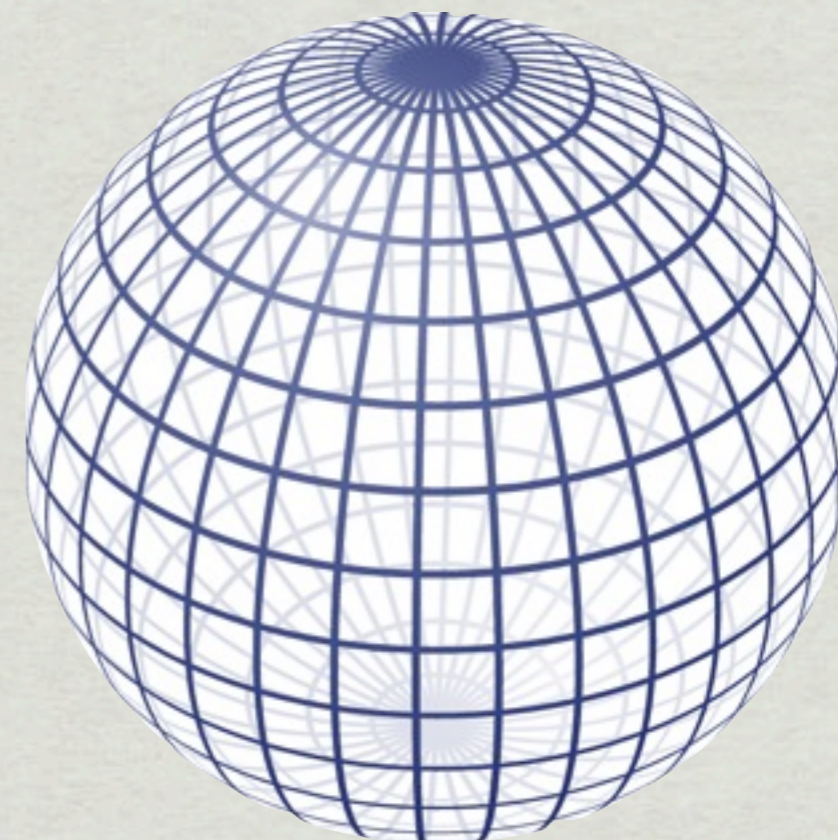
To prove a predicate P for all points on the circle,
suffices to prove $P(\text{base})$,
continuously in the loop

We can do interesting
homotopy theory synthetically

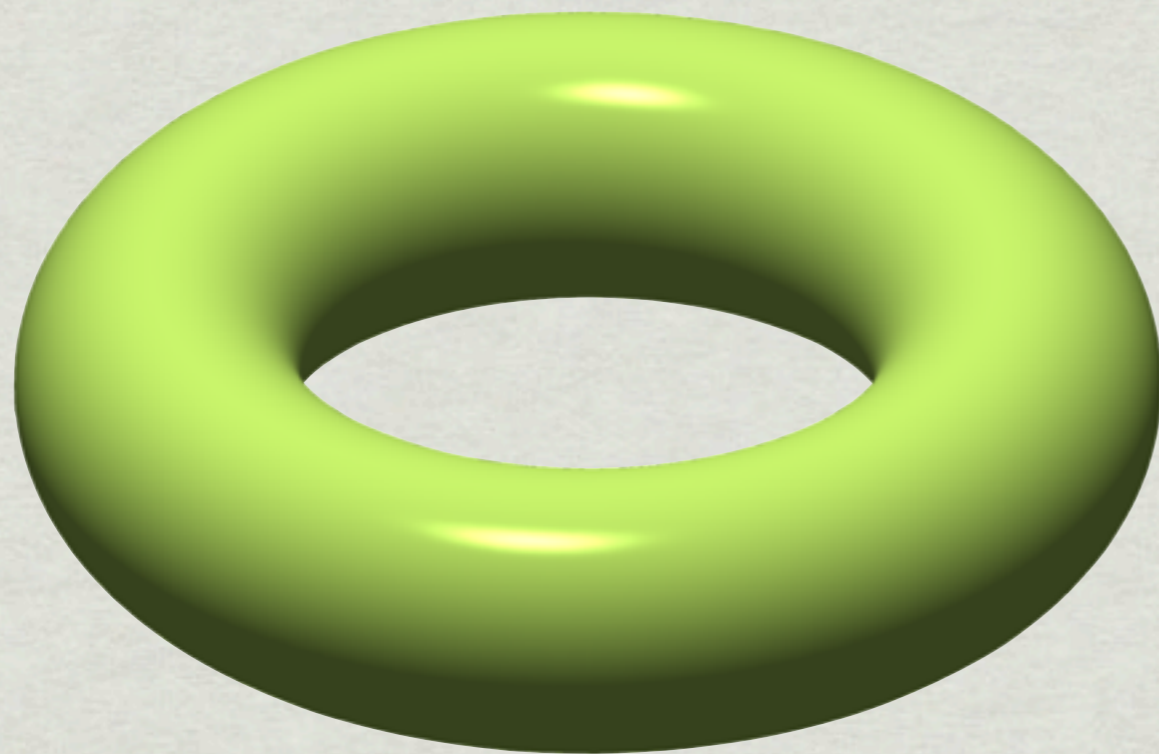
Telling spaces apart



\neq



Telling spaces apart



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

\neq



fundamental group
is trivial

Homotopy Groups

Homotopy groups of a space X :

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

Homotopy groups

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]

Computer-checked proofs

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]

Computer-checked proofs

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}													
S^3	0	0	\mathbb{Z}	\mathbb{Z}_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]

Computer-checked proofs

1. $\pi_n(S^n) = \mathbb{Z}$ (w/ G. Brunerie)
2. $\pi_k(S^n)$ trivial for $k < n$
3. Freudenthal suspension theorem
(w/ P. Lumsdaine; Blakers-Massey w.i.p)
4. Eilenberg-Mac Lane spaces $K(G, n)$

	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}													
S^3	0	0	\mathbb{Z}	\mathbb{Z}_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	

Computer-checked proofs

1. $\pi_n(S^n) = \mathbb{Z}$ (w/ G. Brunerie)
2. $\pi_k(S^n)$ trivial for $k < n$
3. Freudenthal suspension theorem
(w/ P. Lumsdaine; Blakers-Massey w.i.p)
4. Eilenberg-Mac Lane spaces $K(G, n)$

Computer-checked proofs

1. $\pi_n(S^n) = \mathbb{Z}$ (w/ G. Brunerie)

2. $\pi_k(S^n)$ trivial for $k < n$

3. Freudenthal suspension theorem
(w/ P. Lumsdaine; Blakers-Massey w.i.p)

4. Eilenberg-Mac Lane spaces $K(G, n)$

✱ 11,000 lines of Agda code (most since January)

Computer-checked proofs

1. $\pi_n(S^n) = \mathbb{Z}$ (w/ G. Brunerie)
2. $\pi_k(S^n)$ trivial for $k < n$
3. Freudenthal suspension theorem
(w/ P. Lumsdaine; Blakers-Massey w.i.p)
4. Eilenberg-Mac Lane spaces $K(G, n)$

- * 11,000 lines of Agda code (most since January)
- * Proofs are **programs**: you can run them

Computer-checked proofs

1. $\pi_n(S^n) = \mathbb{Z}$ (w/ G. Brunerie)

2. $\pi_k(S^n)$ trivial for $k < n$

3. Freudenthal suspension theorem
(w/ P. Lumsdaine; Blakers-Massey w.i.p)

4. Eilenberg-Mac Lane spaces $K(G, n)$

- * 11,000 lines of Agda code (most since January)
- * Proofs are **programs**: you can run them
- * Computer-checked proofs **shorter** than “informalized”

Computer-checked proofs

1. $\pi_n(S^n) = \mathbb{Z}$ (w/ G. Brunerie)
2. $\pi_k(S^n)$ trivial for $k < n$
3. Freudenthal suspension theorem
(w/ P. Lumsdaine; Blakers-Massey w.i.p)
4. Eilenberg-Mac Lane spaces $K(G, n)$

- * 11,000 lines of Agda code (most since January)
- * Proofs are **programs**: you can run them
- * Computer-checked proofs **shorter** than “informalized”
- * Proofs are **new**: I discovered a type-theoretic method that is used in all of these proofs

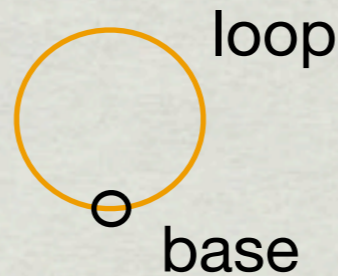
Computer-checked proofs

1. $\pi_n(S^n) = \mathbb{Z}$ (w/ G. Brunerie)
2. $\pi_k(S^n)$ trivial for $k < n$
3. Freudenthal suspension theorem
(w/ P. Lumsdaine; Blakers-Massey w.i.p)
4. Eilenberg-Mac Lane spaces $K(G, n)$

	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	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S^2	0	Z	0	0	0	0	0	0	0	0	0	0	0	0	0
S^3	0	0	Z	\mathbb{Z}_2	0	0	0	0	0	0	0	0	0	0	0
S^4	0	0	0	Z	\mathbb{Z}_2	\mathbb{Z}_2	0	0	0	0	0	0	0	0	0
S^5	0	0	0	0	Z	\mathbb{Z}_2	\mathbb{Z}_2	\mathbb{Z}_{24}	0	0	0	0	0	0	0
S^6	0	0	0	0	0	Z	\mathbb{Z}_2	\mathbb{Z}_2	\mathbb{Z}_{24}	0	0	0	0	0	0
S^7	0	0	0	0	0	0	Z	\mathbb{Z}_2	\mathbb{Z}_2	\mathbb{Z}_{24}	0	0	0	0	0
S^8	0	0	0	0	0	0	0	Z	\mathbb{Z}_2	\mathbb{Z}_2	\mathbb{Z}_{24}	0	0	0	\mathbb{Z}_2

Fundamental group of circle

[LICS'13]



Two functions:

$$1. \text{winding} : (\text{base} = \text{base}) \rightarrow \mathbb{Z}$$

$$2. \text{loop}^n : \mathbb{Z} \rightarrow (\text{base} = \text{base})$$

Three proofs:

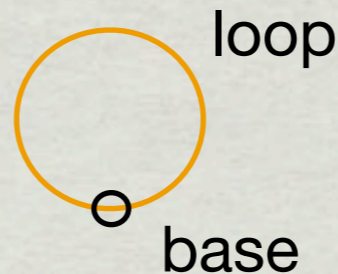
$$1. \forall n : \mathbb{Z}. \text{winding}(\text{loop}^n) = n$$

$$2. \forall p. \text{loop}^{\text{winding}(p)} = p$$

$$3. \forall n, m. \text{loop}^{n+m} = \text{loop}^n \circ \text{loop}^m$$

Fundamental group of circle

[LICS'13]



Two functions:

1. $\text{winding} : (\text{base} = \text{base}) \rightarrow \mathbb{Z}$ **uses circle recursion**

2. $\text{loop}^n : \mathbb{Z} \rightarrow (\text{base} = \text{base})$

Three proofs:

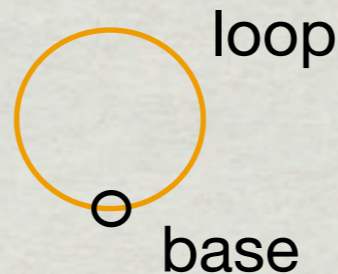
1. $\forall n : \mathbb{Z}. \text{winding}(\text{loop}^n) = n$

2. $\forall p. \text{loop}^{\text{winding}(p)} = p$

3. $\forall n, m. \text{loop}^{n+m} = \text{loop}^n \circ \text{loop}^m$

Fundamental group of circle

[LICS'13]



Two functions:

1. $\text{winding} : (\text{base} = \text{base}) \rightarrow \mathbb{Z}$ **uses circle recursion**

2. $\text{loop}^n : \mathbb{Z} \rightarrow (\text{base} = \text{base})$

Three proofs:

1. $\forall n : \mathbb{Z}. \text{winding}(\text{loop}^n) = n$

2. $\forall p. \text{loop}^{\text{winding}(p)} = p$

3. $\forall n, m. \text{loop}^{n+m} = \text{loop}^n \circ \text{loop}^m$

**induction principles for
circle, paths, int; and
calculations using my
computational
interpretation**

Fundamental group of the circle

Informal

Computer-checked

7.2.1.1 Encode/decode proof

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

$$P(x : S^1) := \{\text{base} \Rightarrow x\}$$

with one end-point free.

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

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

$$\begin{aligned}\text{code}(\text{base}) &:= \mathbb{Z} \\ \text{code}(\text{loop}) &:= \text{us}(\text{succ})\end{aligned}$$

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

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

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

Lemma 7.2.2. $\text{transport}^{\text{code}}(\text{loop}, x) = x + 1$ and $\text{transport}^{\text{code}}(\text{loop}^{-1}, x) = x - 1$

Proof. For the first, we calculate as follows:

$$\begin{aligned}\text{transport}^{\text{code}}(\text{loop}, x) &= \text{transport}^{\text{code}}(\text{code}(\text{loop}), x) && \text{associativity} \\ &= \text{transport}^{\text{code}}(\text{us}(\text{succ}), x) && \text{reduction for circle-recursion} \\ &= x + 1 && \text{reduction for us}\end{aligned}$$

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

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

[DRAFT OF MARCH 19, 2013]

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

Definition 7.2.3. Define $\text{encode} : \prod (x : S^1), \rightarrow P(x) \rightarrow \text{code}(x)$ by

$$\text{encode } p := \text{transport}^{\text{code}}(p, 0)$$

(we leave the argument x implicit).

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

$$\text{loop} \circ \text{loop}^{-1} \circ \text{loop} \circ \dots$$

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

$$\{ \rightarrow + 1 \} \circ \{ \rightarrow - 1 \} \circ \{ \rightarrow + 1 \} \circ \dots$$

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

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

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

Definition 7.2.4. Define $\text{loop}'' : \mathbb{Z} \rightarrow \text{base} = \text{base}$ by

$$\text{loop}'' = \begin{cases} \text{loop} \circ \text{loop} \circ \dots \circ \text{loop} \text{ (n times)} & \text{for positive } n \\ \text{loop}^{-1} \circ \text{loop}^{-1} \circ \dots \circ \text{loop}^{-1} \text{ (n times)} & \text{for negative } n \\ \text{refl} & \text{for } 0 \end{cases}$$

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

[DRAFT OF MARCH 19, 2013]

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

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

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

$$\begin{aligned}\text{transport}^{(\text{code} \circ \text{loop} \circ \text{code})}(\text{loop}, \text{loop}'') &= \text{transport}^{\text{loop} \circ \text{loop}''} \circ \text{transport}^{\text{code}} \text{loop}^{-1} \\ &= \{ \rightarrow + \text{loop} \} \circ \{ \text{loop}'' \} \circ \text{transport}^{\text{code}} \text{loop}^{-1} \\ &= \{ \rightarrow + \text{loop} \} \circ \{ \text{loop}'' \} \circ \{ \rightarrow - 1 \} \\ &= \{ n \mapsto \text{loop}^{n-1} \circ \text{loop}'' \}\end{aligned}$$

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

7.2.1.1.3 Decoding after encoding

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

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

7.2.1.1.4 Encoding after decoding

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

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

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

$$\text{encode}'(\text{loop}'' n) = n$$

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

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

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

- The cases for negatives are analogous. \square

7.2.1.1.5 Tying it all together

Theorem 7.2.8. There is a family of equivalences $\prod (x : S^1) \prod [P(x) \simeq \text{code}(x)]$.

Proof. The maps encode and decode are mutually inverse by Lemmas 7.2.6 and 7.2.7, and this can be improved to an equivalence. \square

Instantiating at base gives

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

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

Corollary 7.2.10. $\pi_k(S^1) = \mathbb{Z}$ if $k = 1$ and 0 otherwise.

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

[DRAFT OF MARCH 19, 2013]

Cover $x = S^1 \rightarrow \text{rec } \text{Int} \text{ (} \lambda u \text{ succEquiv) } x$

```
transport-Cover-loop : Path (transport Cover loop) succ
transport-Cover-loop =
  transport Cover loop
  <= transport-ap-assoc Cover loop >
  transport (λ x → x) (ap Cover loop)
  <= ap (transport (λ x → x)) (ap (λ Cover loop)
    (gloop/rec Int (λ u succEquiv))) >
  transport (λ x → x) (λ u succEquiv)
  <= typeβ _ >
  succ *
```

```
transport-Cover-ll-loop : Path (transport Cover (l loop)) pred
transport-Cover-ll-loop =
  transport Cover (l loop)
  <= transport-ap-assoc Cover (l loop) >
  transport (λ x → x) (ap Cover (l loop))
  <= ap (transport (λ x → x)) (ap (l Cover loop)) >
  transport (λ x → x) (l (ap Cover loop))
  <= ap (λ y → transport (λ x → x) (l y))
    (gloop/rec Int (λ u succEquiv))) >
  transport (λ x → x) (l (λ u succEquiv))
  <= ap (transport (λ x → x)) (l (λ u succEquiv)) >
  transport (λ x → x) (λ u succEquiv)
  <= typeβ _ >
  pred *
```

```
encode : (x : S1) → Path base x → Cover x
encode a = transport Cover a Zero
```

```
encode' : Path base base → Int
encode' a = encode (base) a
```

```
loopA : Int → Path base base
loopA Zero = id
loopA (Pos One) = loop
loopA (Pos (S n)) = loop · loopA (Pos n)
loopA (Neg One) = l loop
loopA (Neg (S n)) = l loop · loopA (Neg n)
```

```
loopA-preserves-pred
: (n : Int) → Path (loopA (pred n)) (l loop · loopA n)
loopA-preserves-pred (Pos One) = l (l-inv-1 loop)
loopA-preserves-pred (Pos (S y)) =
  l (l-assoc (l loop) loop (loopA (Pos y)))
  · l (ap (λ x → x · loopA (Pos y)) (l-inv-1 loop))
  · l (l-unit-1 (loopA (Pos y)))
loopA-preserves-pred Zero = id
loopA-preserves-pred (Neg One) = id
loopA-preserves-pred (Neg (S y)) = id
```

```
decode : (x : S1) → Cover x → Path base x
decode (x) =
```

S^1 -induction

$(\lambda x' \rightarrow \text{Cover } x' \rightarrow \text{Path base } x')$

loop^A

loop^A-respects-loop

x where

abstract -- prevent Agda from normalizing

```
loopA-respects-loop : transport (λ x' → Cover x' → Path base x') loop loopA = (λ n → loopA n)
loopA-respects-loop =
  (transport (λ x' → Cover x' → Path base x') loop loopA
  <= transport-- Cover (Path base) loop loopA >
  transport (λ x' → Path base x') loop
  <= loopA
  <= transport Cover (l loop)
  <= λ y → transport-Path-right loop (loopA (transport Cover (l loop) y))) >
  (λ p → loop · p)
  <= loopA
  <= transport Cover (l loop)
  <= λ y → ap (λ x' → loop · loopA x') (ap= transport-Cover-ll-loop)) >
  (λ p → loop · p)
  <= loopA
  <= pred
  <= id >
  (λ n → loop · (loopA (pred n)))
  <= λ y → move-left-l _ loop (loopA y) (loopA-preserves-pred y)) >
  (λ n → loopA n)
  <= *
```

abstract -- prevent Agda from normalizing

```
encode-loopA : (n : Int) → Path (encode (loopA n)) n
encode-loopA Zero = id
encode-loopA (Pos One) = ap= transport-Cover-loop
encode-loopA (Pos (S n)) =
  encode (loopA (Pos (S n)))
  <= id >
  transport Cover (loopA (Pos n)) Zero
  <= ap= (transport-- Cover loop (loopA (Pos n))) >
  transport Cover loop
  (transport Cover (loopA (Pos n)) Zero)
  <= ap= transport-Cover-loop >
  succ (transport Cover (loopA (Pos n)) Zero)
  <= id >
  succ (encode (loopA (Pos n)))
  <= ap succ (encode-loopA (Pos n)) >
  succ (Pos n) *
encode-loopA (Neg One) = ap= transport-Cover-ll-loop
encode-loopA (Neg (S n)) =
  transport Cover (l loop · loopA (Neg n)) Zero
  <= ap= (transport-- Cover (l loop) (loopA (Neg n))) >
  transport Cover (l loop) (transport Cover (loopA (Neg n)) Zero)
  <= ap= transport-Cover-ll-loop >
  pred (transport Cover (loopA (Neg n)) Zero)
  <= ap pred (encode-loopA (Neg n)) >
  pred (Neg n) *
```

```
encode-decode : (x : S1) → (c : Cover x)
  → Path (encode (decode (x) c)) c
encode-decode (x) = S1-induction
  (λ (x : S1) → (c : Cover x)
    → Path (encode (x) (decode (x) c)) c)
encode-loopA (λ x' → fst (use-level (use-level HSet-Int _ _) _ _)) x
```

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

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

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

```
n[S1]-is-Int : x One S1 base = Int
n[S1]-is-Int = UnTrunc.path _ _ HSet-Int · ap (Trunc (λ l 0)) Ω[S1]-is-Int
```

Outline

1. Computer-checked homotopy theory

2. Computer-checked software

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)}]

Spec: Conversion is a *bijection*:
converting back and forth
doesn't change the data

Type theory

```
conv1 : (Nat × String × ((Nat × Nat) × Nat))
       → (Nat × String × ((Nat × Nat) × Nat))
conv1 (key , name , ((x , y) , year)) =
      (key , name , ((y , x) , year))

convert : DB → DB
convert = map conv1

map-fusion : ∀ {A B C} (g : B → C)
            (f : A → B) (l : List A)
            → map (g ∘ f) l = map g (map f l)
map-fusion g f [] = id
map-fusion g f (x :: xs) =
  ap (_::_ (g (f x))) (map-fusion g f xs)

map-idfunc : ∀ {A} (l : List A) → map (\ x → x) l = l
map-idfunc [] = id
map-idfunc (x :: xs) = ap (_::_ x) (map-idfunc xs)

convert-inv : convert ∘ convert = (λ x → x)
convert-inv = map conv1 ∘ map conv1
            =⟨ ! (λ= (map-fusion conv1 conv1)) ⟩
            map (conv1 ∘ conv1)
            =⟨ id ⟩
            map (\ x → x)
            =⟨ λ= map-idfunc ⟩
            (λ x → x) ■

convert-bijection : Bijection DB DB
convert-bijection =
  (convert ,
   is-bijection convert
    (λ x → (ap= convert-inv))
    (λ x → (ap= convert-inv))))
```

Homotopy Type Theory

```
swapf : (Nat × Nat) → (Nat × Nat)
swapf (x , y) = (y , x)

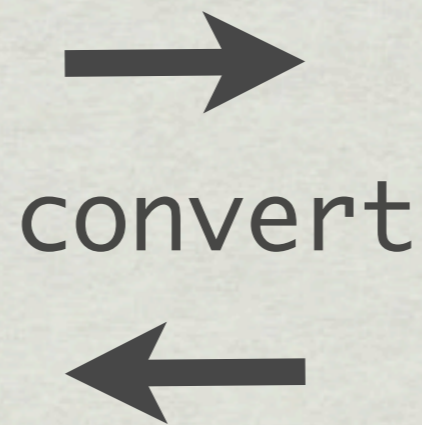
swap : Bijection (Nat × Nat) (Nat × Nat)
swap = (swapf ,
       is-bijection swapf (λ _ → id) (λ _ → id))

There : Type → Type
There A = List (Nat × String × A × Nat)

convert : DB → DB
convert = cast There swap

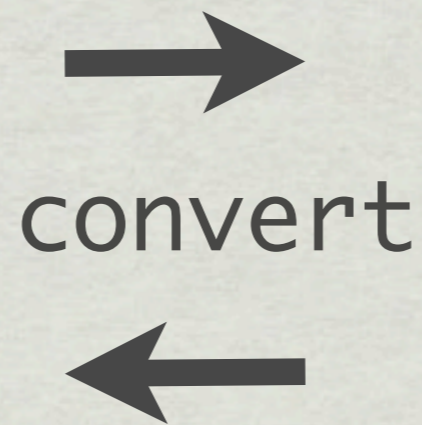
convert-bijection : Bijection DB DB
convert-bijection =
  (convert , cast-is-bijection There swap)
```

[{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)}]

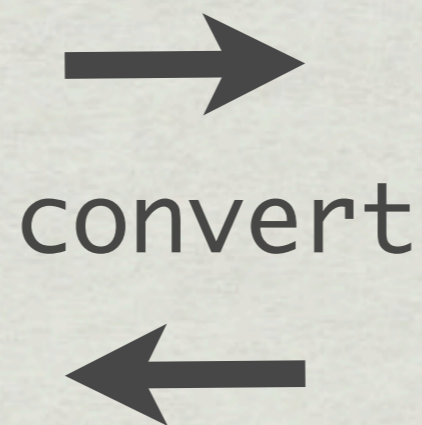


[{key=4,n="John",d=(5,30,1956)},
{key=8,n="Hugo",d=(12,29,1978)},
{key=15,n="James",d=(7,1,1968)},
{key=16,n="Sayid",d=(10,2,1967)},
{key=23,n="Jack",d=(12,3,1969)},
{key=42,n="Sun",d=(3,20,1980)}]

1. Define a function

$\text{swap}(x, y) = (y, x)$

[{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)}]



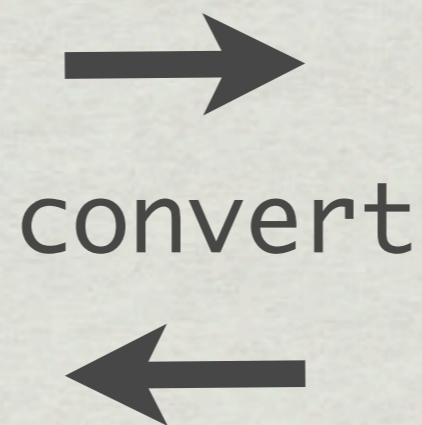
[{key=4,n="John",d=(5,30,1956)},
{key=8,n="Hugo",d=(12,29,1978)},
{key=15,n="James",d=(7,1,1968)},
{key=16,n="Sayid",d=(10,2,1967)},
{key=23,n="Jack",d=(12,3,1969)},
{key=42,n="Sun",d=(3,20,1980)}]

1. Define a function

$\text{swap}(x, y) = (y, x)$

2. Prove that swap is a bijection (it's self-inverse)

[{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

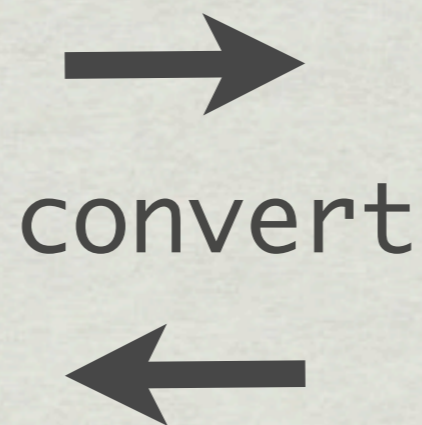
$\text{swap}(x, y) = (y, x)$

2. Prove that swap is a bijection (it's self-inverse)

3. Define a *parametrized type* describing where to swap:

$\text{There}(X) = \text{List}\{\text{key}:\text{int}, \text{ n}:\text{string}, \text{ d}:X \times \text{int}\}$

```
[{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

`swap(x,y) = (y,x)`

2. Prove that `swap` is a bijection (it's self-inverse)

3. Define a *parametrized type* describing where to swap:

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

4. Define

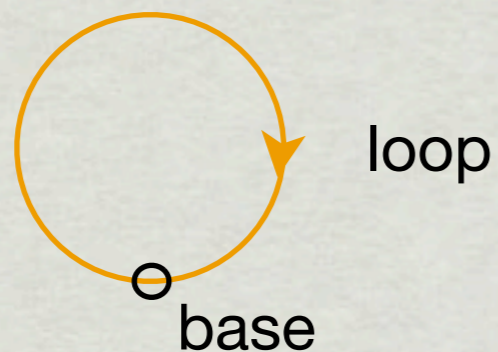
`convert(db) = castThere(swap,db)`

Types write code
and proofs for you

Functions on spaces

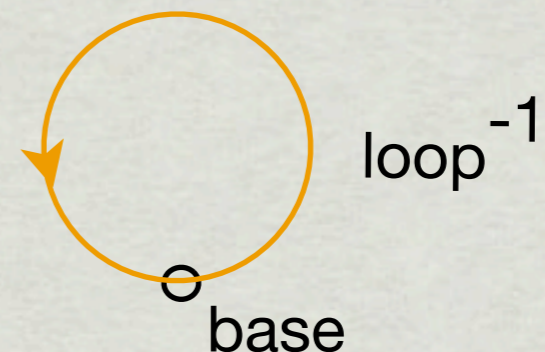
Function on a space gives the image of each point

Circle



reverse

Circle



and each path!

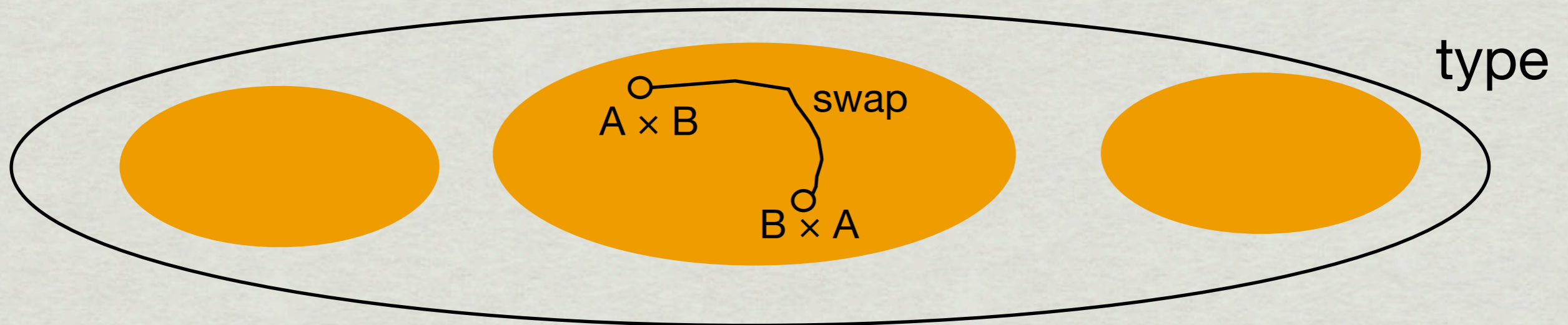
Functions on types

Functions on types

1. $\text{There}(X) = \text{List}\{\text{key}:\text{int}, \text{ n}:\text{string}, \text{ d}:X \times \text{int}\}$
is a function on the **space of types**, so it must also
give an image for each path between types

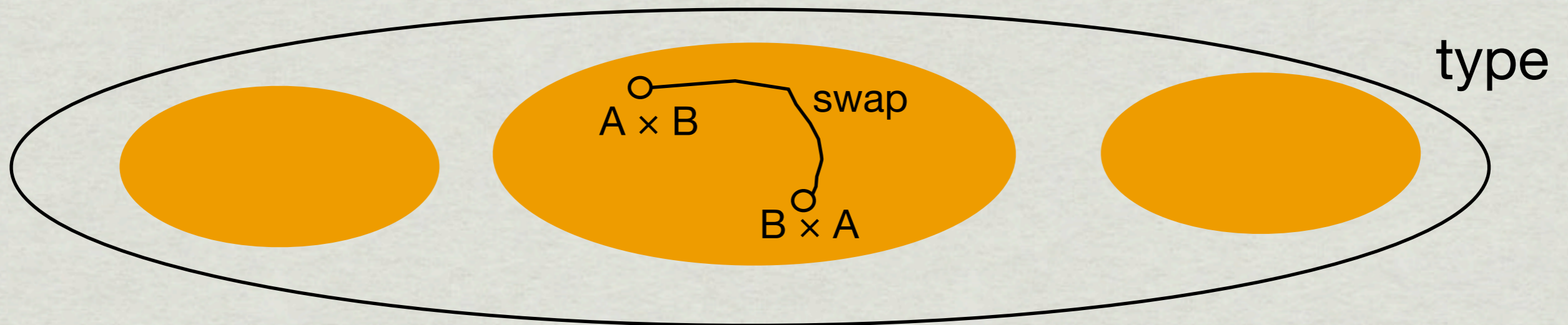
Functions on types

1. $\text{There}(X) = \text{List}\{\text{key}:\text{int}, \text{ n}:\text{string}, \text{ d}:X \times \text{int}\}$
is a function on the **space of types**, so it must also
give an image for each path between types
2. We define the paths between types to be **bijections**



Functions on types

1. $\text{There}(X) = \text{List}\{\text{key}:\text{int}, \text{ n}:\text{string}, \text{ d}:X \times \text{int}\}$
is a function on the **space of types**, so it must also
give an image for each path between types
2. We define the paths between types to be **bijections**



3. \therefore There gives an image for swap

Cast

`castThere(swap)`

applies `There` to `swap`: a *type-directed program* that builds bigger bijections from smaller ones

Computational interpretation of cast:

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

cast**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 cast:

`There(X)=List{key:int, n:string, d:X×int}`
`castThere(swap, db)`
`= map (castThere1 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 cast:

$\text{There1}(X) = \{\text{key}:\text{int}, \text{n}:\text{string}, \text{d}:X \times \text{int}\}$

$\text{cast}_{\text{There}}(\text{swap}, \text{db})$
 $= \text{map } (\text{cast}_{\text{There1}} \text{ swap}) \text{ db}$

[{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 cast:

$\text{There1}(X) = \{\text{key}:\text{int}, \text{n}:\text{string}, \text{d}:X \times \text{int}\}$

$\text{cast}_{\text{There}}(\text{swap}, \text{db})$
 $= \text{map } (\text{cast}_{\text{There1}} \text{ swap}) \text{ db}$
 $= \text{map } (\{\text{key}, \text{n}, (\text{d}, \text{m}, \text{y})\} \Rightarrow$
 $\quad \{\text{key}, \text{n}, (\text{m}, \text{y})\}) \text{ db}$

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



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

Computational interpretation of cast:

$\text{There1}(X) = \{\text{key}:\text{int}, \text{n}:\text{string}, \text{d}:X \times \text{int}\}$

$\text{cast}_{\text{There}}(\text{swap}, \text{db})$
 $= \text{map } (\text{cast}_{\text{There1}} \text{ swap}) \text{ db}$
 $= \text{map } (\{\text{key}, \text{n}, (\text{d}, \text{m}, \text{y})\} \Rightarrow$
 $\quad \{\text{key}, \text{n}, (\text{cast}_{\text{Here}}(\text{swap}, (\text{d}, \text{m})), \text{y})\}) \text{ db}$

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



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

Computational interpretation of cast:

Here(X)=X

cast^{There}(swap, db)
= map (cast^{There1} swap) db
= map ({key,n,(d,m,y)} =>
 {key,n,(cast^{Here}(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 cast:

Here(X)=X

cast^{There}(swap, db)
= map (cast^{There1} swap) db
= map ({key,n,(d,m,y)} =>
 {key,n,(cast^{Here}(swap,(d,m)),y)}) db
= 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)}]

Type theory

```
conv1 : (Nat × String × ((Nat × Nat) × Nat))
       → (Nat × String × ((Nat × Nat) × Nat))
conv1 (key , name , ((x , y) , year)) =
      (key , name , ((y , x) , year))
```

```
convert : DB → DB
convert = map conv1
```

```
map-fusion : ∀ {A B C} (g : B → C)
             (f : A → B) (l : List A)
             → map (g ∘ f) l = map g (map f l)
```

```
map-fusion g f [] = id
map-fusion g f (x :: xs) =
  ap (_::_ (g (f x))) (map-fusion g f xs)
```

```
map-idfunc : ∀ {A} (l : List A) → map (\ x → x) l = l
map-idfunc [] = id
map-idfunc (x :: xs) = ap (_::_ x) (map-idfunc xs)
```

```
convert-inv : convert ∘ convert = (λ x → x)
convert-inv = map conv1 ∘ map conv1
              =⟨ ! (λ= (map-fusion conv1 conv1)) ⟩
              map (conv1 ∘ conv1)
              =⟨ id ⟩
              map (\ x → x)
              =⟨ λ= map-idfunc ⟩
              (λ x → x) ■
```

```
convert-bijection : Bijection DB DB
convert-bijection =
  (convert ,
   is-bijection convert
    (λ x → (ap= convert-inv))
    (λ x → (ap= convert-inv))))
```

Homotopy Type Theory

```
swapf : (Nat × Nat) → (Nat × Nat)
swapf (x , y) = (y , x)
```

```
swap : Bijection (Nat × Nat) (Nat × Nat)
swap = (swapf ,
        is-bijection swapf (λ _ → id) (λ _ → id)))
```

```
There : Type → Type
There A = List (Nat × String × A × Nat)
```

```
convert : DB → DB
convert = cast There swap
```

```
convert-bijection : Bijection DB DB
convert-bijection =
  (convert , cast-is-bijection There swap)
```

*Writes **proofs** for you!*

Identity and composition for $\Gamma \vdash \theta : \Delta$

$$\frac{\Gamma \vdash \Delta \quad \Gamma_2 \vdash \theta_2 : \Gamma_3 \quad \Gamma_1 \vdash \theta_1 : \Gamma_2 \quad \Gamma \vdash \theta : \Delta \quad \Gamma_0 \vdash \delta : \theta_1 \simeq_{\Gamma} \theta_2}{\Gamma \vdash \text{id}_{\Delta}^{\theta} : \Delta \quad \Gamma_1 \vdash \theta_2[\theta_1] : \Gamma_3 \quad \Gamma_0 \vdash \delta[\delta] : \delta[\theta_1] \simeq_{\Delta} \theta[\theta_2]} \quad \begin{array}{l} \theta_0[\theta[\theta']] = \theta_0[\theta][\theta'] \quad l\text{-subst assoc/unit} \\ \theta_0[\text{id}_{\Delta}^{\theta}] = \theta_0 \\ \text{id}_{\Delta}^{\theta}[\theta] = \theta \end{array}$$

$$\frac{\theta[\delta][\delta']} = \theta[\delta][\delta'] \quad l\text{-resp assoc} \quad \frac{\theta[\text{refl}_{\theta}]} = \text{refl}_{\theta[\theta]'} \quad l\text{-resp preserves refl} \quad \frac{\theta[\theta'][\delta]} = \theta[\theta][\delta] \quad l\text{-resp for l-subst}$$

Identity, Inverses, and Composition for $\Gamma \vdash \delta : \theta \simeq_{\Delta} \theta'$

$$\frac{\Gamma \vdash \delta : \theta_1 \simeq_{\Delta} \theta_2 \quad \Gamma \vdash \delta_1 : \theta_1 \simeq_{\Delta} \theta_2 \quad \Gamma \vdash \delta : \theta \simeq_{\Delta} \theta' \quad \Gamma_0 \vdash \delta_0 : \theta_0 \simeq_{\Gamma} \theta'_0}{\Gamma \vdash \text{refl}_{\theta}^{\delta} : \theta \simeq_{\Delta} \theta \quad \Gamma \vdash \delta^{-1} : \theta_2 \simeq_{\Delta} \theta_1 \quad \Gamma \vdash \delta_2 \circ \delta_1 : \theta_1 \simeq_{\Delta} \theta_3 \quad \Gamma_0 \vdash \delta[\delta_0] : \theta[\theta_0] \simeq_{\Delta} \theta'[\theta'_0]} \quad \begin{array}{l} (\delta_3 \circ \delta_2) \circ \delta_1 = \delta_3 \circ (\delta_2 \circ \delta_1) \quad \text{trans associativity} \\ (\delta \circ \text{refl}) = \delta \\ (\text{refl} \circ \delta) = \delta \\ (\delta \circ \delta^{-1}) = \text{refl} \\ (\delta^{-1} \circ \delta) = \text{refl} \end{array} \quad \begin{array}{l} \delta_0[\delta[\delta']] = \delta_0[\delta][\delta'] \quad 2\text{-resp associativity} \\ \delta_0[\text{refl}_{\theta}] = \delta_0 \\ \text{refl}_{\Delta}^{\delta}[\delta] = \delta \\ (\delta_1 \circ \delta_2)[\delta_3 \circ \delta_4] = \delta_1[\delta_2] \circ \delta_2[\delta_3] \quad \text{interchange delegate} \\ \text{refl}_{\Delta}^{\delta}[\delta] = \theta[\delta] \end{array}$$

Composition for $\Gamma \vdash A$ type

$$\frac{\Gamma \vdash \theta : \Delta \quad \Delta \vdash A \text{ type} \quad \Delta \text{ ctx} \quad \Delta \vdash C \text{ type} \quad \Gamma \vdash \delta : \theta_1 \simeq_{\Delta} \theta_2 \quad \Gamma \vdash M : C[\theta_2]}{\Gamma \vdash A[\theta] \text{ type} \quad \Gamma \vdash M[\theta] : A[\theta]}$$

$$\text{map}_{\Delta} \quad \text{map}_{\Delta, C} \quad \text{map}_{\Delta, C} \quad \text{map}_{\Delta, C} \quad \text{map}_{\Delta, C} \quad \text{map}_{\Delta, C}$$

Composition for $\Gamma \vdash M : A$

$$\frac{\Gamma \vdash \theta : \Delta \quad \Delta \vdash M : A}{\Gamma \vdash M[\theta] : A[\theta]}$$

Identity, Inverses, and Composition for $\Gamma \vdash M : A$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M' : A}{\Gamma \vdash \text{refl}_M^M : M \simeq_A M} \quad \begin{array}{l} (\alpha_3 \circ \alpha_2) \circ \alpha_1 = \alpha_3 \circ (\alpha_2 \circ \alpha_1) \\ (\alpha \circ \text{refl}) = \alpha \\ (\text{refl} \circ \alpha) = \alpha \\ (\alpha \circ \alpha^{-1}) = \text{refl} \\ (\alpha^{-1} \circ \alpha) = \text{refl} \\ \alpha[\delta[\delta']] = \alpha[\delta][\delta'] \\ \alpha[\text{refl}_{\theta}] = \alpha \\ (\alpha_1 \circ \alpha_2)[\delta_3 \circ \delta_4] = \alpha_1[\delta_2] \circ \alpha_2[\delta_3] \\ \text{refl}_M^M[\delta] = \text{refl}_M^M \end{array}$$

Omitted Rules: All judgements respect

$$\text{Derived forms: } \frac{\text{resp}(x.F) \alpha}{\text{map}_{\Delta, A, B}^{\alpha} \alpha M} \quad \text{trans means } \frac{(x.F)[\text{refl}_{\Delta}, \alpha/x]}{\text{map}_{\Gamma, \Gamma \vdash A, B}^{\alpha} (\text{refl}_{\Delta}, \alpha/x) M}$$

Figure 1. Judgemental Presentation of Equivalence

Empty context:

$$\frac{\theta : \cdot = \text{id} \quad l\text{-}\eta}{\delta : \theta \simeq \theta' = \text{refl} \quad 2\text{-}\eta}$$

Term variables:

$$\frac{\Gamma \text{ ctx} \quad \Gamma \vdash A \text{ type} \quad x:A \in \Gamma \quad \Gamma \vdash \theta : \Delta \quad \Gamma \vdash M : A[\theta]}{\Gamma \vdash x:A \text{ ctx} \quad \Gamma \vdash x:A \quad \Gamma \vdash \theta, M/x : \Delta, x:A \quad \Gamma \vdash \{\delta, \alpha/x\} : \{\theta, M/x\} \simeq_{\Delta, x:A} \{\theta', N/x\}} \quad \begin{array}{l} \text{id}_{\Gamma, x:A}^{\theta, M/x} = \theta \quad l\text{-}\beta \\ x[\theta, M/x] = M \quad l\text{-}\beta \\ \theta : (\Gamma, x:A) = \text{id}_{\Gamma}[\theta], x[\theta]/x \quad l\text{-}\eta \\ \text{id}_{\Gamma, x:A}^{\theta, \alpha/x} = \delta \quad 2\text{-}\beta \\ x[\delta, \alpha/x] = \alpha \quad 2\text{-}\beta \\ \delta : \theta \simeq_{(\Gamma, x:A)} \theta' = \text{id}_{\Gamma}[\delta], x[\delta]/x \quad 2\text{-}\eta \end{array} \quad \begin{array}{l} \text{id}_{\Gamma, x:A} = \text{id}_{\Gamma}, x/x \\ (\theta, M/x)[\theta_0] = \theta[\theta_0], M[\theta_0]/x \quad l\text{-subst} \\ (\theta, M/x)[\delta_0] = \theta[\delta_0], M[\delta_0]/x \quad l\text{-resp} \\ \text{refl}_{\theta, M/x} = \text{refl}_{\theta}, \text{refl}_M/x \quad \text{refl} \\ (\delta, \alpha/x) = (\delta^{-1}, (\text{resp}(x, \text{map}_{\Delta, A} \delta^{-1} x) \alpha^{-1})/x) \quad \text{sym} \\ (\delta_2, \alpha_2/x) \circ (\delta_1, \alpha_1/x) = (\delta_2 \circ \delta_1), (\alpha_2 \circ \text{resp}(x, \text{map}_{\Delta, A} \delta_2 x) \alpha_1)/x \quad \text{trans} \\ (\delta, \alpha/x)[\delta_0] = \delta[\delta_0], \alpha[\delta_0]/x \quad 2\text{-resp} \end{array}$$

Figure 2. Contexts

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x:A \vdash B \text{ type} \quad \Gamma \vdash M : A \quad \Gamma \vdash M_1 : B[M/x]}{\Gamma \vdash \Pi x:A. B \text{ type} \quad \Gamma \vdash \lambda x. M : \Pi x:A. B \quad \Gamma \vdash M_1 : B[M/x]} \quad \frac{\Gamma \vdash M_1 : \Pi x:A. B \quad \Gamma \vdash M_2 : B[M_1/x]}{\Gamma \vdash M_1 M_2 : B[M_1/x]} \quad \frac{\Gamma, x:A \vdash \alpha : (M x) \simeq_B (N x) \quad \Gamma \vdash \alpha : M \simeq_{\Pi x:A. B} N \quad \Gamma \vdash \beta : M_1 \simeq_A N_1}{\Gamma \vdash \alpha \beta : \text{map}_{\Pi x:A. B}^{\beta} \beta (M M_1) \simeq_{B[N_1/x]} (N N_1)}$$

$$\frac{\theta_0, x/x \quad (M_2[\theta_0]) \quad \text{refl}/x \quad \delta}{\theta_0, x/x \quad (M_2[\theta_0]) \quad \text{refl}/x \quad \delta} \quad \begin{array}{l} l\text{-subst} \\ l\text{-subst} \\ l\text{-resp} \\ l\text{-resp} \end{array}$$

$$\frac{\theta_0, x/x \quad (M_2[\theta_0]) \quad \text{refl}/x \quad \delta}{\theta_0, x/x \quad (M_2[\theta_0]) \quad \text{refl}/x \quad \delta} \quad \begin{array}{l} \text{refl} \\ \text{sym} \\ \text{trans} \\ 2\text{-resp} \\ 2\text{-resp} \end{array}$$

$$\frac{\text{true}/x \quad M_2[\text{true}/x] \quad \text{false}/x \quad M_2[\text{false}/x]}{\text{true}/x \quad M_2[\text{true}/x] \quad \text{false}/x \quad M_2[\text{false}/x]} \quad \begin{array}{l} l\text{-subst} \\ l\text{-subst} \\ l\text{-subst} \end{array}$$

$$\frac{\text{true}/x \quad M_2[\text{true}/x] \quad \text{false}/x \quad M_2[\text{false}/x]}{\text{true}/x \quad M_2[\text{true}/x] \quad \text{false}/x \quad M_2[\text{false}/x]} \quad \begin{array}{l} \text{refl} \\ \text{sym} \\ \text{trans} \\ 2\text{-resp} \end{array}$$

Figure 4. Booleans (as an Extensional Set)

$$\frac{\Gamma \vdash \text{set type} \quad \Gamma \vdash \text{El}(S) \text{ type}}{\Gamma \vdash \alpha : M \simeq_{\text{El}(S)} N \quad \Gamma \vdash \alpha : M \simeq_{\text{El}(S)} N} \quad \frac{\Gamma \vdash M : 2 \quad \Gamma \vdash M : \text{El}(\text{bool})}{\Gamma \vdash \text{in } M : \text{El}(\text{bool}) \quad \Gamma \vdash \text{out } M : 2} \quad \begin{array}{l} \text{set}[\theta_0] = \text{set} \quad l\text{-subst} \\ (\text{El}(S))[\theta_0] = \text{El}(S[\theta_0]) \\ \text{map}_{\Delta, \text{set}} \delta M = M \quad l\text{-subst} \\ -^{-1}, - \circ - \text{ for set} \quad \text{groupoid laws plus below} \\ \text{refl}_M : \text{El}(S) \quad \text{is canonical} \\ -^{-1}, - \circ -, 2\text{-resp for El}(S) \quad \text{trivial by ulp} \end{array}$$

Figure 5. Universe

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma \vdash \{M, N\} : A \quad \Gamma \vdash \alpha : M \simeq_A N \quad \Gamma \vdash M : \text{Id}_A M N}{\Gamma \vdash \text{Id}_A M N \text{ type} \quad \Gamma \vdash \text{in } \alpha : \text{Id}_A M N \quad \Gamma \vdash \text{out } M : M \simeq_A N} \quad \frac{\Gamma \vdash \alpha : P \simeq_{\text{Id}_A M N} Q \quad \Gamma \vdash \alpha : P \simeq_{\text{Id}_A M N} Q}{\Gamma \vdash P = Q} \quad \begin{array}{l} \text{out}(\text{in } \alpha) = \alpha \quad l\text{-}\beta \\ \text{in}(\text{out } M) = M \quad l\text{-}\eta \\ (\text{Id}_A M N)[\theta] = \text{Id}_A[\theta] M[\theta] N[\theta] \quad l\text{-subst} \\ \text{map}_{\Delta, \text{Id}_A M N} \delta P = P \quad 0\text{-resp} \\ \text{in}(N[\delta] \circ (\text{resp}(x, \text{map}_A \delta x) (\text{out } P)) \circ M[\delta]^{-1}) \quad 2\text{-resp} \end{array} \quad \begin{array}{l} (\text{in } \alpha)[\theta] = \text{in}(\alpha[\text{refl}_{\theta}]) \quad l\text{-subst} \\ (\text{in } \alpha)[\delta] = \text{in}(\alpha[\delta]) \quad l\text{-resp} \\ \text{refl} \quad \text{is canonical} \\ -^{-1}, - \circ - \quad \text{trivial by extensionality} \\ (\text{out } M)[\delta] \quad \text{stuck until } M \text{ reduces (neutral)} \end{array}$$

Figure 6. Identity Types

More applications

- ✱ For modular code, can reason about a fast implementation using a reference implementation: `cast` a proof about the reference implementation to the fast implementation
- ✱ Can program domain-specific program verification logics, using `cast` to implement the *structural properties* [thesis + MFPS'11]

Conclusion

Types are ∞ -groupoids

type theory

$\langle \text{program} \rangle : \langle \text{type} \rangle$

$\langle \text{proof} \rangle : \langle \text{prog}_1 \rangle = \langle \text{prog}_2 \rangle$

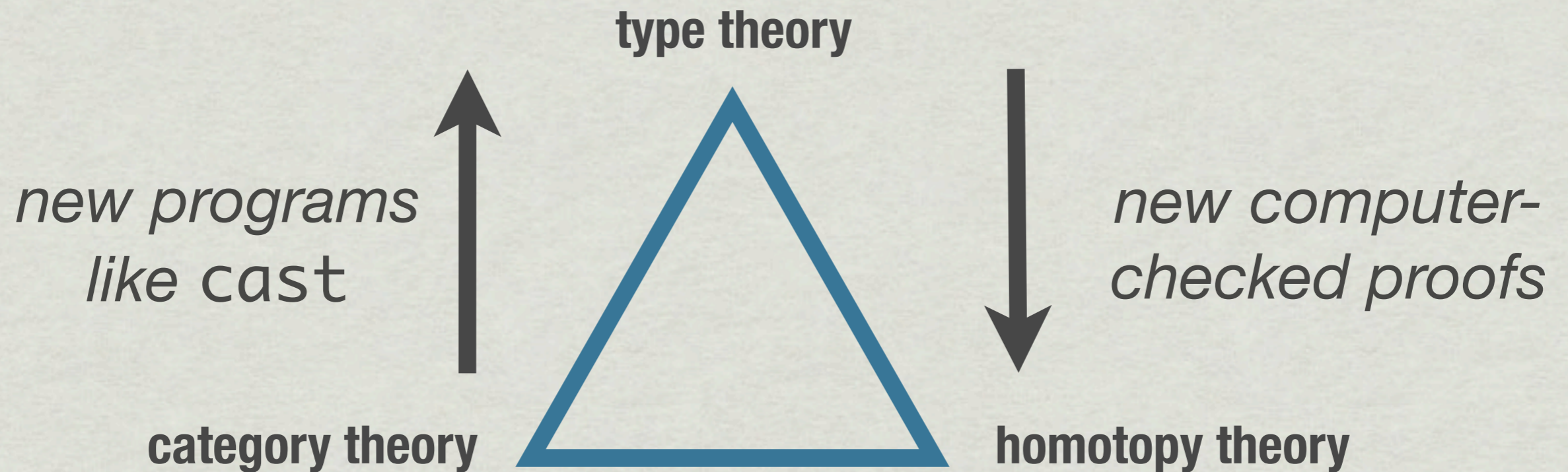
$\langle 2\text{-proof} \rangle : \langle \text{proof}_1 \rangle = \langle \text{proof}_2 \rangle$

$\langle 3\text{-proof} \rangle : \langle 2\text{-proof}_1 \rangle = \langle 2\text{-proof}_2 \rangle$

\vdots

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

Homotopy Type Theory



Papers and code

1. Fundamental group of the circle [LICS'13]
Formal homotopy: github.com/dlicata335/
2. Computational interpretation
of 2D type theory [POPL'12]
3. Domain-specific program verification logics
[thesis+MFPS'11]
4. The HoTT Book (coming soon!): doing math
informally in Homotopy Type Theory
5. Blog: homotopytypetheory.org

Research Agenda

- * Develop a computational interpretation for infinite-dimensional types (in progress)
- * Implement a new proof assistant based on it
- * Computer-checked math, especially in category theory and homotopy theory
- * Computer-checked software

Parallelism and Verification

```
signature SEQUENCE =  
sig  
  type 'a seq  
  
  val length : 'a seq -> int  
  val nth     : int -> 'a seq -> 'a  
  val tabulate : (int -> 'a) -> int -> 'a seq  
  
  val map : ('a -> 'b) -> 'a seq -> 'b seq  
  val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a  
  
end
```

Goal: fast parallel implementation,
proved correct relative to list implementation,
in a proof assistant!

Research Agenda

*Make it easier to use proof assistants
to develop math and software*

- ✱ **PL: languages for expressing mathematics**
- ✱ SE: managing large codebases
- ✱ Compilers + distributed computing: speed
- ✱ Machine learning: automated proof search
- ✱ HCI: usable by “working mathematicians”
- ✱ Graphics: visualization

I am developing
a computational theory of ∞ -groupoids
and applying it to
computer-checked math and software