Programming in Homotopy Type Theory

Dan Licata
Institute for Advanced Study

Joint work with Robert Harper

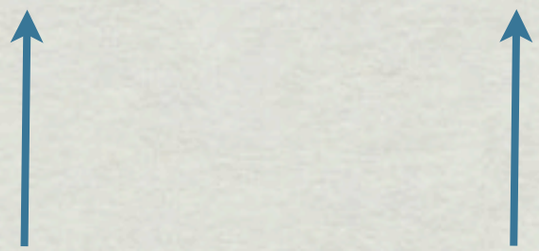
Dependent Type Theory

Dependent Type Theory

Basis of proof assistants (Agda, Coq, NuPRL)

- Formalization of math
- Certified programming

sort : Π xs:int list. Σ ys:int list.Sorted(ys)



Identity Types

"propositional equality"

$$\frac{}{\mathrm{refl} : \mathrm{Id}_A(M,M)} \qquad \frac{C : A \to \mathrm{type} \quad \alpha : \mathrm{Id}_A(M,N) \quad P : C[M]}{\mathrm{subst}_C\ \alpha\ P : C[N]}$$

Computation rule: $\mathrm{subst}_C\ \mathrm{refl}\ P \equiv P$

Identity Types

"propositional equality"

$$\frac{}{\mathrm{refl} : \mathrm{Id}_A(M,M)} \qquad \frac{C : A \to \mathrm{type} \quad \alpha : \mathrm{Id}_A(M,N) \quad P : C[M]}{\mathrm{subst}_C\ \alpha\ P : C[N]}$$

all families respect Identity

Computation rule: $\mathrm{subst}_C\ \mathrm{refl}\ P \equiv P$

Identity Types

"propositional equality"

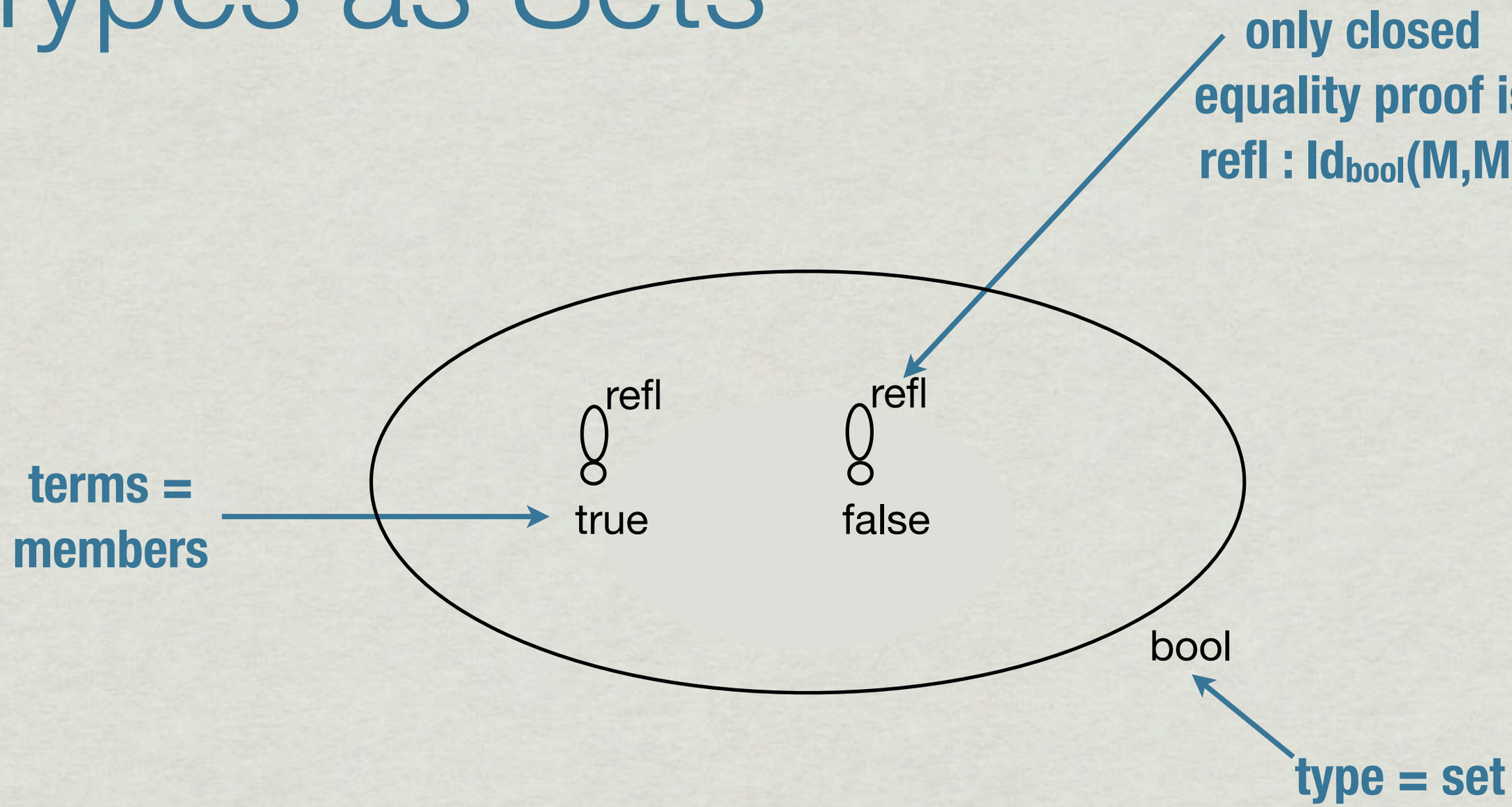
$$\frac{}{\mathrm{refl} : \mathrm{Id}_A(M,M)} \qquad \frac{C : A \to \mathrm{type} \quad \alpha : \mathrm{Id}_A(M,N) \quad P : C[M]}{\mathrm{subst}_C\ \alpha\ P : C[N]}$$

(simplified)

all families respect Identity

Computation rule: $\mathrm{subst}_C\ \mathrm{refl}\ P \equiv P$

Types as Sets



Homotopy Type Theory

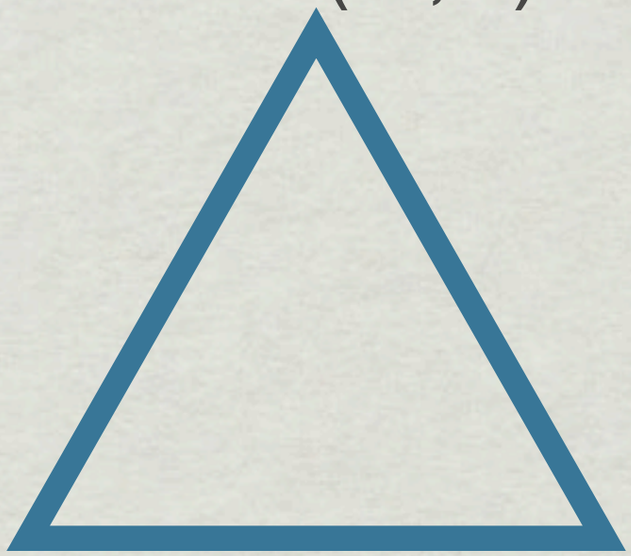
Homotopy Type Theory

dependent type theory

A : type

M, N : A

$\alpha : Id_A(M,N)$



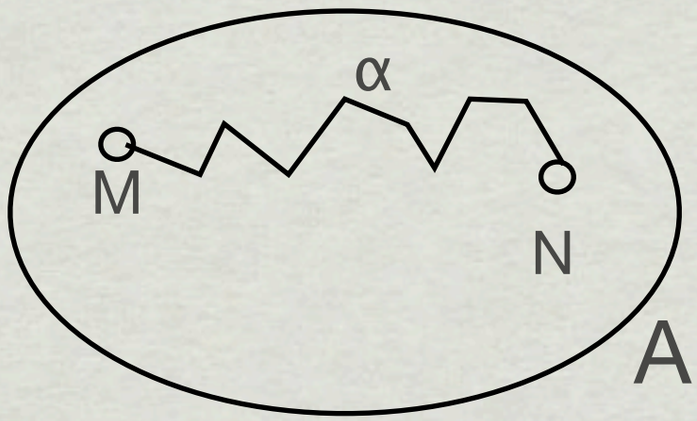
category theory

A is a groupoid

M,N objects

$\alpha : M \to N$ in A

homotopy theory

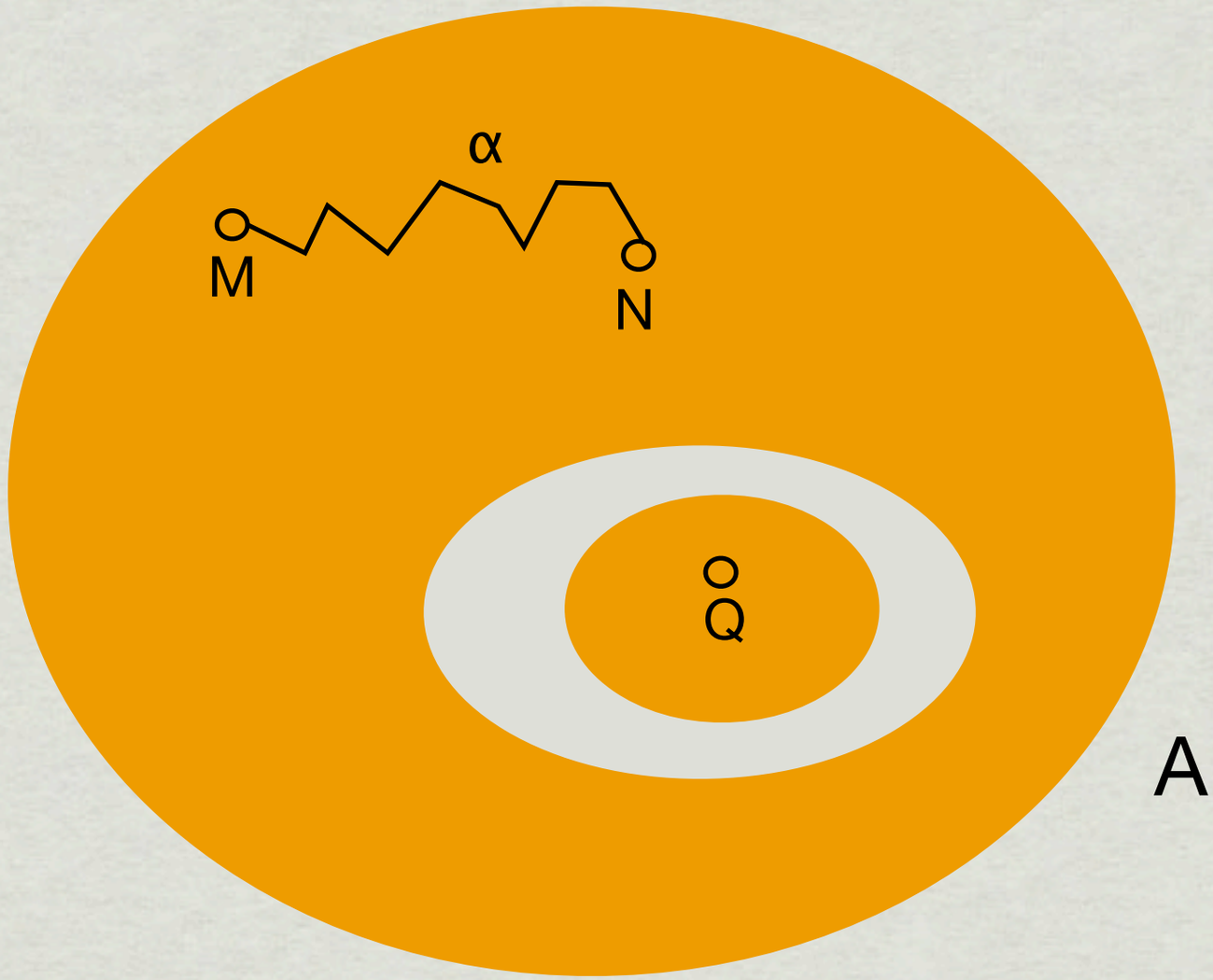


Homotopy Type Theory

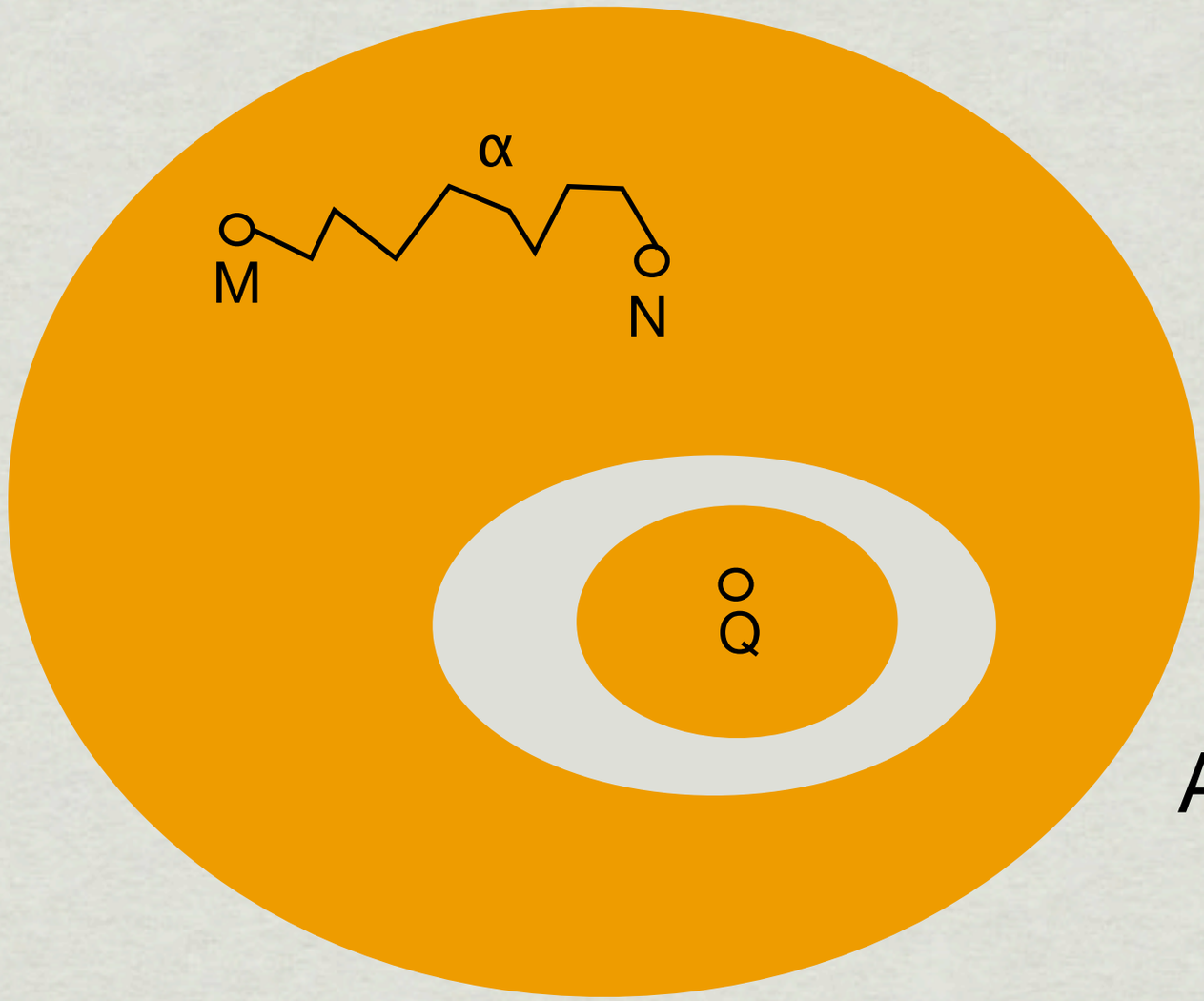
Theorem: Martin-Löf's intensional type theory has semantics in homotopy theory (spaces as types) and category theory (groupoids as types)

[Hofmann&Streicher,Awodey, Warren,Lumsdaine,Garner, Voevodsky, 1990's and 2000's]

Types as Spaces

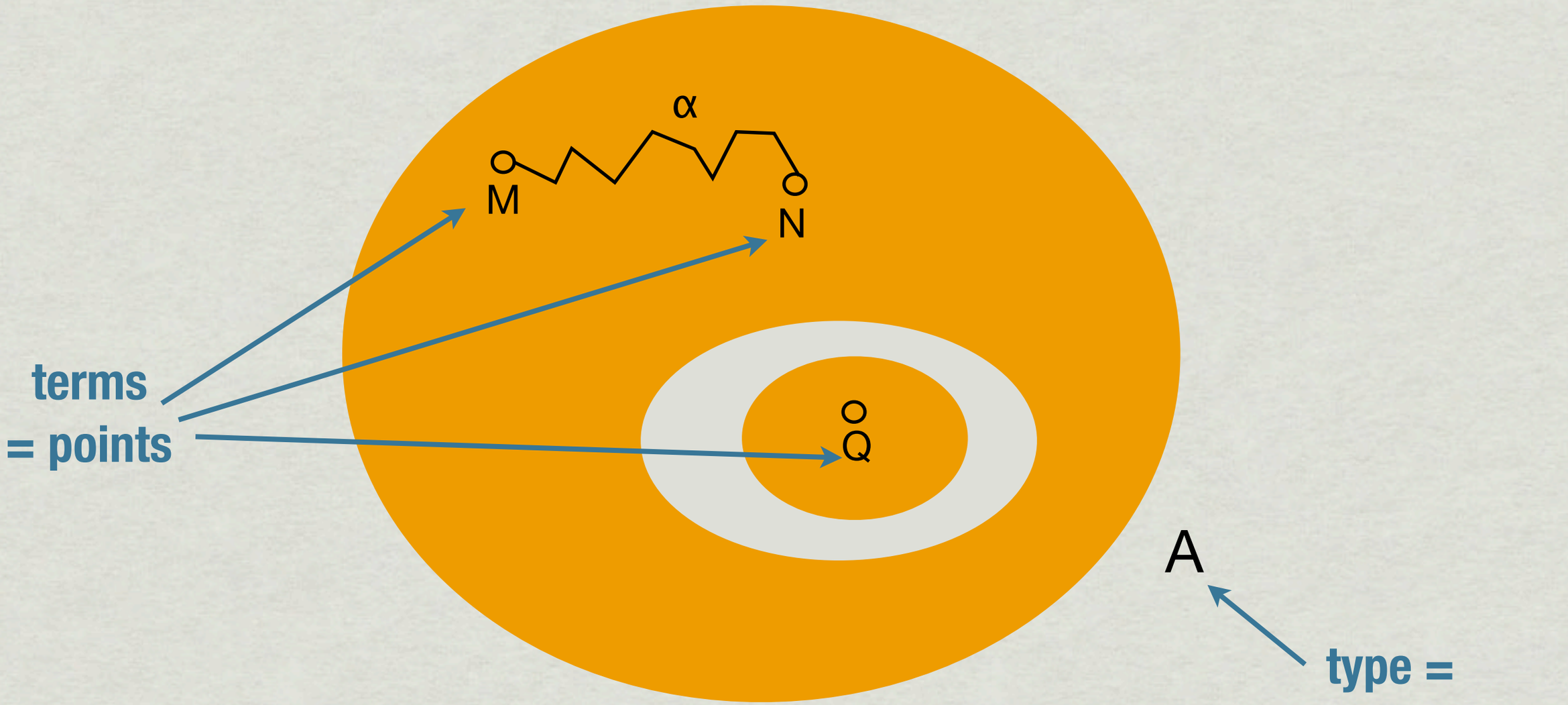


Types as Spaces

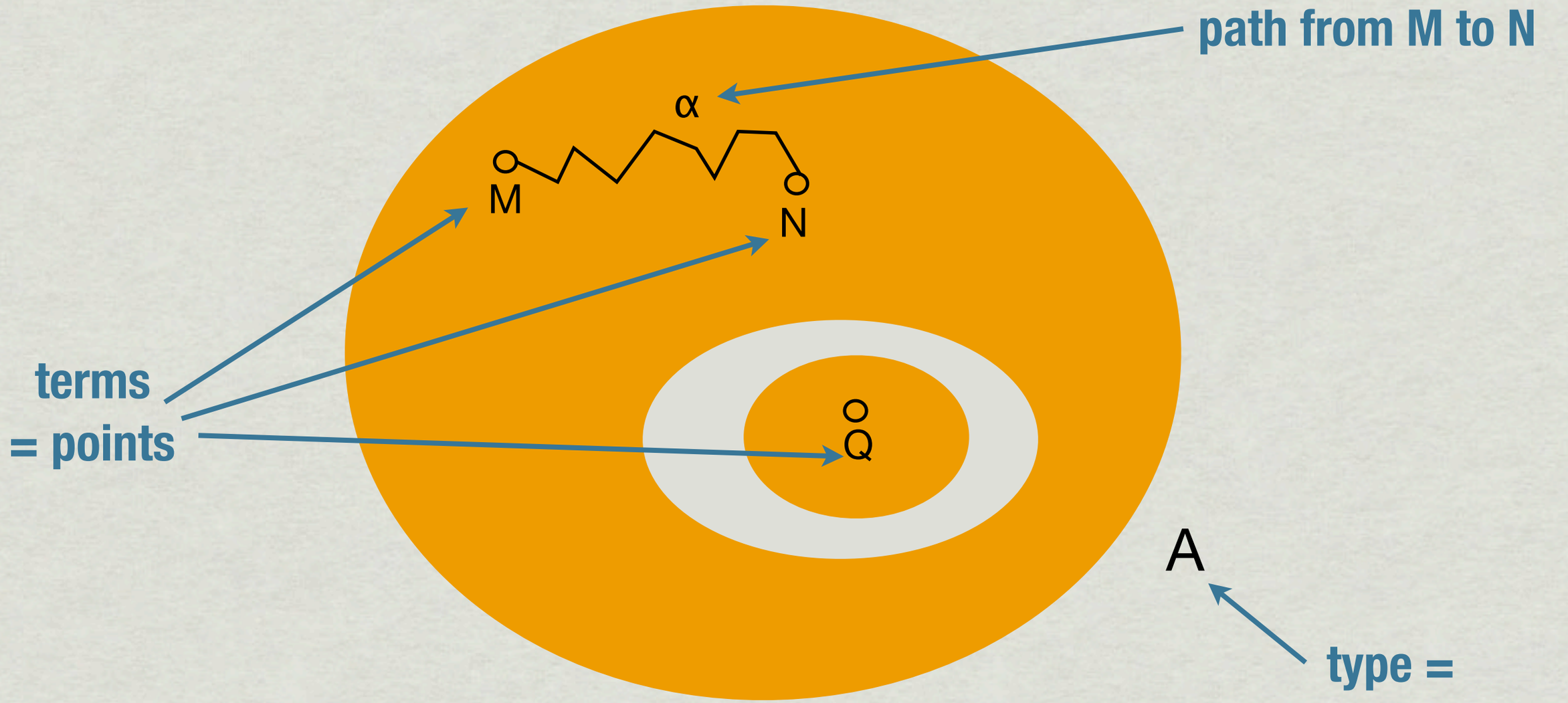


type =
topological space
(up to homotopy)

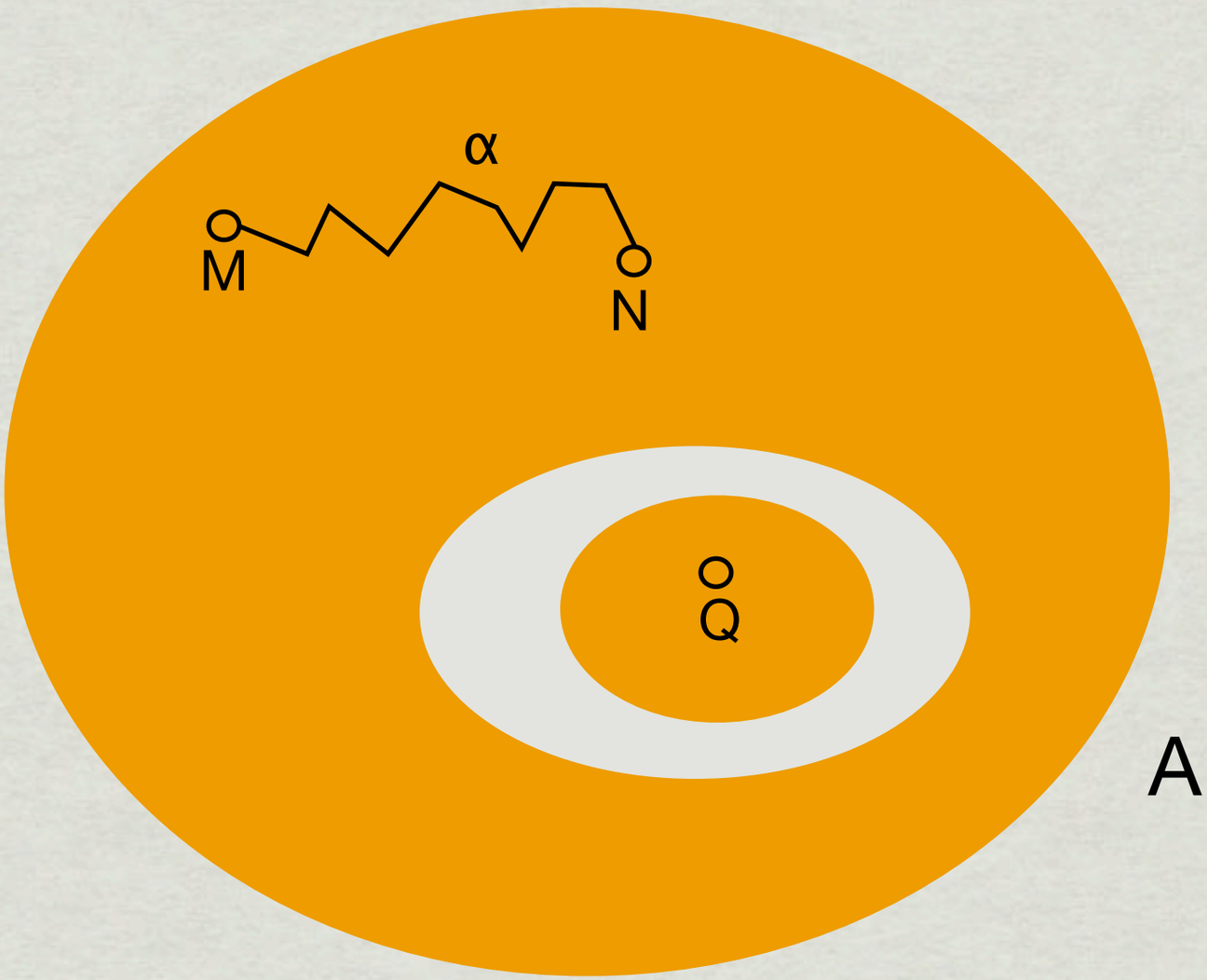
Types as Spaces



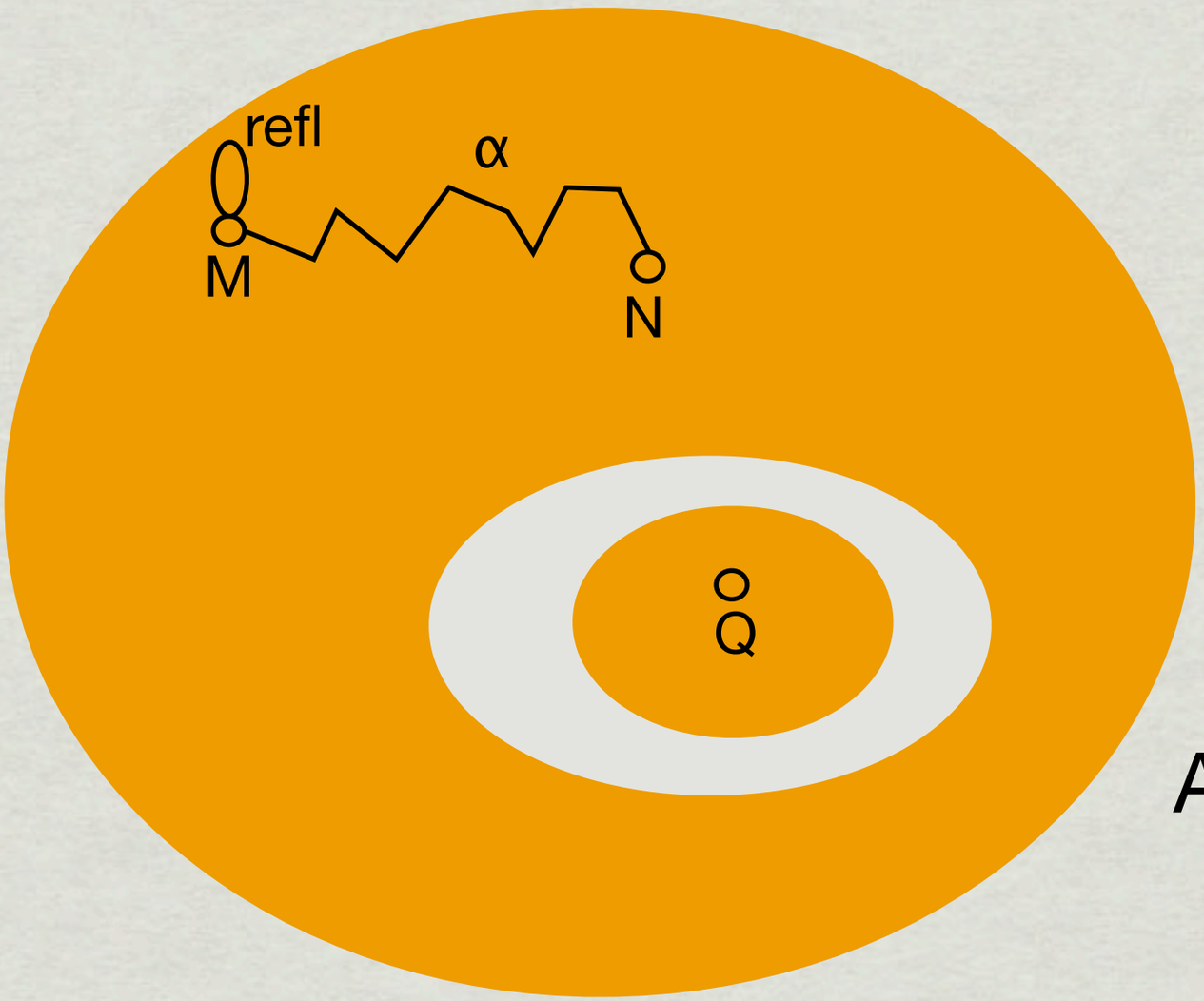
Types as Spaces



Operations on Paths

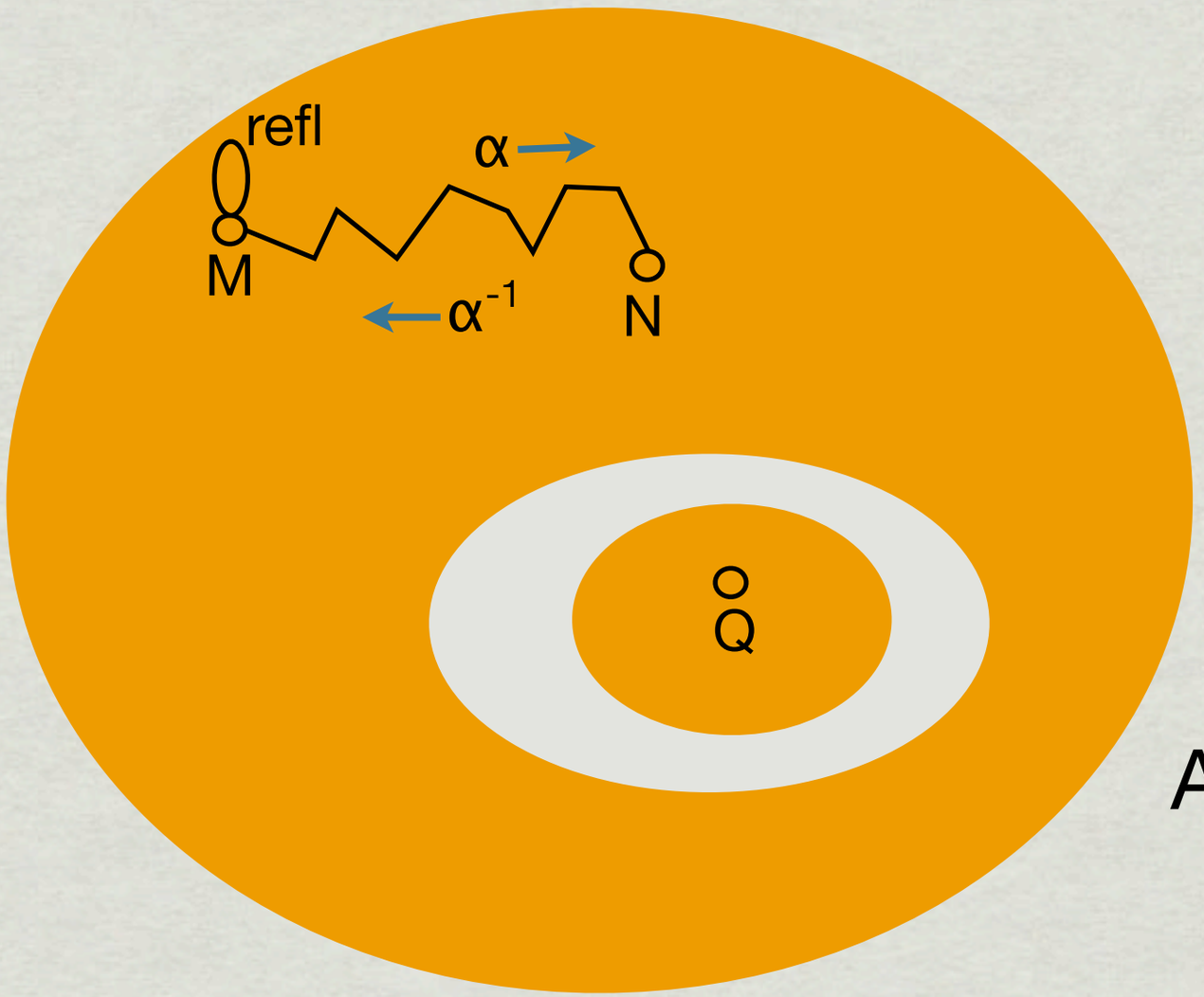


Operations on Paths



refl : $Id_A(M,M)$

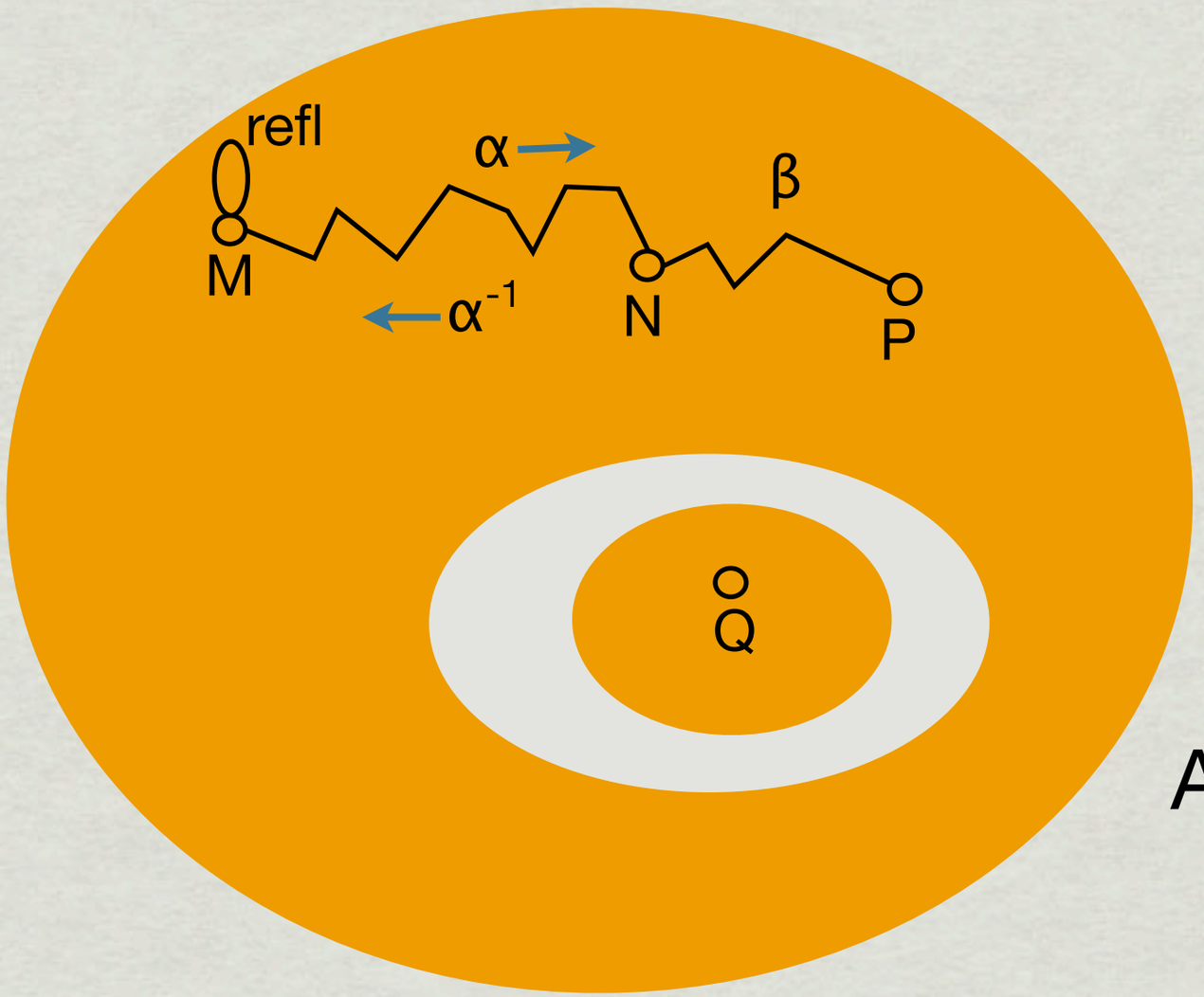
Operations on Paths



refl : $Id_A(M,M)$

$\alpha^{-1} : Id_A(N,M)$

Operations on Paths

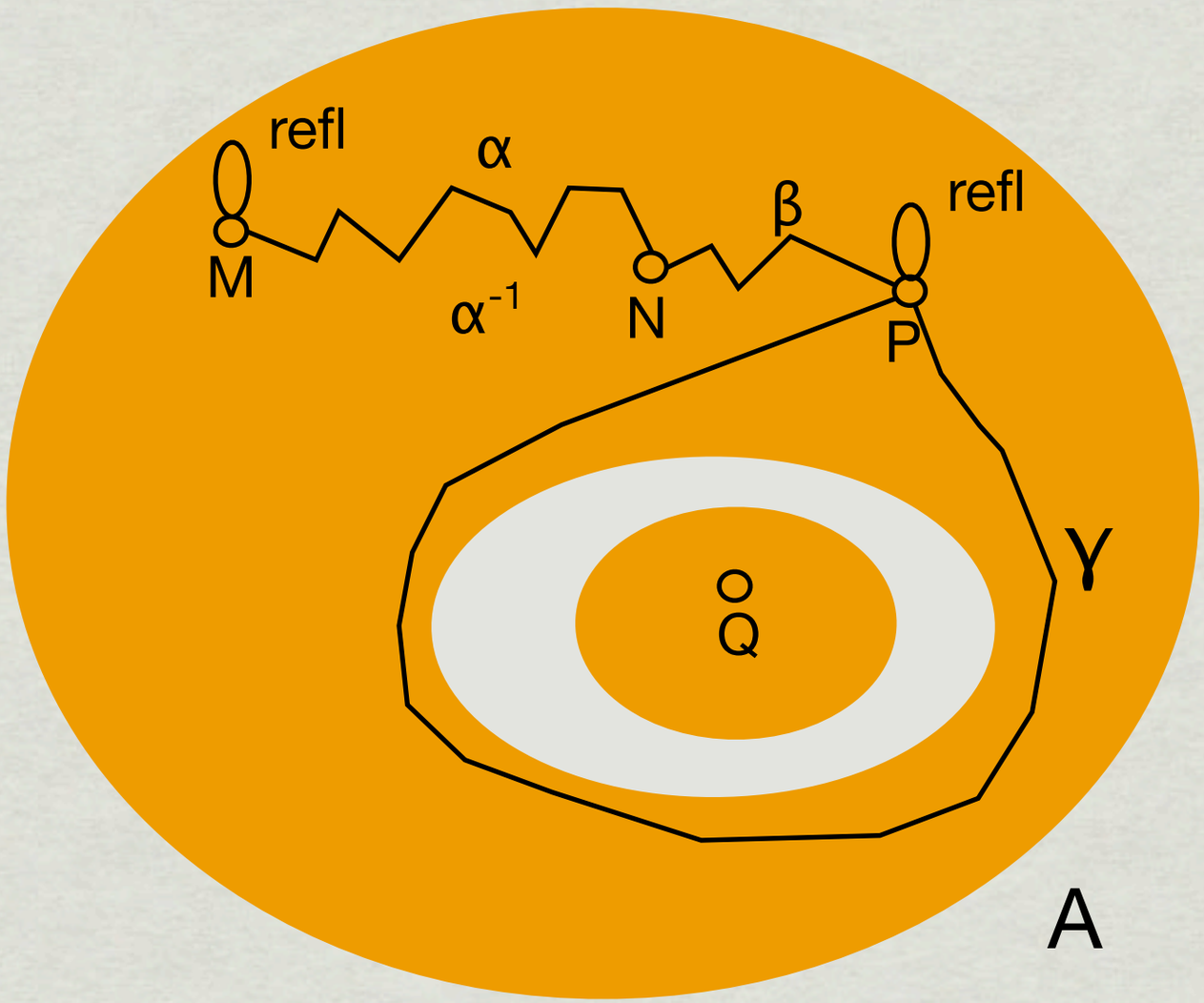


refl : $Id_A(M,M)$

$\alpha^{-1} : Id_A(N,M)$

$\beta \circ \alpha : Id_A(M,P)$

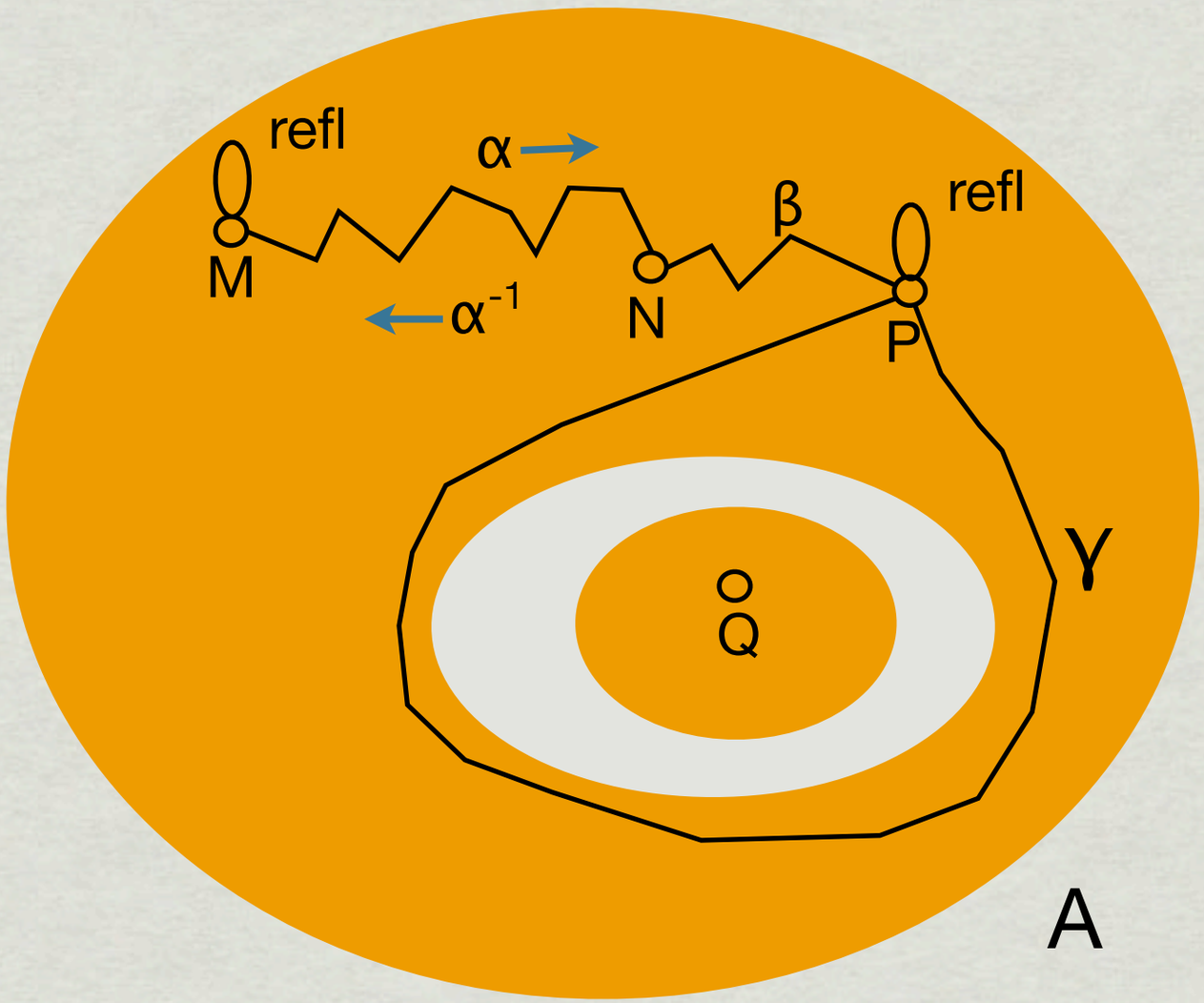
Equations on Paths



- associativity of o
- inverses cancel

γ != refl

Equations on Paths



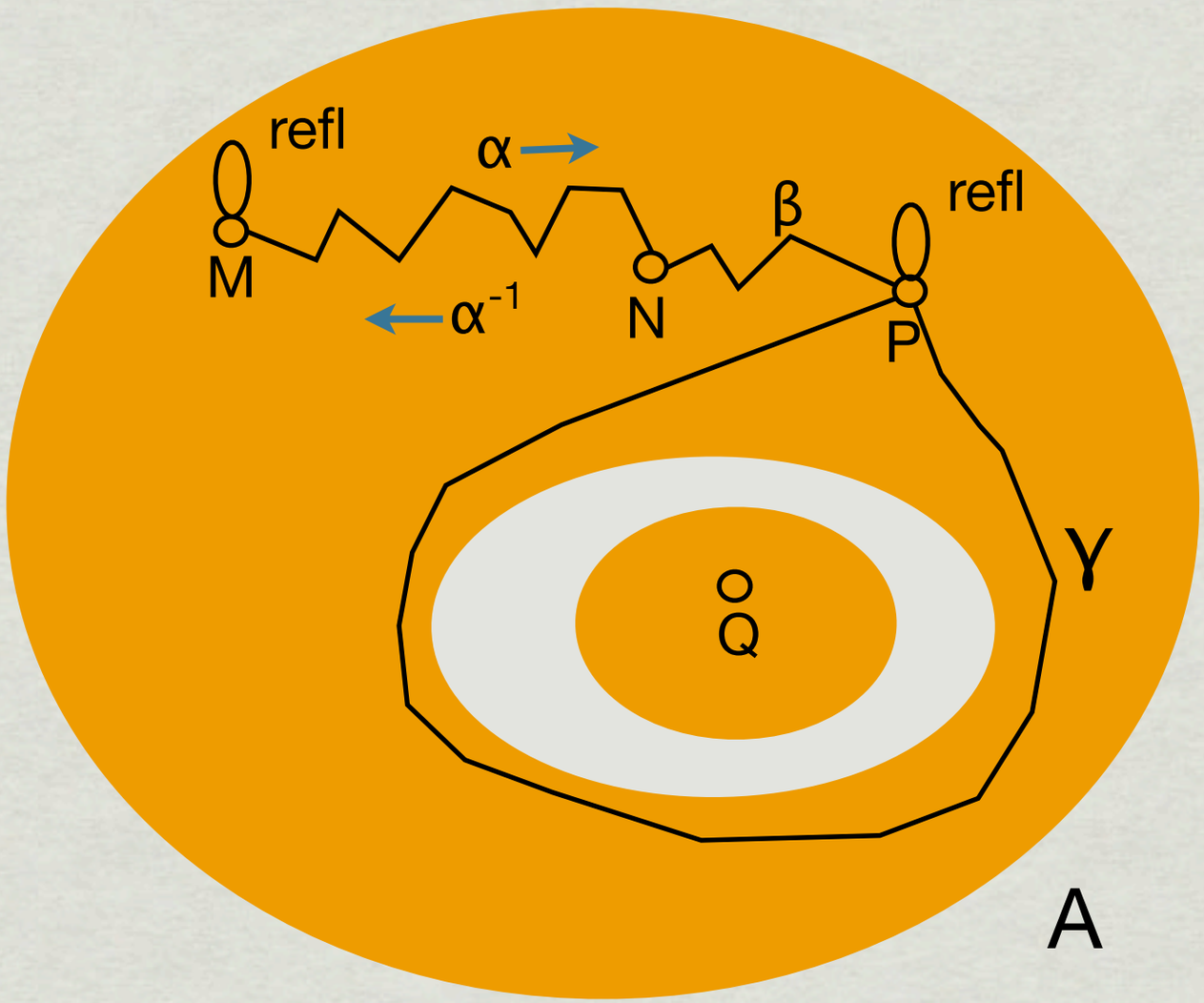
Yes:

- associativity of o
- inverses cancel

No:

γ != refl

Equations on Paths



Yes:

- associativity of o
- inverses cancel

No:

γ != refl

can have non-reflexivitiy paths!

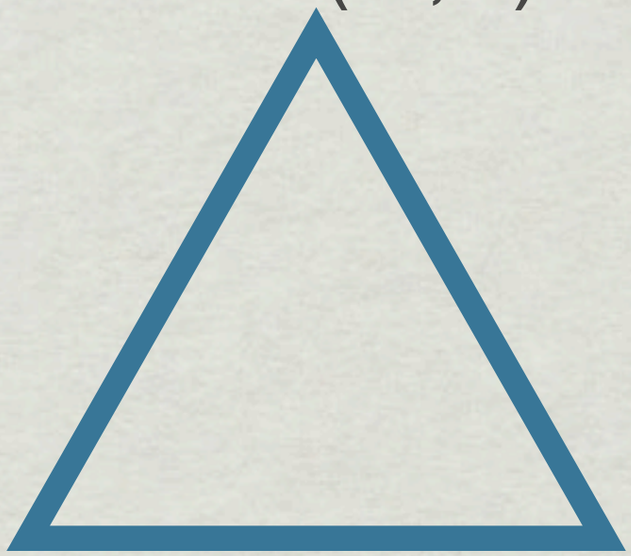
Homotopy Type Theory

dependent type theory

A : type

M, N : A

$\alpha : Id_A(M,N)$



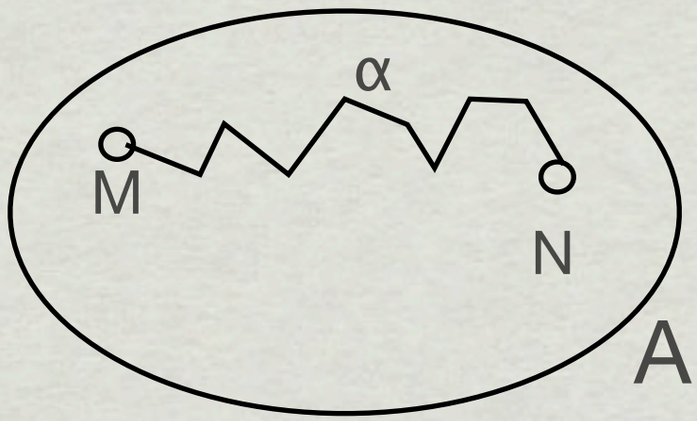
category theory

A is a groupoid

M,N objects

$\alpha : M \to N$ in A

homotopy theory



So what?

Mechanized Math

Type theory for math, cat. theory, homotopy theory

- Work “up to isomorphism” (and higher equivalences)
- Logical formalization of homotopy theory
  (e.g. fundamental groups of spheres)
- Quotient types in intensional type theory
- “Internal language” of higher categories:
  use syntax to reason about specific denotational semantics

A Simpler Proof that $\pi_1(S^1)$ is Z

Posted on 7 June 2012 by Dan Licata

Last year, Mike Shulman proved that $\pi_1(S^1)$ is Z in Homotopy Type Theory. While trying to understand Mike's proof, I came up with a simplification that shortens the proof a bunch (100 lines of Agda as compared to 380 lines of Coq).

My proof uses essentially the same ideas—e.g., the universal cover comes up in the same way, and the functions witnessing the isomorphism are the same. The main new idea is a simpler way of proving the hard direction of "and the composites are the identity": that if you start with a path on the circle, encode it as its winding number, and then decode it back as a path, then you get the original path. If you already understand Mike's proof, the diff is essentially this: you can replace steps 2-5 at the end by

1. Define an "encoding" function of type
   `forall (x : circle), (base ~~> x) -> circle_cover x`
   as the transport along `circle_cover` of 0.
2. Use J to prove that decoding (Mike's step 1) after encoding is the identity (this is the hard composite to work with, because it starts and ends with a path). The trick here is to ask the question for paths `(base ~~> x)` for an arbitrary `x`, not just for loops `base ~~> base`, so that you can use J.

This is clearly similar to Mike's proof (we could even ask if these two proofs are homotopic!): his proof uses contractibility of a path space, which is exactly J. But the amount of machinery about total spaces that you need here is much less.

A question, though: I came up with this proof basically on type-theoretic grounds ("there must be a simpler proof term of this type"). Is there anything to say about it in geometric or higher-categorical terms? In the semantics, is it the same proof as Mike's? A known

- Isomorphism implies equality
- Positive h-levels are closed under W
- Truncations and truncated higher inductive types
- A master thesis on homotopy type theory
- Running Spheres in Agda, Part

Recent Comments

- Peter Divianszky on Truncations and truncated higher inductive types
- Guillaume Brunerie on Truncations and truncated higher inductive types
- Mike Shulman on Truncations, truncated higher inductive types
- Guillaume Brunerie on Truncations and truncated higher inductive types
- Peter Divianszky on Truncations and truncated higher inductive types

Categories

- Applications
- Code
- Foundations
- Higher Inductive Types
- Models
- Paper
- Programming
- Talk
- Uncategorized
- Univalence

Archives

- September 2012

Programming

Cat. theory & homotopy theory for type theory

- Work “up to isomorphism”
  1. code reuse
  2. specs for abstract types
- Functorial abstract syntax [LH, MFPS 2011]

Programming

Cat. theory & homotopy theory for type theory

- Work “up to isomorphism”
  1. code reuse
  2. specs for abstract types
- Functorial abstract syntax [LH, MFPS 2011]

There is a *generic program* hidden inside of dependent type theory

Programming

Cat. theory & homotopy theory for type theory

- Work “up to isomorphism”

  1.code reuse

  2.specs for abstract types

- Functorial abstract syntax [LH, MFPS 2011]

There is a *generic program* hidden inside of dependent type theory

“up to isomorphism”

$$\frac{f : A \to B \qquad \alpha : Id_{A \to A}(g \,.\, f, \lambda x.x) \qquad g : B \to A \qquad \beta : Id_{B \to B}(f \,.\, g, \lambda y.y)}{(f,g,\alpha,\beta) : Iso(A,B)}$$

- Lots of isomorphic types, e.g. from refinements: List A and Σn:nat. Vec A n
- Everything in type theory respects isomorphism
- But you have to prove this by hand for each construction

“up to isomorphism”

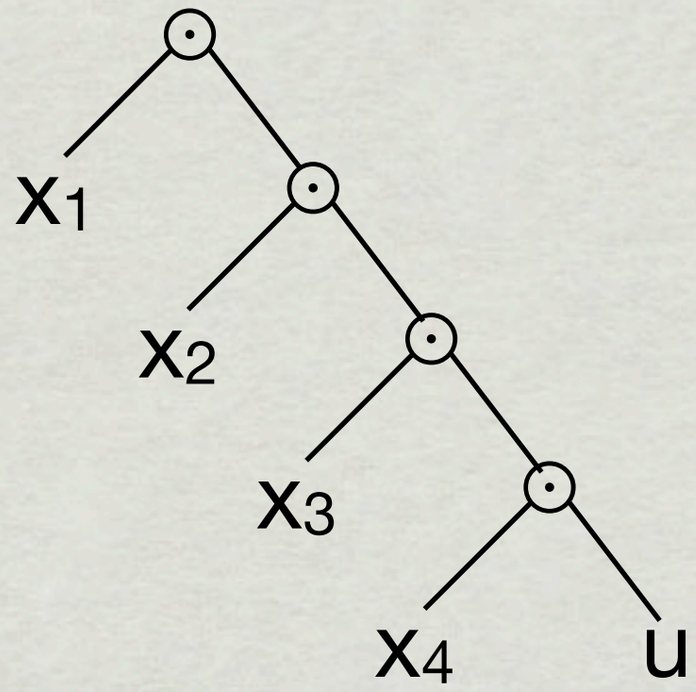
$$\frac{f : A \to B \qquad \alpha : Id_{A \to A}(g \,.\, f, \lambda x.x) \qquad g : B \to A \qquad \beta : Id_{B \to B}(f \,.\, g, \lambda y.y)}{(f,g,\alpha,\beta) : Iso(A,B)}$$

- Lots of isomorphic types, e.g. from refinements: List A and Σn:nat. Vec A n
- **Everything in type theory respects isomorphism**
- **But you have to prove this by hand for each construction**

Parallel Programming

reduce : (A → A → A) → A → A sequence → A

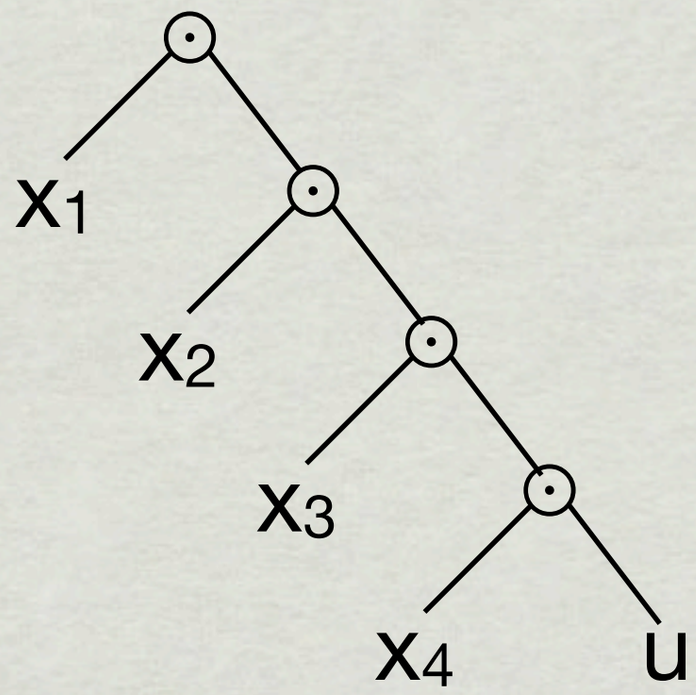
reduce ⊙ u [x_1,x_2,x_3,x_4] =



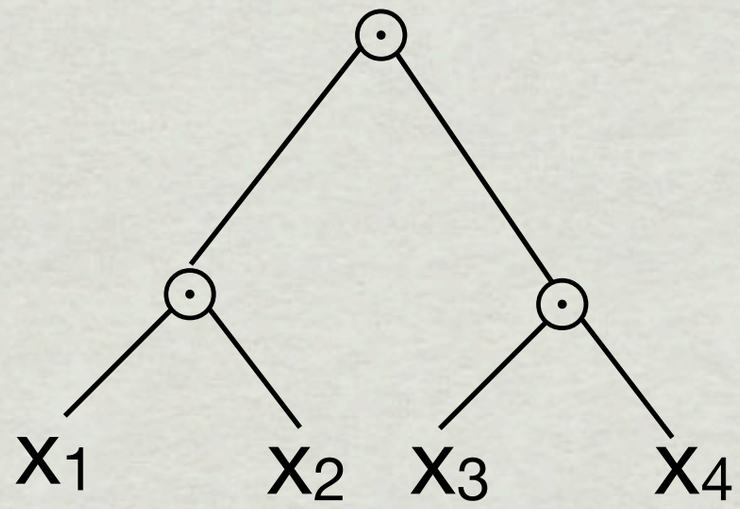
Parallel Programming

reduce : (A → A → A) → A → A sequence → A

reduce ⊙ u [x_1,x_2,x_3,x_4] =



if ⊙ is associative with unit u, can evaluate in parallel with same result:

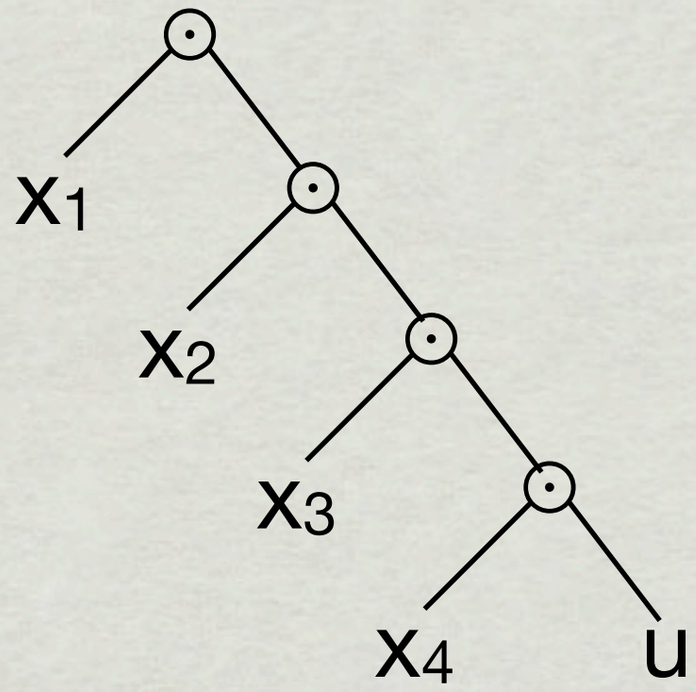


Parallel Programming

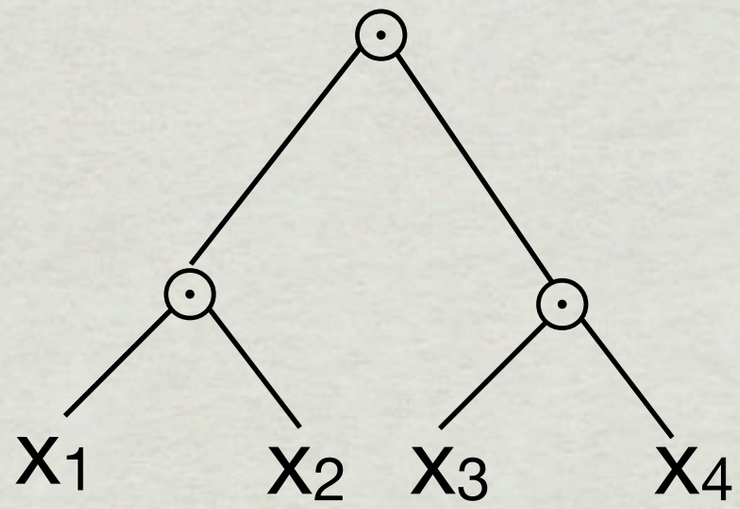
monoid

reduce : (A → A → A) → A → A sequence → A

reduce ⊙ u [x_1,x_2,x_3,x_4] =



if ⊙ is associative with unit u, can evaluate in parallel with same result:



Monoid

Monoid : type → type
Monoid X = Σ ⊙ : X → X → X.
Σ u : X.
(Πx,y,z. Id (x ⊙ (y ⊙ z)) ((x ⊙ y) ⊙ z))
* (Πx.Id (x ⊙ u) x)
* (Πx.Id (u ⊙ x) x)

Example: (append, [], …) : Monoid (List A)

Monoid respects isomorphism:
e.g. because List A is isomorphic to Σn.Vec A n, can make Monoid(Σn.Vec A n) following general principles

Monoid : type → type
Monoid X = Σ ⊙:X→X→X. Σ u:X. ...

Given (f,f^{-1},α,β) : Iso(A, B)
and (⊙,u,…) : Monoid(A)
make (⊙',u',…) : Monoid(B)

f : A → B
f^{-1} : B → A
α : $Id_{A→A}(f^{-1} \circ f, id)$
β : $Id_{B→B}(f \circ f^{-1}, id)$

Define:

Monoid : type → type
Monoid X = Σ ⊙:X→X→X. Σ u:X. ...

Given (f,f^{-1},α,β) : Iso(A, B)
and (⊙,u,…) : Monoid(A)
make (⊙',u',…) : Monoid(B)

f : A → B
f^{-1} : B → A
α : $Id_{A→A}(f^{-1} \circ f, id)$
β : $Id_{B→B}(f \circ f^{-1}, id)$

Define:

⊙' = λ y_1,y_2:B . f ((f^{-1} y_1) ⊙ (f^{-1} y_2))
u' = f u

Monoid : type → type
Monoid X = Σ ⊙:X→X→X. Σ u:X. ...

Given (f,f^{-1},α,β) : Iso(A, B)
and (⊙,u,...) : Monoid(A)
make (⊙',u',...) : Monoid(B)

f : A → B
f^{-1} : B → A
α : $Id_{A→A}(f^{-1} \circ f, id)$
β : $Id_{B→B}(f \circ f^{-1}, id)$

Define:

$\odot' = \lambda\, y_1, y_2{:}B\ .\ f\ ((f^{-1}\ y_1) \odot (f^{-1}\ y_2))$

$u' = f\ u$

$(y \odot' u') \simeq f\ ((f^{-1}\ y) \odot (f^{-1}\ (f\ u)))$

$\simeq f\ (f^{-1}\ y \odot u)$ by α

$\simeq f\ (f^{-1}\ y)$ by unit law for ⊙ and u

$\simeq y$ by β

Monoid : type → type
Monoid X = Σ ⊙:X→X→X. Σ u:X. ...

Given (f,f^{-1},α,β) : Iso(A, B)
and (⊙,u,...) : Monoid(A)
make (⊙',u',...) : Monoid(B)

f : A → B
f^{-1} : B → A
α : $Id_{A→A}(f^{-1} \circ f, id)$
β : $Id_{B→B}(f \circ f^{-1}, id)$

Define:

$\odot' = \lambda\, y_1, y_2 : B \,.\, f\,((f^{-1}\, y_1) \odot (f^{-1}\, y_2))$

$u' = f\, u$

"the hard way"

$(y \odot' u') \simeq f\,((f^{-1}\, y) \odot (f^{-1}\,(f\, u)))$

$\simeq f\,(f^{-1}\, y \odot u)$ by α

$\simeq f\,(f^{-1}\, y)$ by unit law for ⊙ and u

$\simeq y$ by β

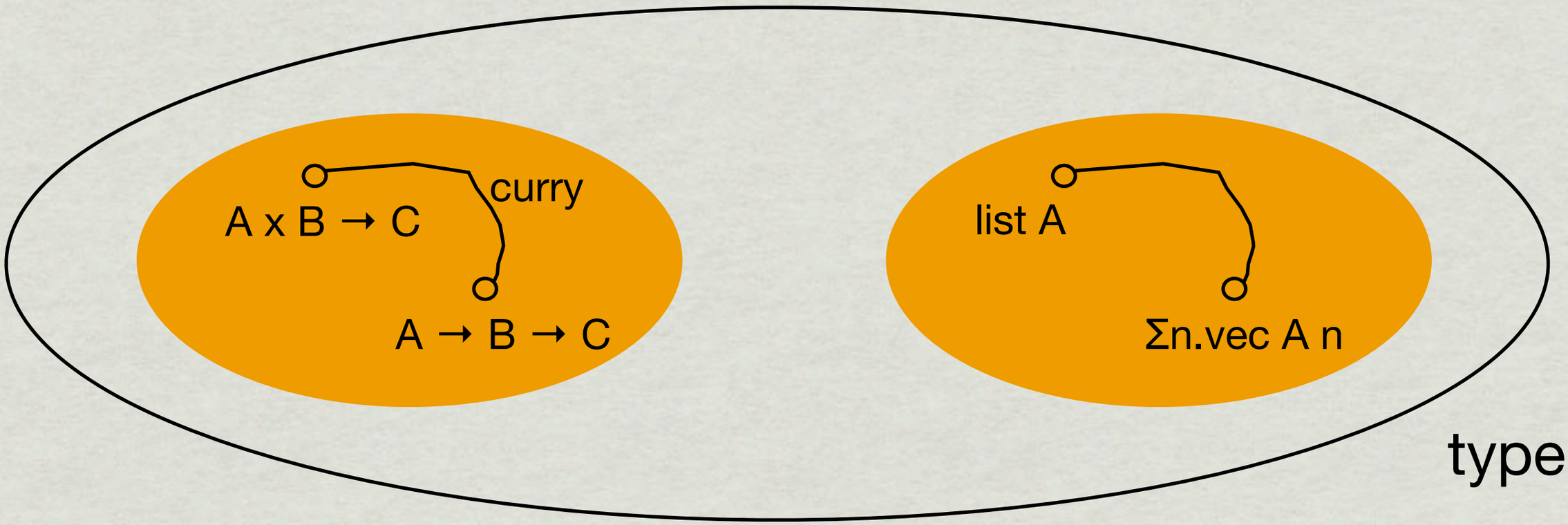
“up to isomorphism”

$$\frac{f : A \to B \qquad \alpha : Id_{A \to A}(g \circ f, id) \\ g : B \to A \qquad \beta : Id_{B \to B}(f \circ g, id)}{(f,g,\alpha,\beta) : Iso(A,B)}$$

- Lots of isomorphic types, e.g. from refinements: List A and Σn:nat.Vec A n
- Every family **(e.g. Monoid)** respects isomorphism
- But you have to prove this by hand for each construction **(syntax is neither here nor there)**

Univalence Axiom

univalence : Iso(A,B) → Id_{type}(A,B)

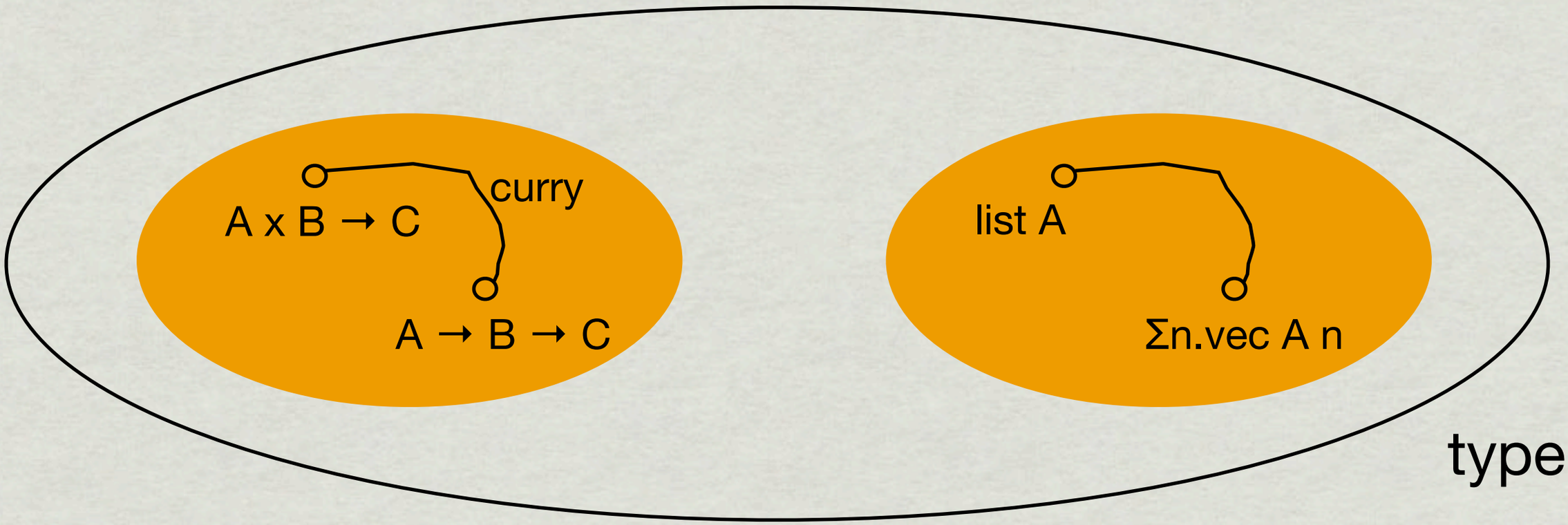


Computationally relevant, non-reflexivity paths

[Voevodsky]

Univalence Axiom

univalence : Iso(A,B) → Id_{type}(A,B)



Computationally relevant, non-reflexivity paths

The Easy Way

From $(f, f^{-1}, \alpha, \beta)$: Iso(A, B)

Make Monoid(A) → Monoid(B)

The Easy Way

From $(f, f^{-1}, \alpha, \beta) : \mathrm{Iso}(A, B)$

Make $\mathrm{Monoid}(A) \to \mathrm{Monoid}(B)$

$$\frac{C : A \to \mathrm{type} \quad \alpha : \mathrm{Id}_A(M,N) \quad P : C[M]}{\mathrm{subst}_C\ \alpha\ P : C[N]}$$

The Easy Way

From $(f, f^{-1}, \alpha, \beta) : \mathrm{Iso}(A, B)$

Make $\mathrm{Monoid}(A) \to \mathrm{Monoid}(B)$

$$\frac{C : A \to \mathrm{type} \quad \alpha : \mathrm{Id}_A(M,N) \quad P : C[M]}{\mathrm{subst}_C\ \alpha\ P : C[N]}$$

$$\mathrm{univalence} : \mathrm{Iso}(A,B) \to \mathrm{Id}_{\mathrm{type}}(A,B)$$

The Easy Way

From (f,f^{-1},α,β) : Iso(A, B)

Make Monoid(A) → Monoid(B)

$subst_{Monoid}$(univalence (f,f^{-1},α,β))

$$\frac{C : A \to type \qquad \alpha : Id_A(M,N) \qquad P : C[M]}{subst_C\ \alpha\ P : C[N]}$$

univalence : Iso(A,B) → Id_{type}(A,B)

The Easy Way

From $(f, f^{-1}, \alpha, \beta) : \mathrm{Iso}(A, B)$

Make $\mathrm{Monoid}(A) \to \mathrm{Monoid}(B)$

$$\mathbf{subst}_{\mathbf{Monoid}}(\mathbf{univalence}\ (f, f^{-1}, \alpha, \beta))$$

↑

type-generic lifting of isos

$$\frac{C : A \to \mathrm{type} \quad \alpha : \mathrm{Id}_A(M,N) \quad P : C[M]}{\mathrm{subst}_C\ \alpha\ P : C[N]}$$

$$\mathrm{univalence} : \mathrm{Iso}(A,B) \to \mathrm{Id}_{\mathrm{type}}(A,B)$$

Computation?

Standard computation rule for subst is:

$$\mathrm{subst}_C\ \mathrm{refl}\ P \equiv P$$

Therefore

$$\mathrm{subst}_{\mathrm{Monoid}}(\mathrm{univalence}\ (f, f^{-1}, \alpha, \beta))$$

is well-typed but *stuck: violates progress*

Computation?

Standard computation rule for subst is:

$$\mathrm{subst}_C\ \mathrm{refl}\ P \equiv P$$

Therefore

$$\mathrm{subst}_{\mathrm{Monoid}}(\mathrm{univalence}\ (f, f^{-1}, \alpha, \beta))$$

is well-typed but *stuck: violates progress*

Solution: $\mathrm{subst}_C\ \alpha\ P$ computes in a type-directed way, guided by the family C

subst for x.A(x) → B(x)

Want $\text{subst}_{x:A0.A(x) \to B(x)}\ \alpha\ F$:

$$A[N] \quad \to \quad B[N]$$

$$A[M] \quad \to \quad B[M]$$

F

subst for x.A(x) → B(x)

x:A0 ⊢ A(x) → B(x) type

$Id_{A0}(M,N)$

Want $subst_{x:A0.A(x) \to B(x)}$ α F :

A[N] → B[N]

A[M] → B[M]

F

subst for x.A(x) → B(x)

x:A0 ⊢ A(x) → B(x) type

$Id_{A0}(M,N)$

Want $subst_{x:A0.A(x) \to B(x)}$ α F :

A[N] → B[N]

$subst_A\ \alpha^{-1}$

A[M] → B[M]

F

subst for x.A(x) → B(x)

x:A0 ⊢ A(x) → B(x) type

$Id_{A0}(M,N)$

Want $subst_{x:A0.A(x) \to B(x)}$ α F :

$$
\begin{array}{ccc}
A[N] & \to & B[N] \\
\downarrow & & \uparrow \\
A[M] & \to & B[M]
\end{array}
$$

$subst_A\ \alpha^{-1}$

$subst_B\ \alpha$

F

subst for x.A(x) → B(x)

x:A0 ⊢ A(x) → B(x) type

$Id_{A0}(M,N)$

Want $subst_{x:A0.A(x) \to B(x)}\ \alpha\ F$:

$$
\begin{array}{ccc}
A[N] & \to & B[N] \\
\downarrow & & \uparrow \\
A[M] & \to & B[M]
\end{array}
$$

$subst_A\ \alpha^{-1}$ — F — $subst_B\ \alpha$

$subst_{x.A(x) \to B(x)}\ \alpha\ F$

$\equiv subst_{x.B(x)}\ \alpha\ .\ F\ .\ subst_{x.A(x)}\ \alpha^{-1}$

subst for X:type.X

Want $\mathrm{subst}_{\mathbf{X:type.X}}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

subst for X:type.X

Want $\mathrm{subst}_{X:\mathrm{type}.X}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

the only paths in the universe are constructed by univalence

subst for X:type.X

Want $\mathrm{subst}_{X:\mathrm{type}.X}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

the only paths in the universe are constructed by univalence

$\mathrm{subst}_{x.x}\ \alpha \equiv f$ when $\alpha \equiv \mathrm{univalence}(f{:}A \to B, g, \alpha, \beta)$

subst for X:type.X

Want $\mathrm{subst}_{X:\mathrm{type}.X}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

the only paths in the universe are constructed by univalence

$\mathrm{subst}_{x.x}\ \alpha \equiv f$ when $\alpha \equiv \mathrm{univalence}(f{:}A \to B, g, \alpha, \beta)$

β-reduction for univalence: deploy the isomorphism

Monoid : type → type
Monoid X = Σ ⊙:X→X→X. Σ u:X. …

Given (f,f^{-1},α,β) : Iso(A, B)
and (⊙,u,…) : Monoid(A)
make (⊙',u',…) : Monoid(B)

f : A → B
f^{-1} : B → A
α : $Id_{A→A}(f^{-1} \circ f, id)$
β : $Id_{B→B}(f \circ f^{-1}, id)$

Define:

$\odot' = \lambda\, y_1,y_2{:}B\ .\ f\ ((f^{-1}\ y_1) \odot (f^{-1}\ y_2))$

$u' = f\ u$

"the hard way"

$(y \odot' u') \simeq f\ ((f^{-1}\ y) \odot (f^{-1}\ (f\ u)))$

$\simeq f\ (f^{-1}\ y \odot u)$ by α

$\simeq f\ (f^{-1}\ y)$ by unit law for ⊙ and u

$\simeq y$ by β

Easy ⟼ Hard

α : Id A B =
univalence (f,f^{-1},…)

Monoid(A)

↓

: Monoid(B)

“the easy way”

“the hard way”

Easy ⟼ Hard

α : Id A B =
univalence (f,f^{-1},…)

Monoid(A)

↓

$\text{subst}_{\text{X:type.Monoid(X)}}$ α (⊙,u,…): **Monoid(B)**

“the easy way”

≡ (λy_1,y_2. f ((f^{-1} y_1) ⊙ (f^{-1} y_2))
f u, …)

“the hard way”

Easy ⟼ Hard

α : Id A B =
univalence (f,f^{-1},…)

Monoid(A)

subst**X:type.Monoid(X)** α (⊙,u,…)**: Monoid(B)**

“the easy way”

≡ subst**X:type.Σ⊙:X→X→X.Σu:X….** α (⊙,u,…)

≡ (λy_1,y_2. f ((f^{-1} y_1) ⊙ (f^{-1} y_2))
f u, …)

“the hard way”

Easy ⟼ Hard

α : Id A B =
univalence (f,f^{-1},…)

Monoid(A)

subst**X:type.Monoid(X)** α (⊙,u,…)**: Monoid(B)**

"the easy way"

≡ subst**X:type.Σ⊙:X→X→X.Σu:X….** α (⊙,u,…)

≡ (subst**x.x→x→x** α ⊙,
subst**x.x** α u, …)

≡ (λy_1,y_2. f ((f^{-1} y_1) ⊙ (f^{-1} y_2))
f u, …)

"the hard way"

Easy ⟼ Hard

α : Id A B =
univalence (f,f^{-1},…)

Monoid(A)

subst**X:type.Monoid(X)** α (⊙,u,…)**: Monoid(B)**

"the easy way"

≡ subst**X:type.Σ⊙:X→X→X.Σu:X….** α (⊙,u,…)

≡ (subst**x.x→x→x** α ⊙,
subst**x.x** α u, …)

≡ ($\lambda y_1,y_2$. subst**x.x** α ((subst**x.x** α^{-1} y_1) ⊙ (subst**x.x** α^{-1} y_2))
subst**x.x** α u, …)

≡ ($\lambda y_1,y_2$. f ((f^{-1} y_1) ⊙ (f^{-1} y_2))
f u, …)

"the hard way"

Programming

Cat. theory & homotopy theory useful for type theory

- Work “up to isomorphism”
  1. code reuse
  2. **specs for abstract types**
- Functorial abstract syntax [LH, MFPS 2011]

There is a *generic program* hidden inside of dependent type theory

Specs for Abstract Types

```
signature SEQ =
sig  type α seq
     val map    : (α → β) → (α seq → β seq)
     val reduce : ...
end
```

Specs for Abstract Types

```
signature SEQ =
sig  type α seq
     val map    : (α → β) → (α seq → β seq)
     val reduce : ...
end

structure PSeq :> SEQ = <parallel sequences>
```

Specs for Abstract Types

```
signature SEQ =
sig  type α seq
     val map    : (α → β) → (α seq → β seq)
     val reduce : ...
end
```

structure PSeq :> SEQ = <parallel sequences>

Behavioral spec: PSeq.map f $\langle x_1, \ldots, x_n \rangle = \langle f\ x_1, \ldots, f\ x_n \rangle$
Operationally: evaluated in parallel

Specs for Abstract Types

key abstraction for 1st&2nd-year FP/parallel algorithms classes at CMU

```
signature SEQ =
sig  type α seq
     val map    : (α → β) → (α seq → β seq)
     val reduce : ...
end
```

structure PSeq :> SEQ = <parallel sequences>

Behavioral spec: PSeq.map f $\langle x_1, \ldots, x_n \rangle = \langle f\ x_1, \ldots, f\ x_n \rangle$
Operationally: evaluated in parallel

Specs for Abstract Types

Behavioral: PSeq.map f $\langle x_1, \ldots, x_n \rangle = \langle f\ x_1, \ldots, f\ x_n \rangle$

How to make this precise?

Specs for Abstract Types

Behavioral: PSeq.map f $\langle x_1, \ldots, x_n \rangle = \langle f\ x_1, \ldots, f\ x_n \rangle$

How to make this precise?

```
structure ListSeq : SEQ =
struct
    type α seq = α list
    val map = List.map …
end
```

Specs for Abstract Types

Behavioral: PSeq.map f $\langle x_1, \ldots, x_n \rangle = \langle f\ x_1, \ldots, f\ x_n \rangle$

How to make this precise?

```
structure ListSeq : SEQ =
struct
    type α seq = α list
    val map = List.map …
end
```

Spec: “PSeq behaves like ListSeq”

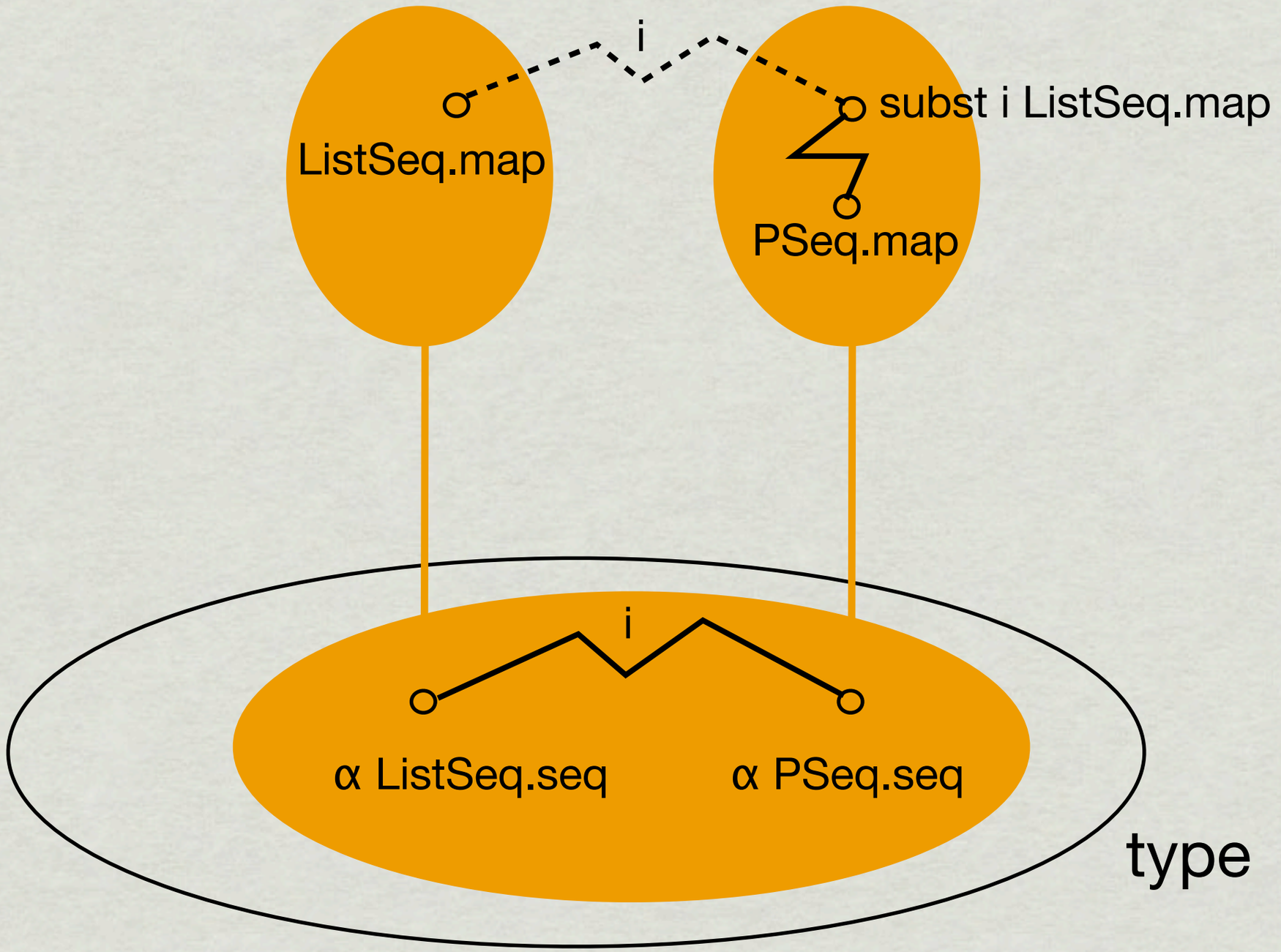
“PSeq behaves like ListSeq”

“PSeq behaves like ListSeq”

spec : $\mathrm{Id}_{\mathrm{SEQ}}$ ListSeq PSeq

“PSeq behaves like ListSeq”

spec : $\mathrm{Id}_{\mathrm{SEQ}}$ ListSeq PSeq



“PSeq behaves like ListSeq”

spec : $\mathrm{Id}_{\mathrm{SEQ}}$ ListSeq PSeq

spec implies

“PSeq behaves like ListSeq”

spec : $\mathrm{Id}_{\mathrm{SEQ}}$ ListSeq PSeq

spec implies

```
i : Iso(α ListSeq.seq, α PSeq.seq) =
  (fromList : α ListSeq.seq → α PSeq.seq,
   toList   : α PSeq.seq → α ListSeq.seq, …)
```

“PSeq behaves like ListSeq”

spec : $\mathrm{Id}_{\mathrm{SEQ}}$ ListSeq PSeq

spec implies

```
i : Iso(α ListSeq.seq, α PSeq.seq) =
  (fromList : α ListSeq.seq → α PSeq.seq,
   toList   : α PSeq.seq → α ListSeq.seq, …)
```

$$\mathrm{PSeq.map} \simeq \mathrm{subst}_{(s:\mathrm{type}\to\mathrm{type}.\ (\alpha\to\beta)\to\ \alpha\ s\ \to\ \beta\ s)}\ i\ \mathrm{ListSeq.map}$$
$$\simeq \lambda f.\ \mathrm{fromList}\ .\ \mathrm{List.map}\ f\ .\ \mathrm{toList}$$

“PSeq behaves like ListSeq”

spec : $\mathrm{Id}_{\mathrm{SEQ}}$ ListSeq PSeq

spec implies

```
i : Iso(α ListSeq.seq, α PSeq.seq) =
  (fromList : α ListSeq.seq → α PSeq.seq,
   toList   : α PSeq.seq → α ListSeq.seq, …)
```

$$\mathrm{PSeq.map} \simeq \mathrm{subst}_{(s:\mathrm{type}\to\mathrm{type}.\ (\alpha\to\beta)\to\ \alpha\ s\ \to\ \beta\ s)}\ i\ \mathrm{ListSeq.map}$$
$$\simeq \lambda f.\ \mathrm{fromList}\ .\ \mathrm{List.map}\ f\ .\ \mathrm{toList}$$

and so on for reduce, … easier than writing out by hand

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

PSeq.map (g . f)

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

PSeq.map (g . f)
≃ fromList . List.map (g.f) . toList

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

PSeq.map (g . f)
≃ fromList . List.map (g.f) . toList
≃ fromList . List.map g . List.map f . toList

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

PSeq.map (g . f)
≃ fromList . List.map (g.f) . toList
≃ fromList . List.map g . List.map f . toList

PSeq.map g . PSeq.map f

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

PSeq.map (g . f)
≃ fromList . List.map (g.f) . toList
≃ fromList . List.map g . List.map f . toList

PSeq.map g . PSeq.map f
≃ (fromList . List.map g . toList) .
(fromList . List.map f . toList)

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

PSeq.map (g . f)
≃ fromList . List.map (g.f) . toList
≃ fromList . List.map g . List.map f . toList

PSeq.map g . PSeq.map f
≃ (fromList . List.map g . toList) .
(fromList . List.map f . toList)
≃ fromList . List.map g . (toList . fromList) .
List.map f . toList

Can use this to reason about PSeq:

PSeq.map (g . f) ≃ PSeq.map g . PSeq.map f

PSeq.map (g . f)
≃ fromList . List.map (g.f) . toList
≃ fromList . List.map g . List.map f . toList

PSeq.map g . PSeq.map f
≃ (fromList . List.map g . toList) .
(fromList . List.map f . toList)
≃ fromList . List.map g . (toList . fromList) .
List.map f . toList
≃ fromList . List.map g .
List.map f . toList

Programming

Cat. theory & homotopy theory useful for type theory

- Work “up to isomorphism”
  1. code reuse
  2. specs for abstract types
- Functorial abstract syntax [LH, MFPS 2011]

There is a *generic program* hidden inside of dependent type theory

Status

- LH, POPL'12: computational interpretation for simplest generalization (2-dimensional -- no paths between paths); non-algorithmic (up to ≡)

- Conjectured algorithm for 2D: have an operational semantics; metatheory in progress

- Generalization to higher dimensions is a hard open problem

More Details on Operational Semantics

Type A * B

- Members: (M, N) where M : A and N : B
- Paths: $$\frac{\alpha : Id_A(M,N) \quad \beta : Id_B(M', N')}{pair{\simeq}\ \alpha\ \beta : Id_{A*B}(M,N)(M',N')}$$
- $Refl_{(M,N)} \equiv pair{\simeq}\ Refl_M\ Refl_N$
- $(pair{\simeq}\ \alpha\ \beta)^{-1} \equiv (pair{\simeq}\ \alpha^{-1}\ \beta^{-1})$
- $(pair{\simeq}\ \alpha\ \beta)\ o\ (pair{\simeq}\ \alpha'\ \beta') \equiv (pair{\simeq}\ \alpha o \alpha'\ \beta o \beta')$

Type type

- Members: <names for small sets>, which determine types
- Paths:

$$\frac{(f,g,\alpha,\beta) : \mathrm{Iso}\ A\ B}{\mathrm{ua}(f,g,\alpha,\beta) : \mathrm{Id}_{\mathrm{type}}(A,\ B)}$$

- $\mathrm{Refl}_A \equiv (x{:}A.x, x{:}A.x, x.\mathrm{Refl}_x, x.\mathrm{Refl}_x)$
- $(\mathrm{ua}(f,g,\alpha,\beta))^{-1} \equiv \ldots$
- $(\mathrm{ua}(f',g',\alpha',\beta')) \circ (\mathrm{ua}(f',g',\alpha',\beta')) \equiv \ldots$

Computational interpretation

1. Define each type by members and paths, with refl $^{-1}$ o satisfying groupoid laws

2. Define subst_C α M for each C

3. Define resp F α for each F

4. Define full Id-elim rule J using subst

(1,2,3 simultaneous)

Subst

$$\frac{C : A \to \text{type} \quad \alpha : \mathrm{Id}_A(M,N) \quad P : C[M]}{\mathrm{subst}_C\ \alpha\ P : C[N]}$$

To be a family of types over A, C must have an associated operation subst_C

Functionality (à la NuPRL) becomes functoriality

Case for x.A(x) * B(x)

Id(θ1,θ2)

Want $\text{subst}_{\text{x.A(x) x B(x)}}\ \alpha$:

$$A[\theta 1] * B[\theta 1] \quad \rightarrow \quad A[\theta 2] * B[\theta 2]$$

Have:

Case for x.A(x) * B(x)

Want $\text{subst}_{\text{x.A(x) x B(x)}}\ \alpha$:

Id(θ1,θ2)

$$A[\theta 1] * B[\theta 1] \quad \rightarrow \quad A[\theta 2] * B[\theta 2]$$

Have:

$$\text{subst}_{A}\ \alpha : A[\theta 1] \rightarrow A[\theta 2]$$

Case for x.A(x) * B(x)

Want $\text{subst}_{x.A(x)\ x\ B(x)}\ \alpha$:

Id(θ1,θ2)

$$A[\theta 1] * B[\theta 1] \quad \rightarrow \quad A[\theta 2] * B[\theta 2]$$

Have:

$$\text{subst}_A\ \alpha : A[\theta 1] \rightarrow A[\theta 2]$$

$$\text{subst}_B\ \alpha : B[\theta 1] \rightarrow B[\theta 2]$$

Case for x.A(x) * B(x)

Id(θ1,θ2)

Want $\text{subst}_{\text{x.A(x) x B(x)}}\ \alpha$:

$$A[\theta 1] * B[\theta 1] \quad \rightarrow \quad A[\theta 2] * B[\theta 2]$$

Have:

$$\text{subst}_{A}\ \alpha : A[\theta 1] \rightarrow A[\theta 2]$$

$$\text{subst}_{B}\ \alpha : B[\theta 1] \rightarrow B[\theta 2]$$

$$\text{subst}_{\text{x.A(x) x B(x)}}\ \alpha\ (M1,M2)$$
$$\equiv (\text{subst}_{\text{x.A(x)}}\ \alpha\ M1,\ \text{subst}_{\text{X:U.B(x)}}\ \alpha\ M2)$$

Case for x.A(x) → B(x)

Id(θ1,θ2)

Want $\text{subst}_{\text{x.A(x)} \to \text{B(x)}}\ \alpha\ F$:

$$A[\theta 2] \quad \to \quad B[\theta 2]$$

$$A[\theta 1] \quad \to \quad B[\theta 1]$$
$$F$$

$$\text{subst}_{\text{x.A(X)} \to \text{B(X)}}\ \alpha\ F$$
$$\equiv (\lambda x.\text{subst}_{\text{xB(X)}}\ \alpha\ (F\ (\text{subst}_{\text{X:U.A(X)}}\ \alpha^{-1}\ x))$$

Case for x.A(x) → B(x)

Id(θ1,θ2)

Want $\text{subst}_{\mathbf{x.A(x) \to B(x)}}\ \alpha\ F$:

$$A[\theta 2] \quad \to \quad B[\theta 2]$$

$\text{subst}_A\ \alpha^{-1}$ ↓ F

$$A[\theta 1] \quad \to \quad B[\theta 1]$$

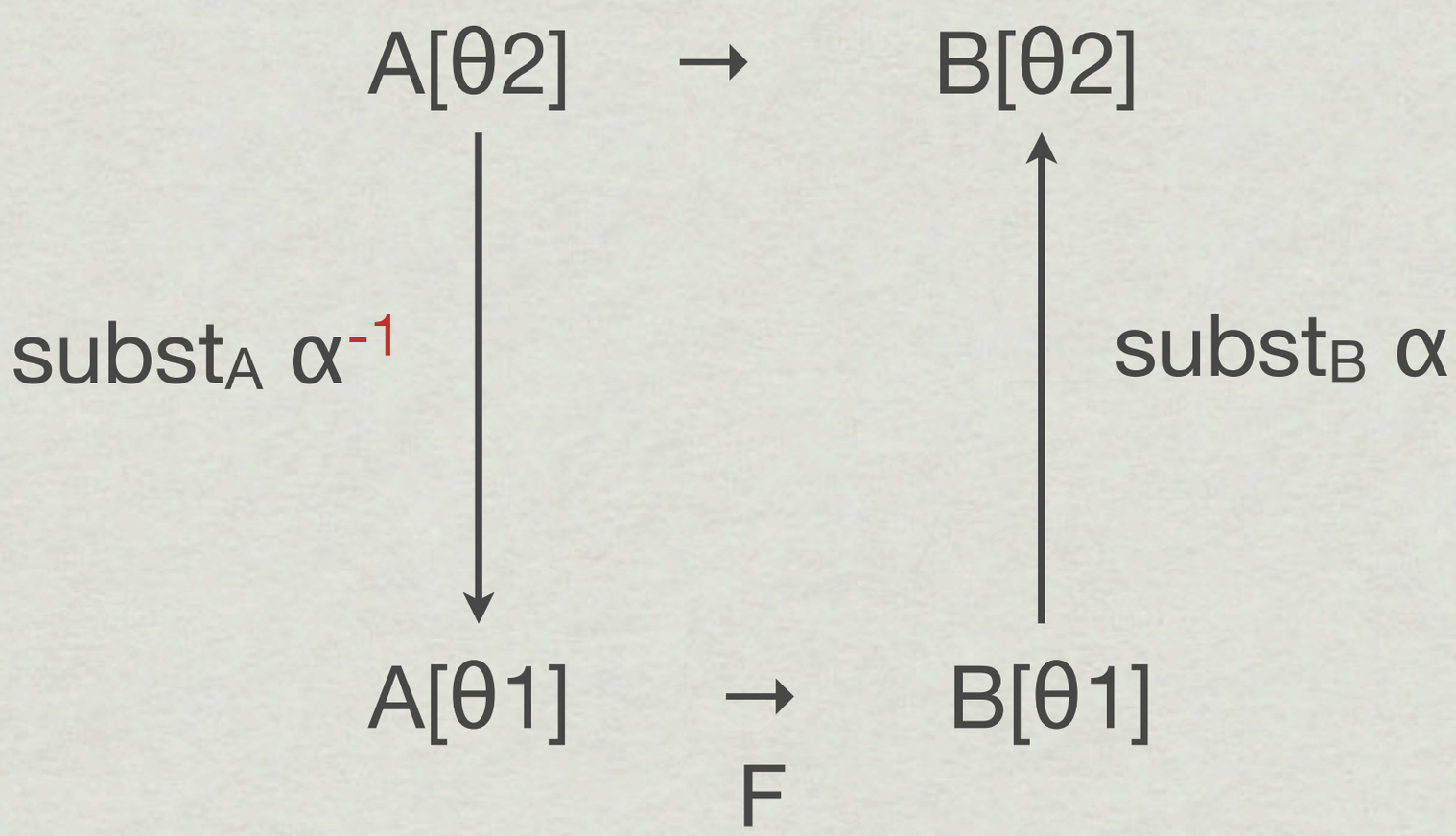
$$\text{subst}_{\mathbf{x.A(X) \to B(X)}}\ \alpha\ F$$

$$\equiv (\lambda x.\text{subst}_{\mathbf{xB(X)}}\ \alpha\ (F\ (\text{subst}_{\mathbf{X:U.A(X)}}\ \alpha^{-1}\ x))$$

Case for x.A(x) → B(x)

Id(θ1,θ2)

Want $\text{subst}_{\mathbf{x.A(x) \to B(x)}}\ \alpha\ F$:



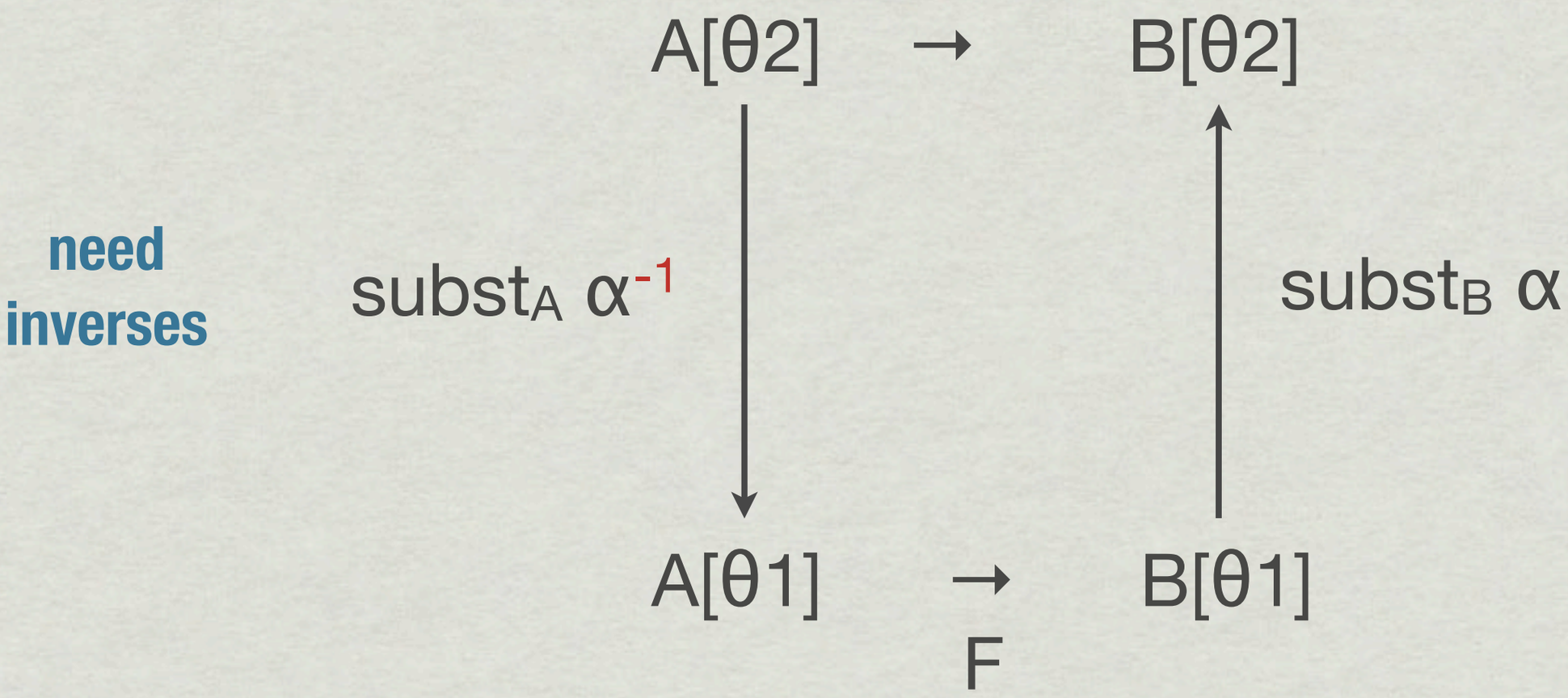
$\text{subst}_{\mathbf{x.A(X) \to B(X)}}\ \alpha\ F$

$\equiv (\lambda x.\text{subst}_{\mathbf{xB(X)}}\ \alpha\ (F\ (\text{subst}_{\mathbf{X:U.A(X)}}\ \alpha^{-1}\ x))$

Case for x.A(x) → B(x)

Id(θ1,θ2)

Want $\text{subst}_{\text{x.A(x)} \to \text{B(x)}}\ \alpha\ F$:



$\text{subst}_{\text{x.A(X)} \to \text{B(X)}}\ \alpha\ F$

$\equiv (\lambda x.\text{subst}_{\text{xB(X)}}\ \alpha\ (F\ (\text{subst}_{\text{X:U.A(X)}}\ \alpha^{-1}\ x))$

Case for X:type.X

Want $\mathrm{subst}_{\mathbf{X:type.X}}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

Case for X:type.X

Want $\mathrm{subst}_{X:\mathrm{type}.X}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

the only paths in the universe are constructed by univalence

Case for X:type.X

Want $\mathrm{subst}_{X:\mathrm{type}.X}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

the only paths in the universe are constructed by univalence

$\mathrm{subst}_{x.x}\ \alpha \equiv f$ when $\alpha \equiv \mathrm{univalence}(f{:}A \to B, g, \alpha, \beta)$

Case for X:type.X

Want $\mathrm{subst}_{X:\mathrm{type}.X}\ \alpha : A \to B$

Have $\alpha : \mathrm{Id}_{\mathrm{type}}(A, B)$

the only paths in the universe are constructed by univalence

$\mathrm{subst}_{x.x}\ \alpha \equiv f$ when $\alpha \equiv \mathrm{univalence}(f{:}A \to B, g, \alpha, \beta)$

β-reduction for univalence: deploy the isomorphism

Resp

$$\frac{F : A \to B \qquad \alpha : Id_A(M,N)}{resp\ F\ \alpha : Id_B(F\ M, F\ N)}$$

To be a family of terms over A, F must have an associated operation resp F α

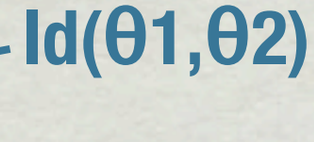
Functionality (à la NuPRL) becomes functoriality

Case for $x.Id_A(M(x),N(x))$

$$M[\theta 2] \simeq N[\theta 2]$$

$$M[\theta 1] \simeq N[\theta 1]$$
$$\beta$$

Case for $x.Id_A(M(x),N(x))$



Want $subst_{x.Id_A(M(x),N(x))}\ \alpha\ \beta$:

$$M[\theta 2] \simeq N[\theta 2]$$

$(resp\ M\ \alpha)^{-1}$ ⌇I

$$M[\theta 1] \simeq N[\theta 1]$$

β

Case for $x.\mathrm{Id}_A(M(x),N(x))$

Want $\mathrm{subst}_{x.\mathrm{Id}_A(M(x),N(x))}\ \alpha\ \beta$:

Id(θ1,θ2)

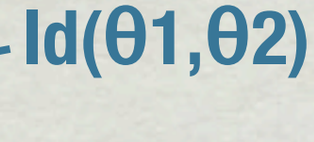
$$M[\theta 2] \simeq N[\theta 2]$$

$$(\mathrm{resp}\ M\ \alpha)^{-1} \wr\mid \qquad \mid\wr\ \mathrm{resp}\ N\ \alpha$$

$$M[\theta 1] \simeq N[\theta 1]$$

$$\beta$$

Case for $x.Id_A(M(x),N(x))$



Want $subst_{x.IdA(M(x),N(x))}\ \alpha\ \beta$:

$$M[\theta 2] \simeq N[\theta 2]$$

$$(resp\ M\ \alpha)^{-1} \wr\mid \qquad \mid\wr\ resp\ N\ \alpha$$

$$M[\theta 1] \simeq N[\theta 1]$$

$$\beta$$

$subst_{x.\ IdA(M(x),N(x))}\ \alpha\ \beta$

$\equiv (resp\ N\ \alpha)\ o\ \beta\ o\ (resp\ M\ \alpha)^{-1}$

Case for $x.Id_A(M(x),N(x))$

Want $subst_{x.Id_A(M(x),N(x))}\ \alpha\ \beta$:

Id(θ1,θ2)

$$M[\theta 2] \simeq N[\theta 2]$$

$$(resp\ M\ \alpha)^{-1} \wr\mid \qquad \mid\wr\ resp\ N\ \alpha$$

$$M[\theta 1] \simeq N[\theta 1]$$

$$\beta$$

need inverses, composition, resp

$subst_{x.\ Id_A(M(x),N(x))}\ \alpha\ \beta$

$\equiv (resp\ N\ \alpha)\ \circ\ \beta\ \circ\ (resp\ M\ \alpha)^{-1}$

$$\frac{F : A \to B \qquad \alpha : Id_A(M,N)}{resp\ F\ \alpha : Id_B(F\ M, F\ N)}$$