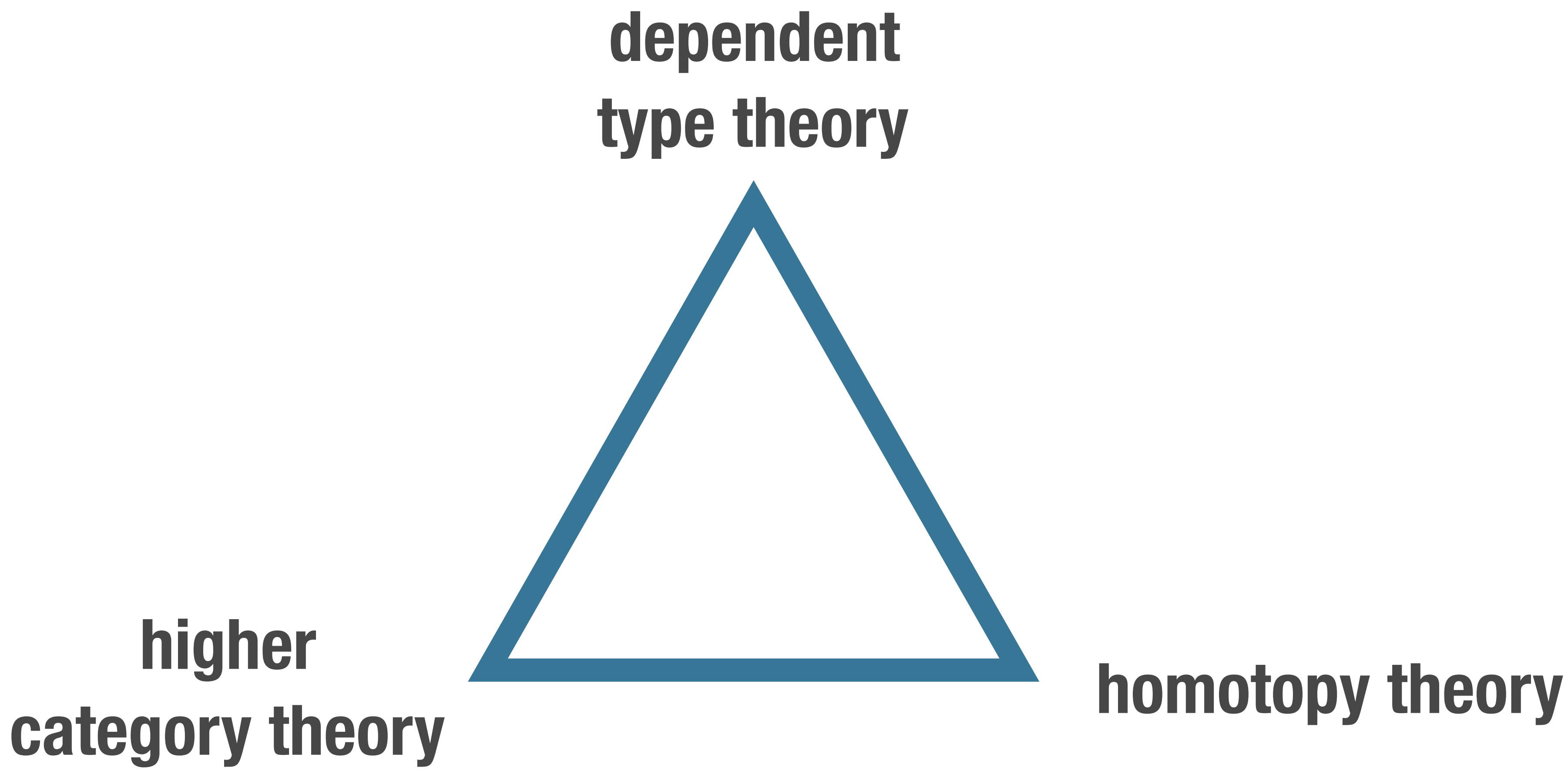
A Functional Programmer's Guide to Homotopy Type Theory

Dan Licata
Wesleyan University

Homotopy type theory



The Coq Proof Assistant

PRL Project "Proof/Program Refinement Logic"

Agda

Agda is a dependently typed functional programming language.

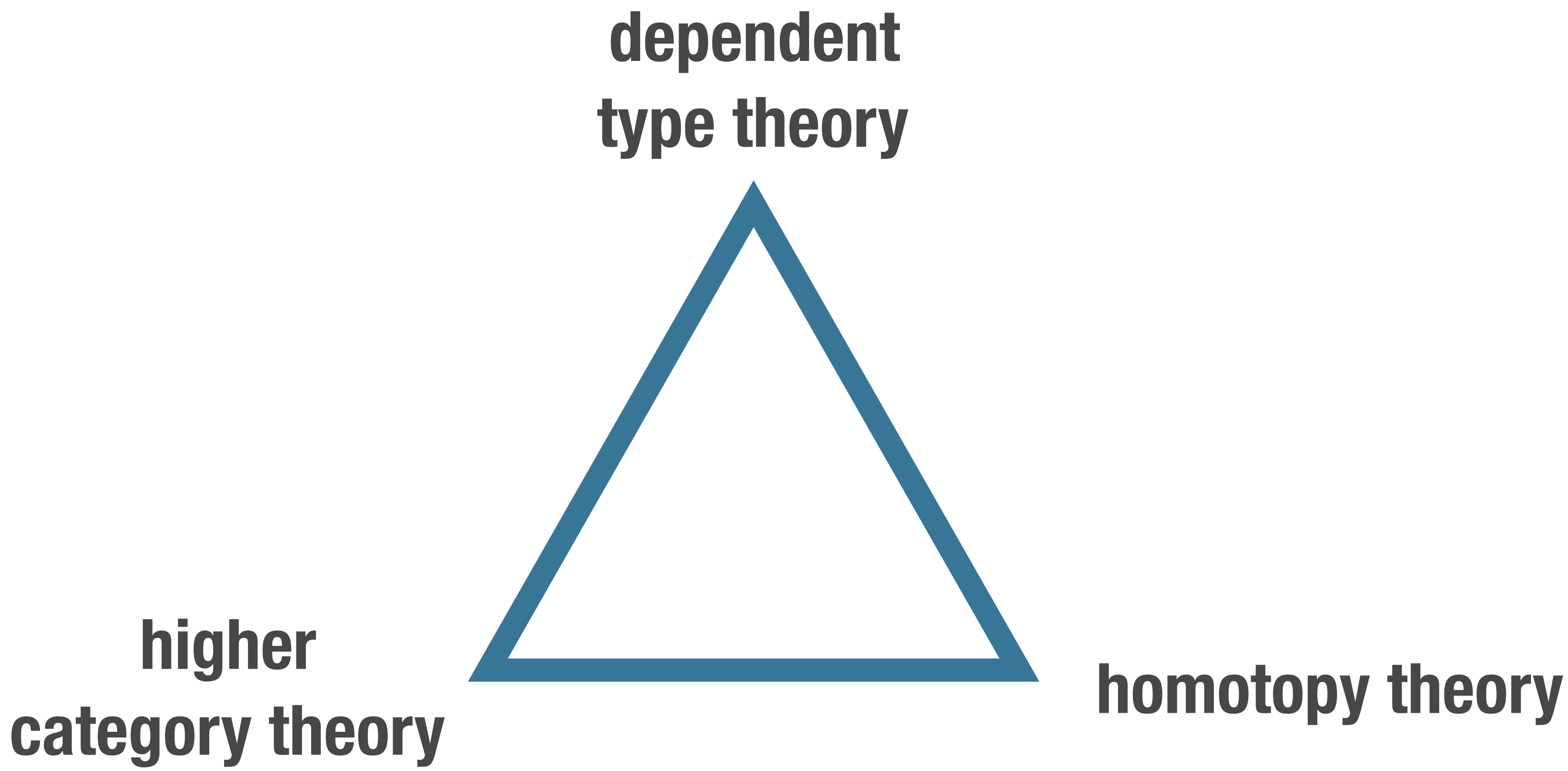
LEAN THEOREM PROVER

Idris

A Language with Dependent Types

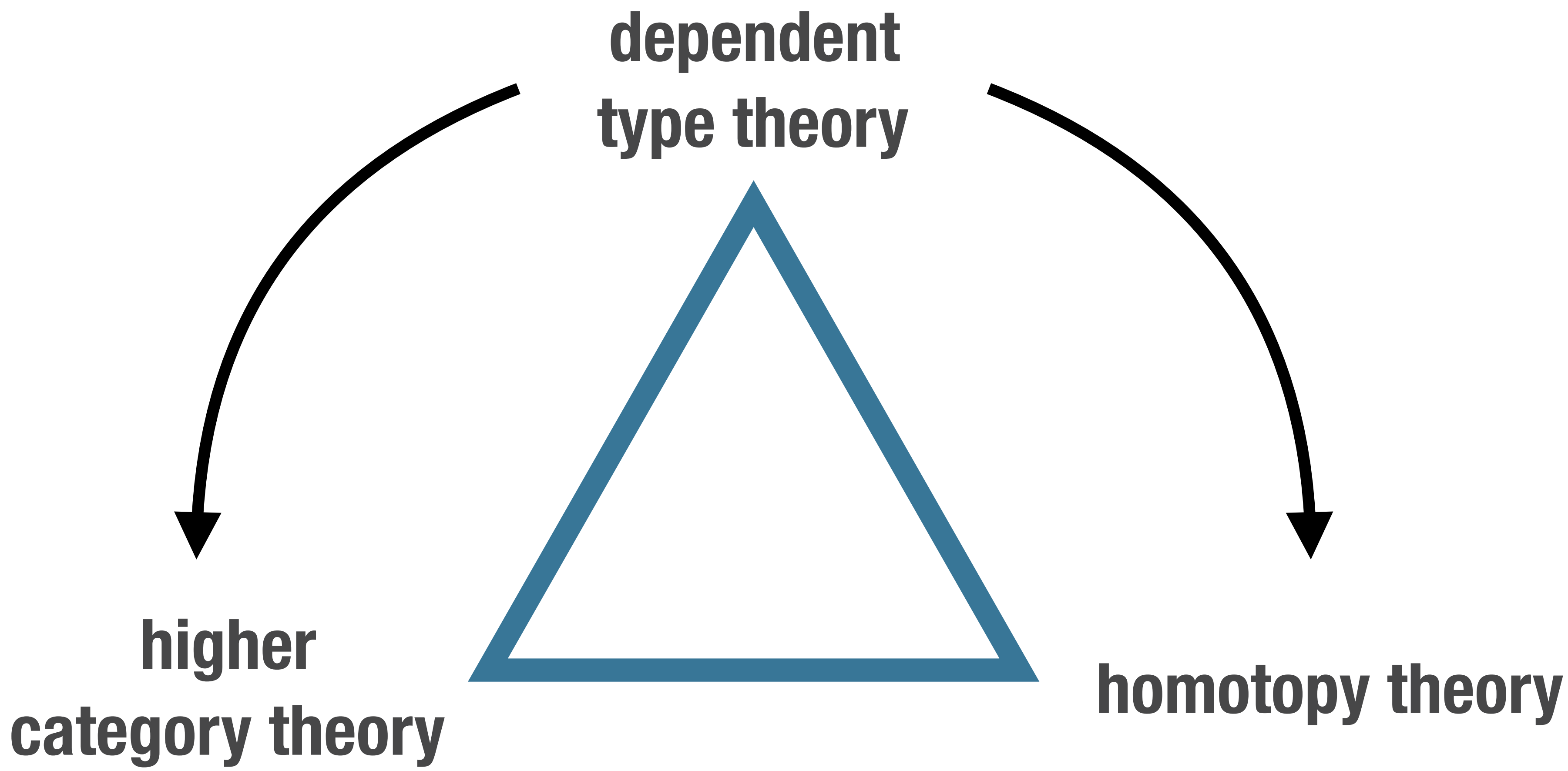
Haskell

Semantics



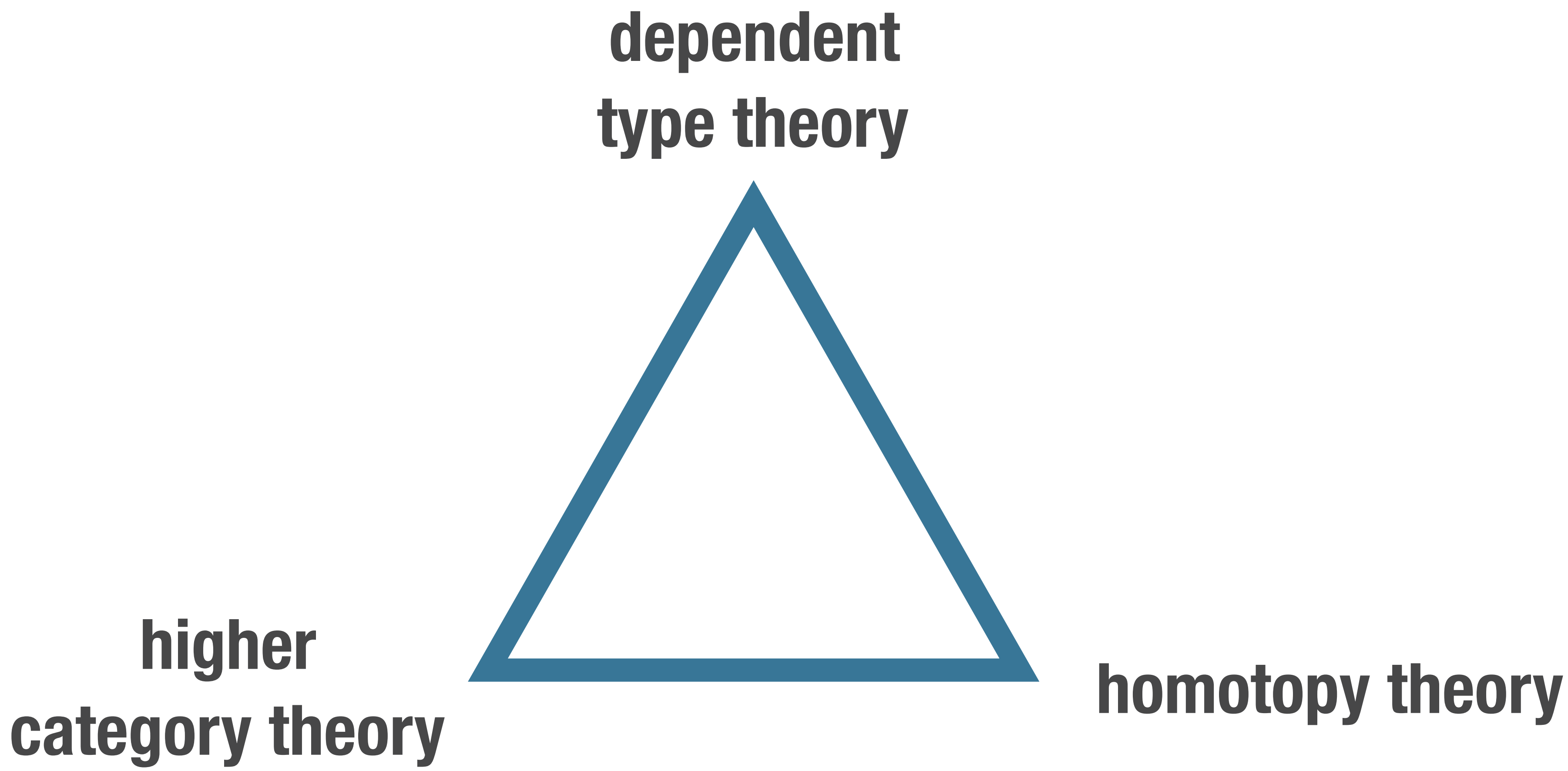
Awodey, van den Berg, Gambino, Garner, Hofmann, Lumsdaine, Streicher, Voevodsky, Warren 1994-2010

Semantics



Awodey, van den Berg, Gambino, Garner, Hofmann, Lumsdaine, Streicher, Voevodsky, Warren 1994-2010

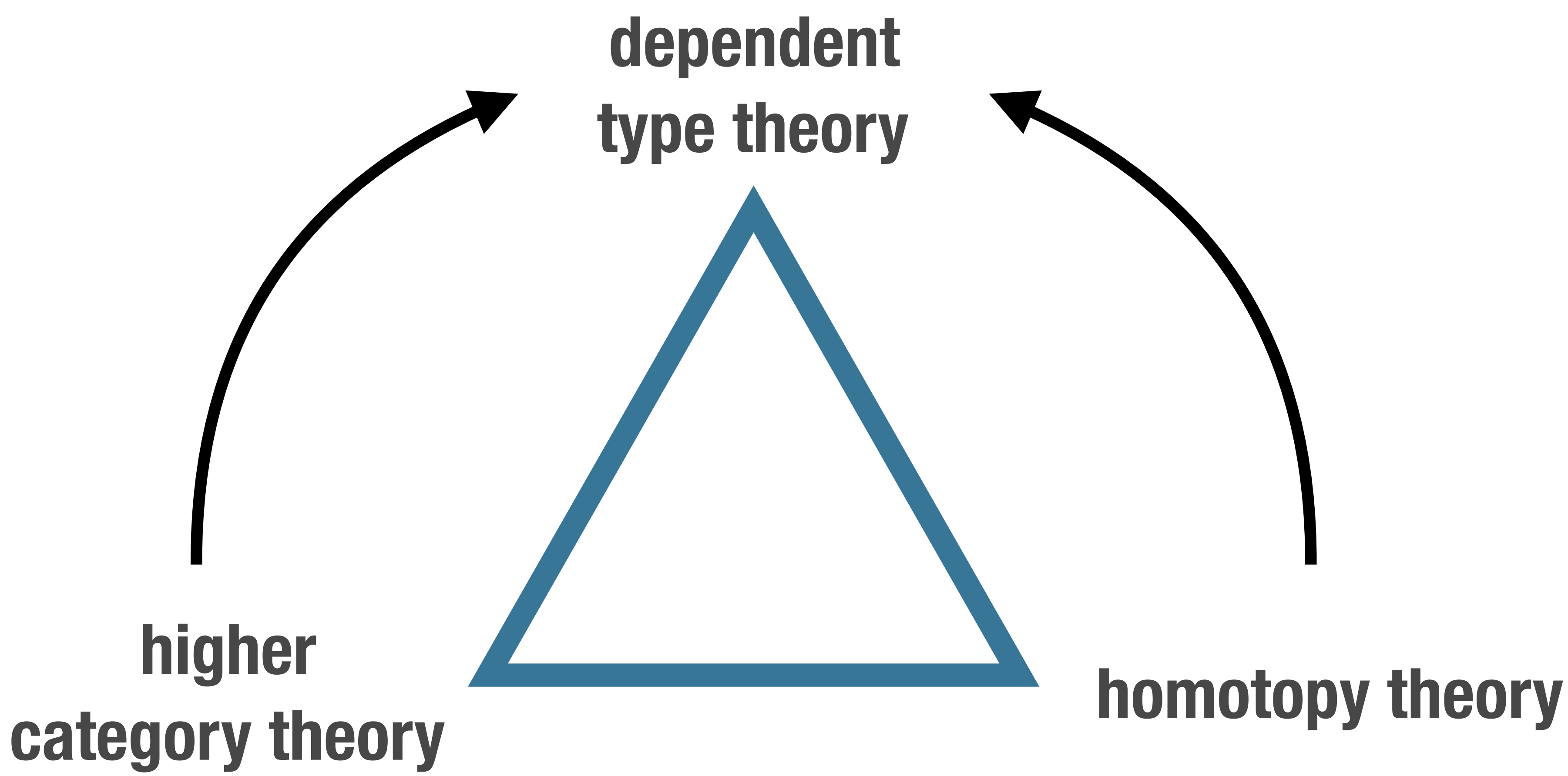
Principles



Univalence [Voevodsky, 2006]

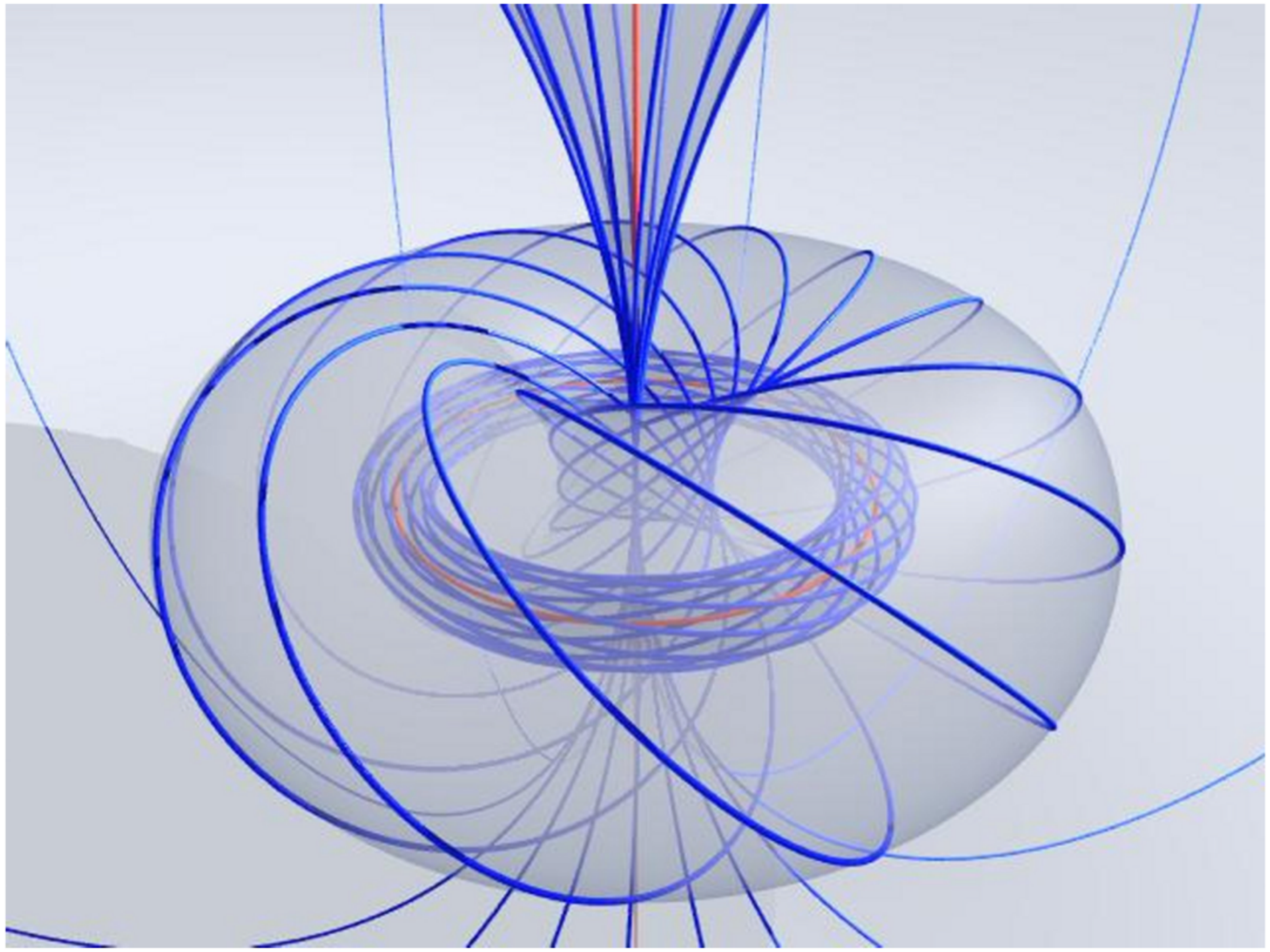
Higher inductive types [Bauer,Lumsdaine,Shulman,Warren, 2011]

Principles

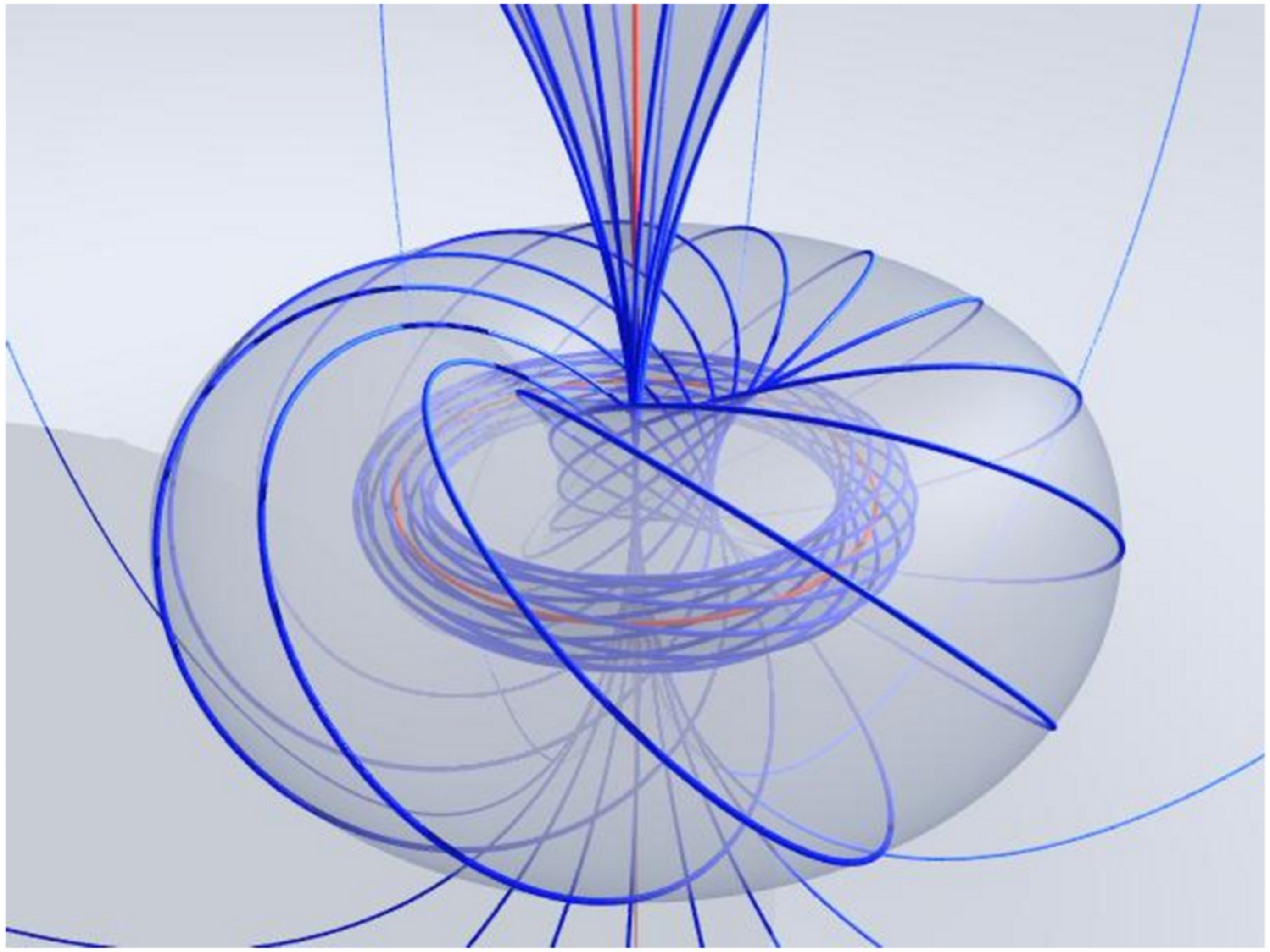


Univalence [Voevodsky, 2006]
Higher inductive types [Bauer,Lumsdaine,Shulman,Warren, 2011]

FP as a language for objects of higher homotopy type



FP as a language for objects of higher homotopy type



```
rotLoop : (a : S1) -> Id S1 a a =
split
  base -> loopS1
  loop @ i -> constSquare @ i

rot : S1 -> S1 -> S1 = split
  base -> (\ (y : S1) -> y)
  loop @ i ->
    (\ (y : S1) -> rotLoop y @ i)

rotpath (x : S1) : Id U S1 S1 =
  ua (rot x, ...)

H : S2 -> U = split
  north -> S1
  south -> S1
  merid a @ x -> rotpath a @ x
```

Mechanized proofs

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

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

Hopf fibrations

$\pi_2(S^2) = \mathbb{Z}$

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

Freudenthal

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

K(G,n)

Blakers-Massey

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

Projective Spaces

Van Kampen

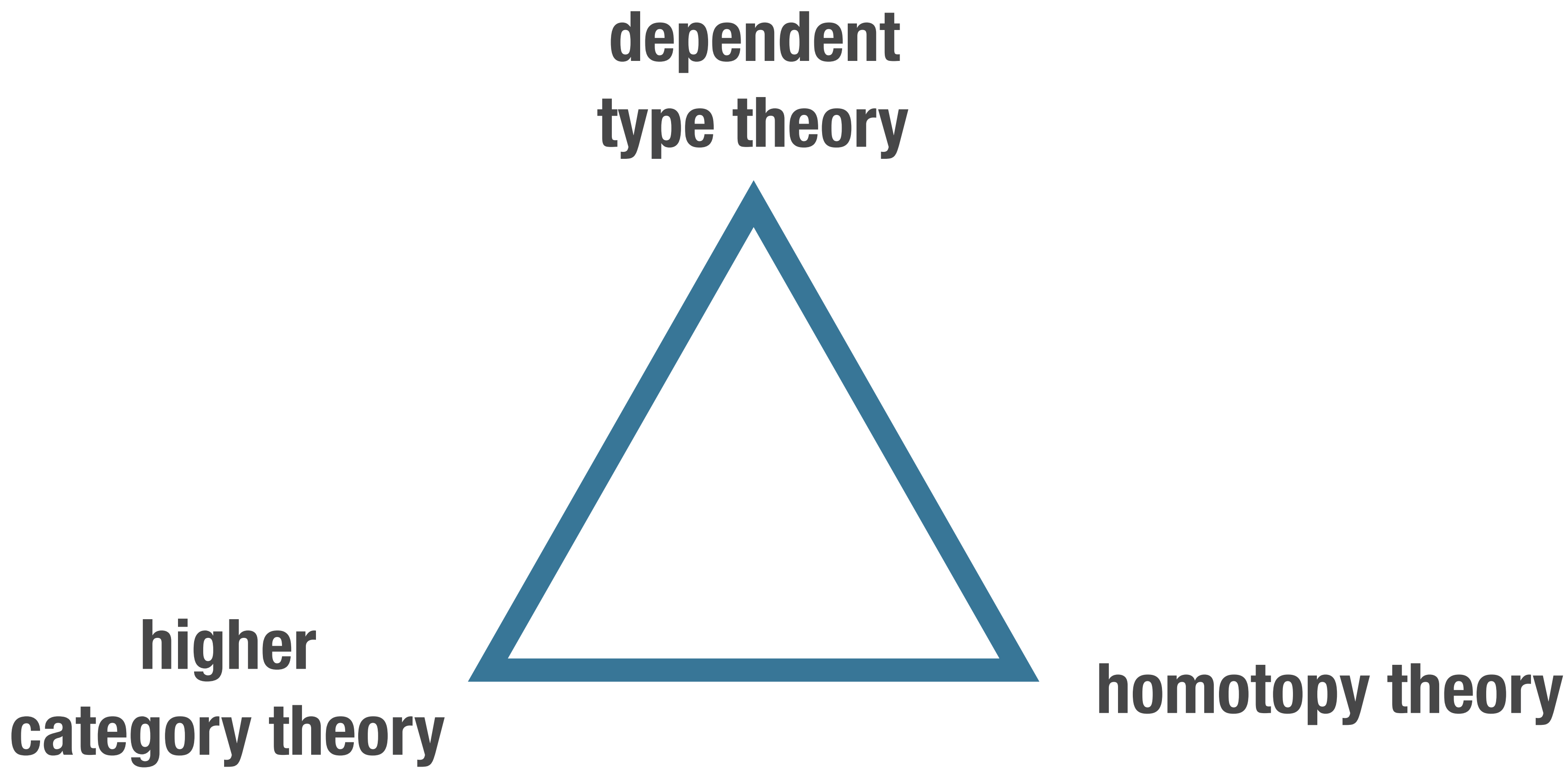
Covering spaces

Whitehead for n-types

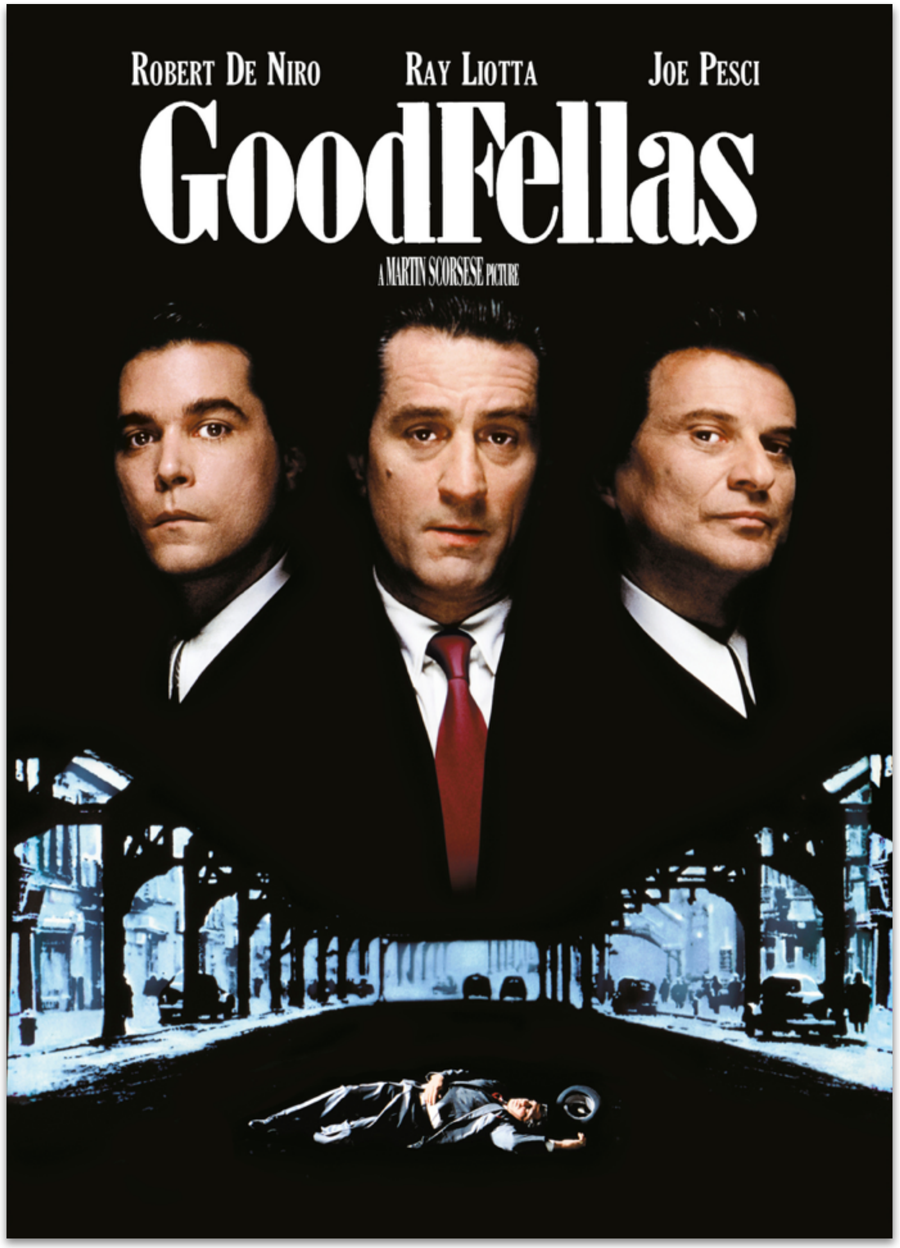
Cohomology axioms

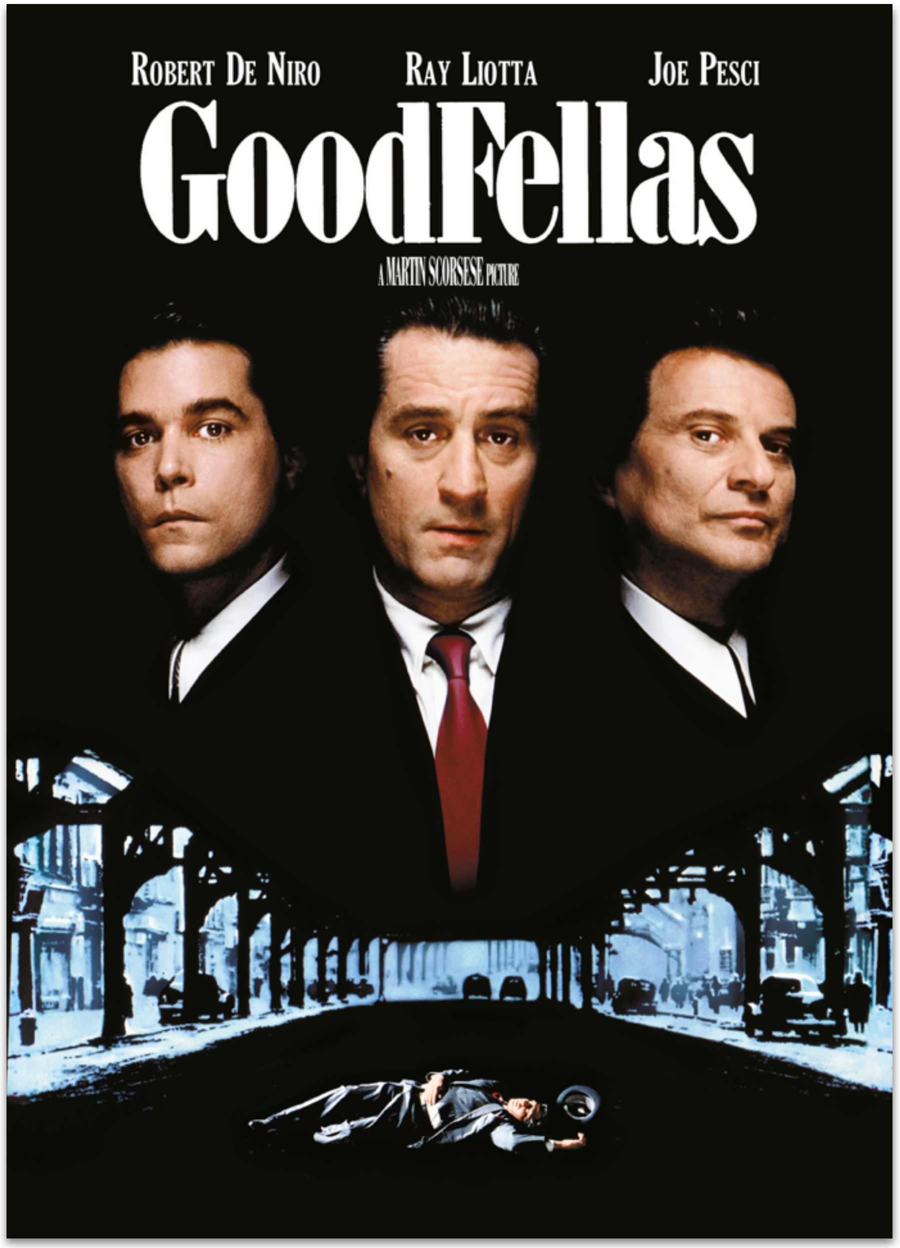
Mayer-Vietoris

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

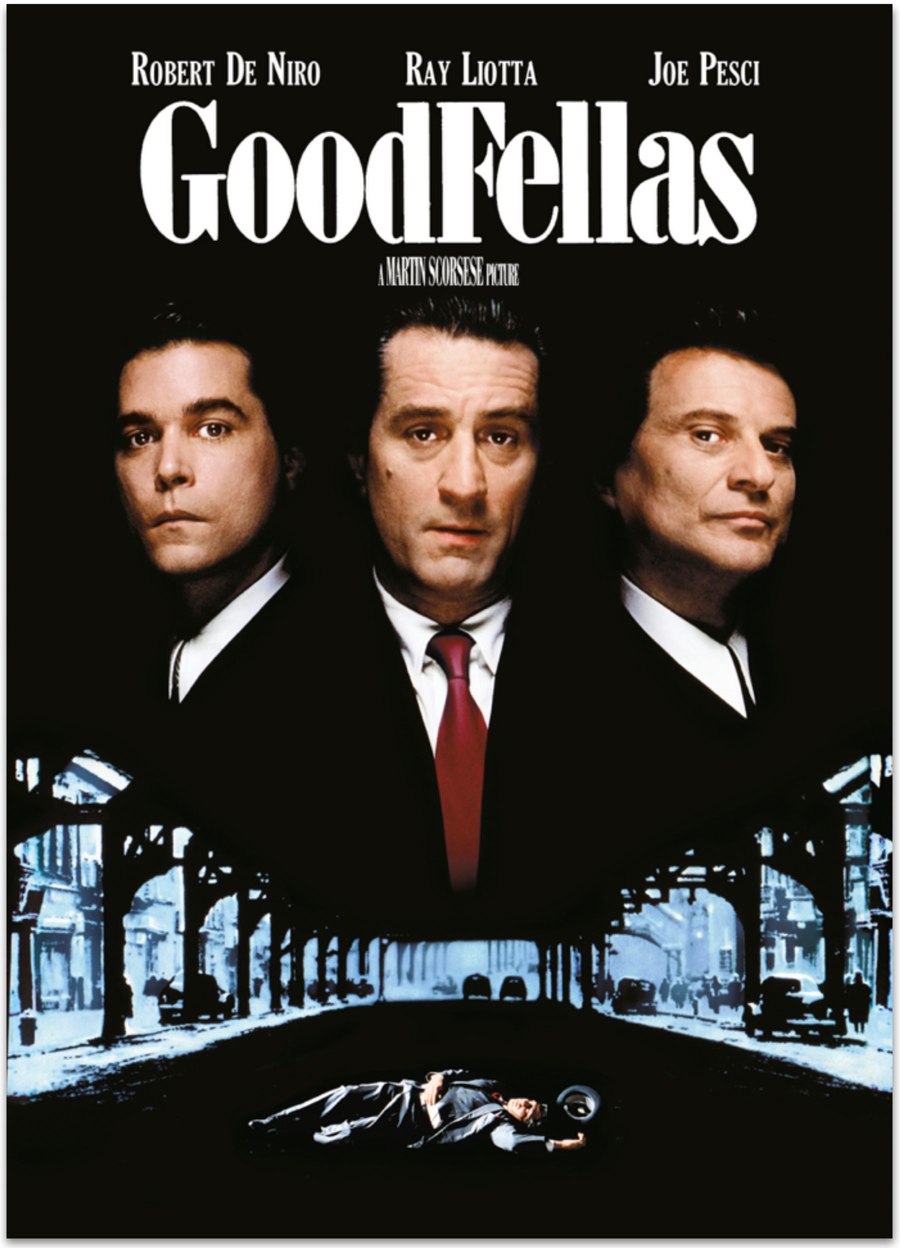

dependent
type theory
higher
category theory
homotopy theory

What does this all mean in programming terms?

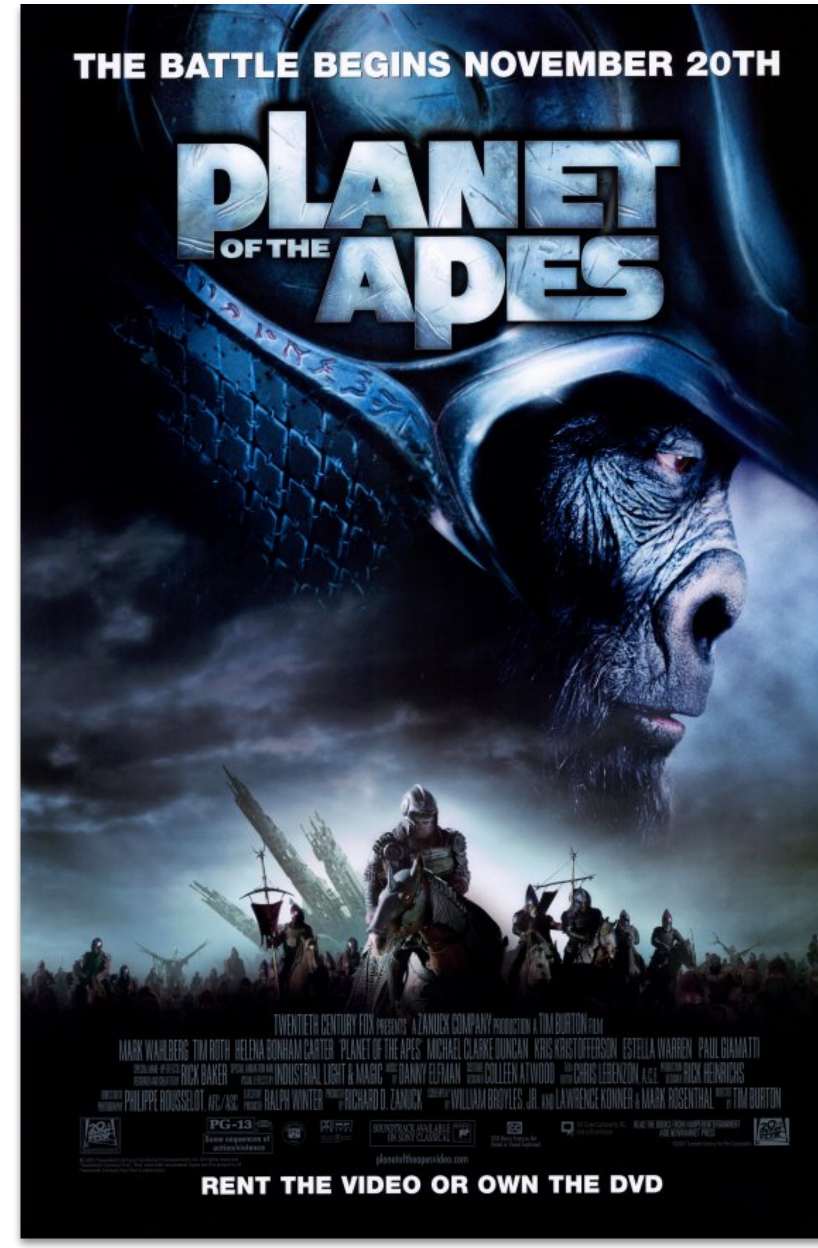

ROBERT DE NIRO
RAY LIOTTA
JOE PESCI
GoodFellas
A MARTIN SCORSESE PICTURE

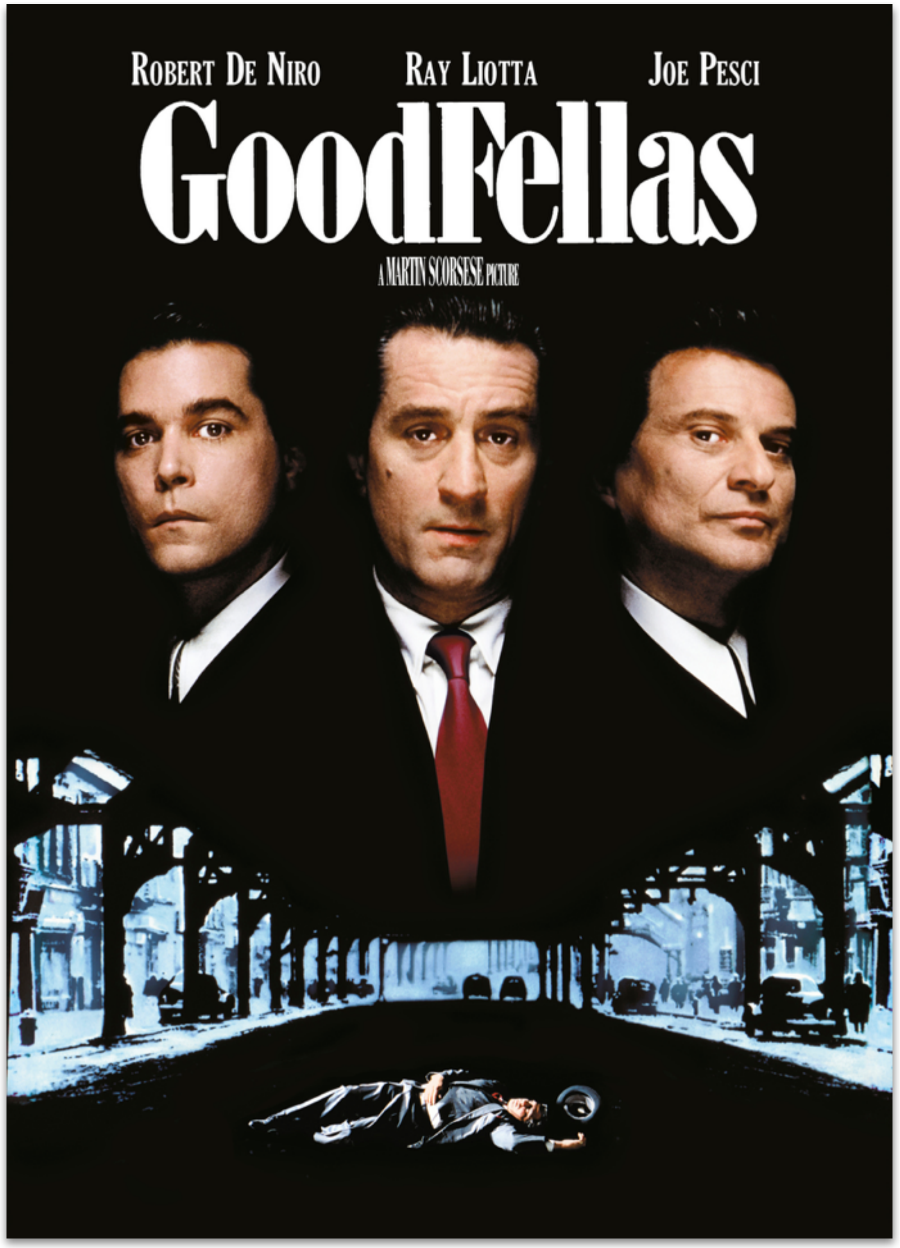

ROBERT DE NIRO
RAY LIOTTA
JOE PESCI
GoodFellas
A MARTIN SCORSESE PICTURE

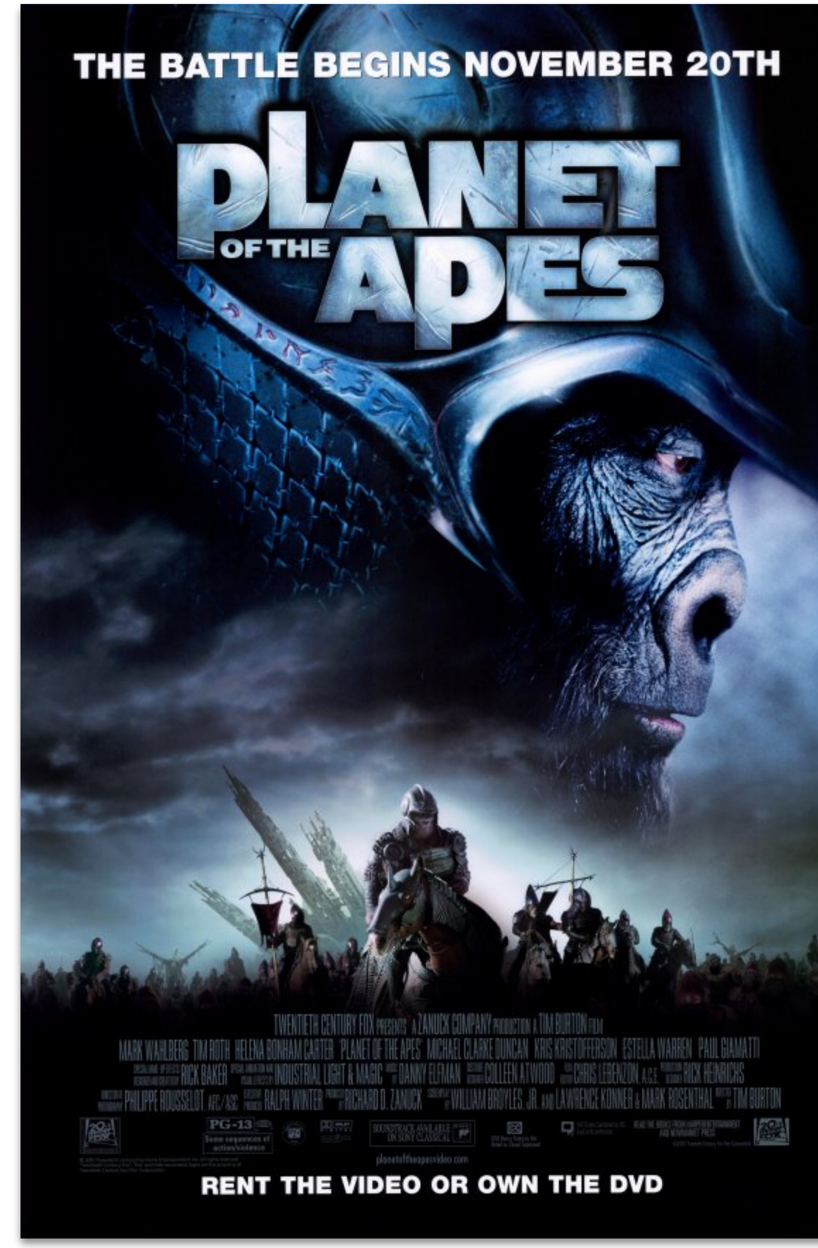

ROBIN WILLIAMS
JAKOB THE LIAR


ROBERT DE NIRO
RAY LIOTTA
JOE PESCI
GoodFellas
A MARTIN SCORSESE PICTURE

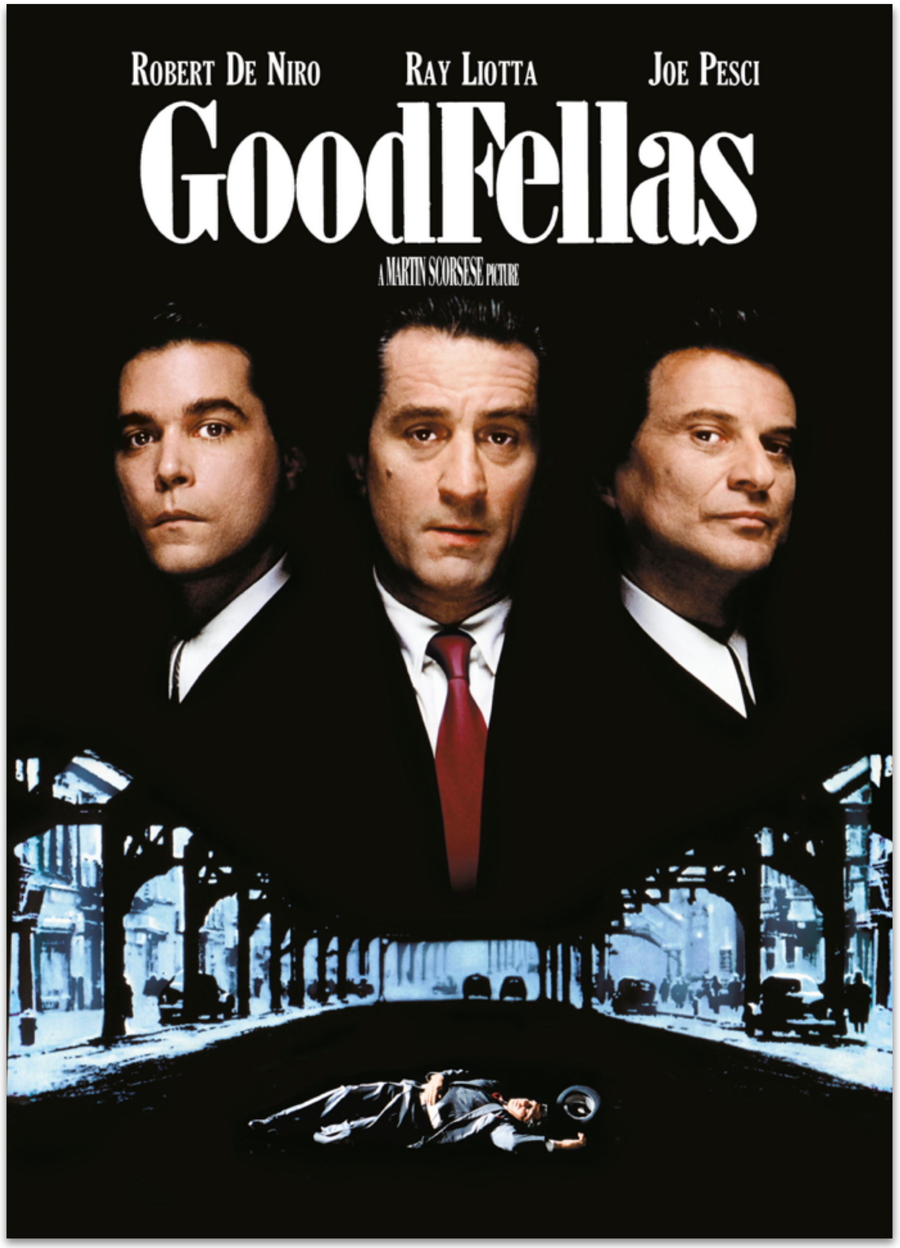

ROBIN WILLIAMS
JAKOB THE LIAR

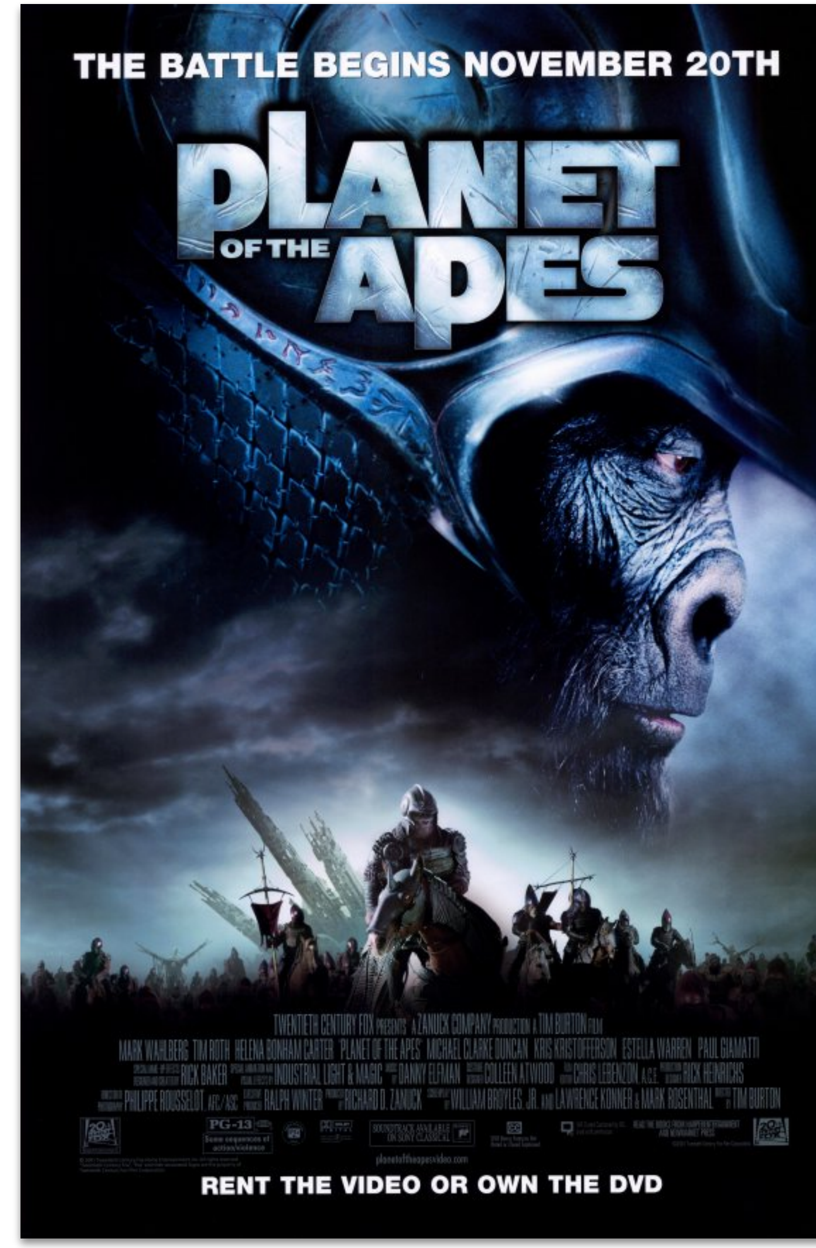

THE BATTLE BEGINS NOVEMBER 20TH
PLANET OF THE APES
RENT THE VIDEO OR OWN THE DVD





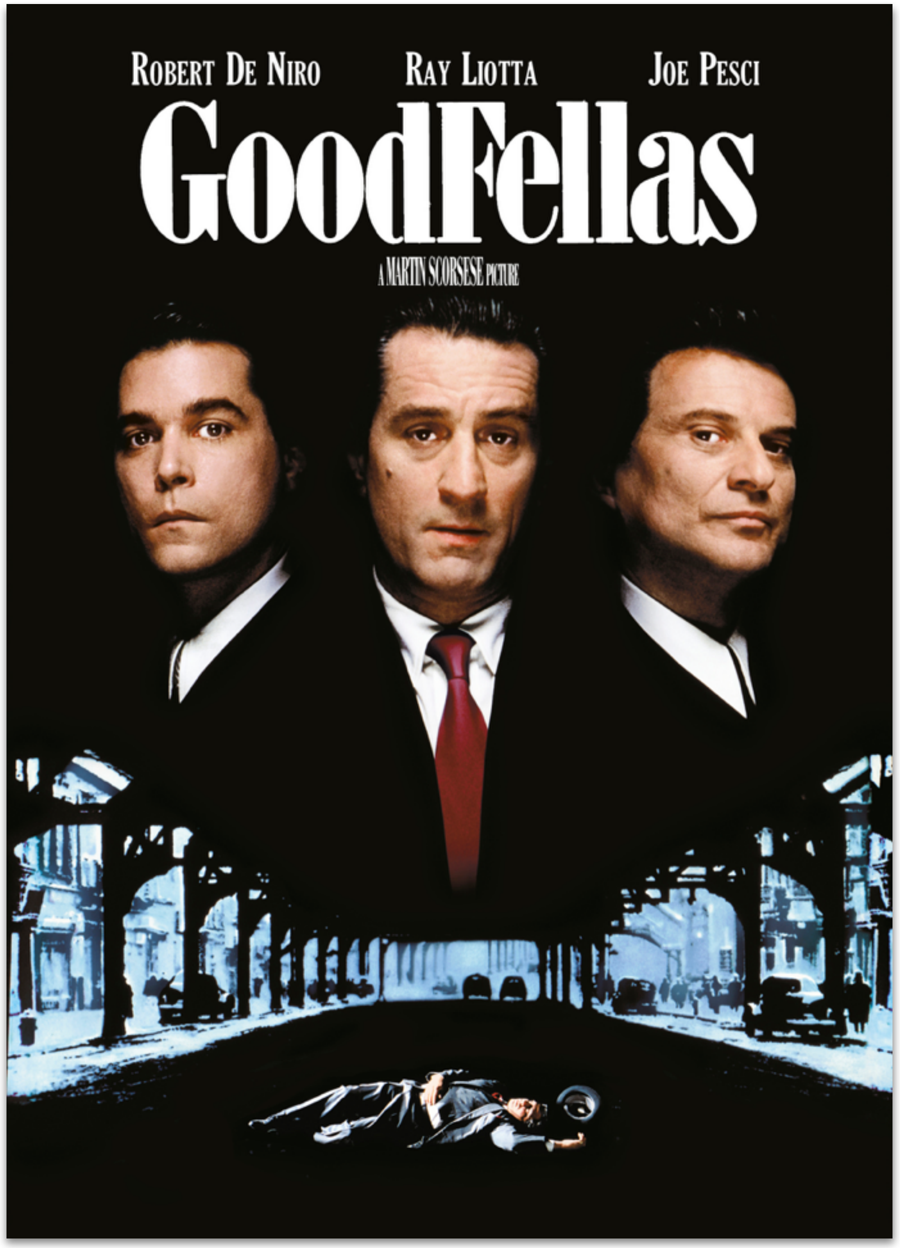
In a world that's powered by violence...





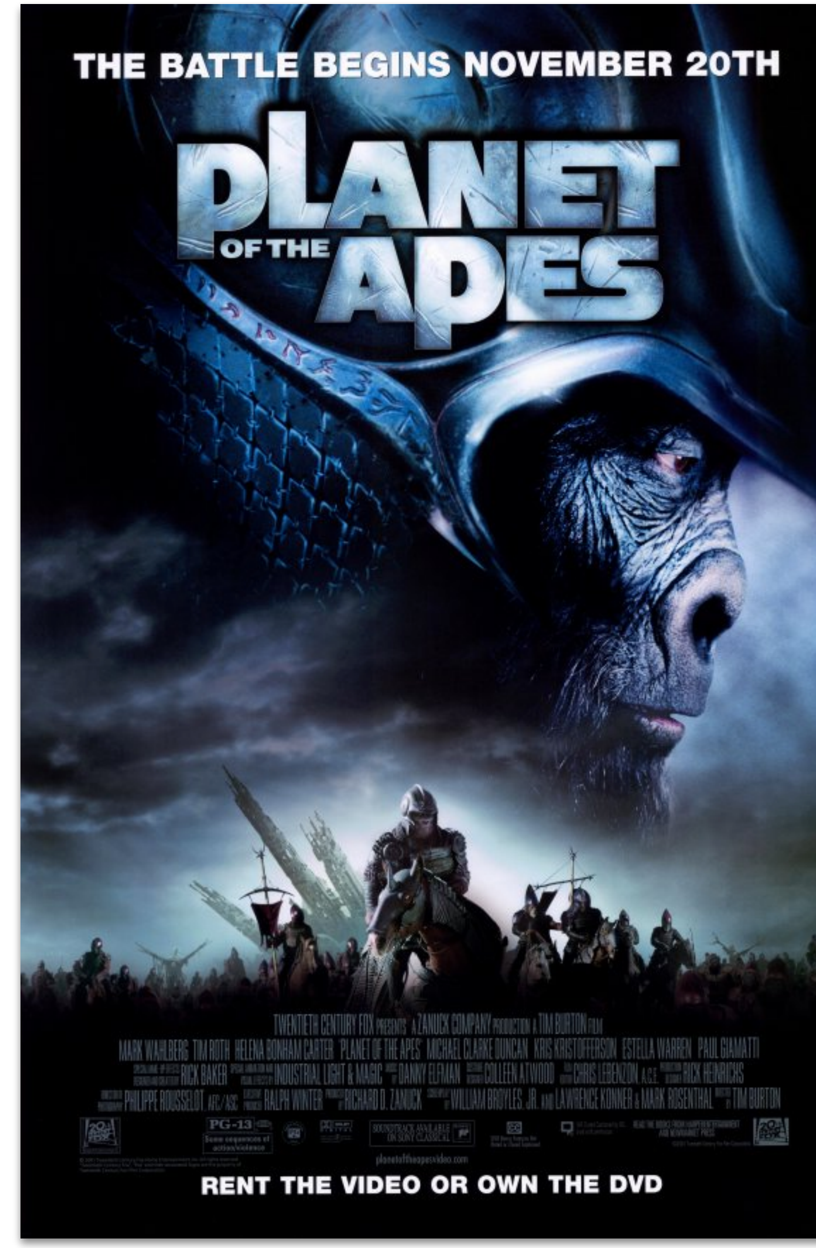
In a world that's powered by violence...

In a world where owning a radio was strictly forbidden…



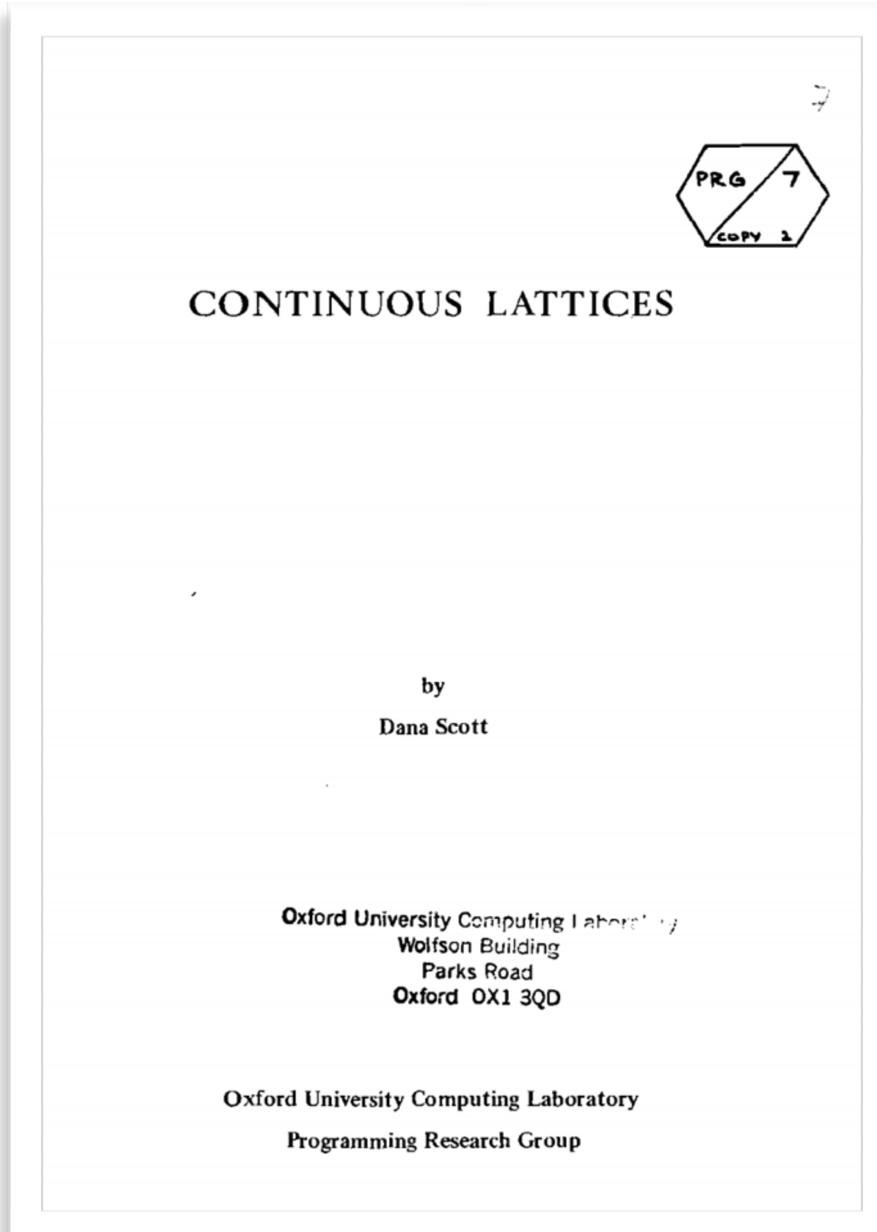
In a world that's powered by violence...

In a world where owning a radio was strictly forbidden…



In a world where freedom is history…

In a world where all functions are monotone and preserve least upper bounds…



PRG 7 COPY 2

CONTINUOUS LATTICES

by

Dana Scott

Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD

Oxford University Computing Laboratory
Programming Research Group

In a world where all functions are monotone and preserve least upper bounds…

In a world where all functions are continuous…

$$
\begin{tikzcd}
\lambda\text{-terms} \arrow[d] \\
\text{CPOs}
\end{tikzcd}
$$

In a world where all functions are continuous…

λ-terms

CPOs

`λf.λx.λy.f x y`

In a world where all functions are continuous…

λ-terms

$\downarrow$

CPOs

`λf.λx.λy.f x y`

$\downarrow$

function with the *property* of being continuous

In a world where all functions are continuous…

λ-terms → CPOs

`Y(f) = f(Y(f))` → something that only exists in that world

In a world where all functions secretly **are** something…

In a world where all functions secretly **do** something…

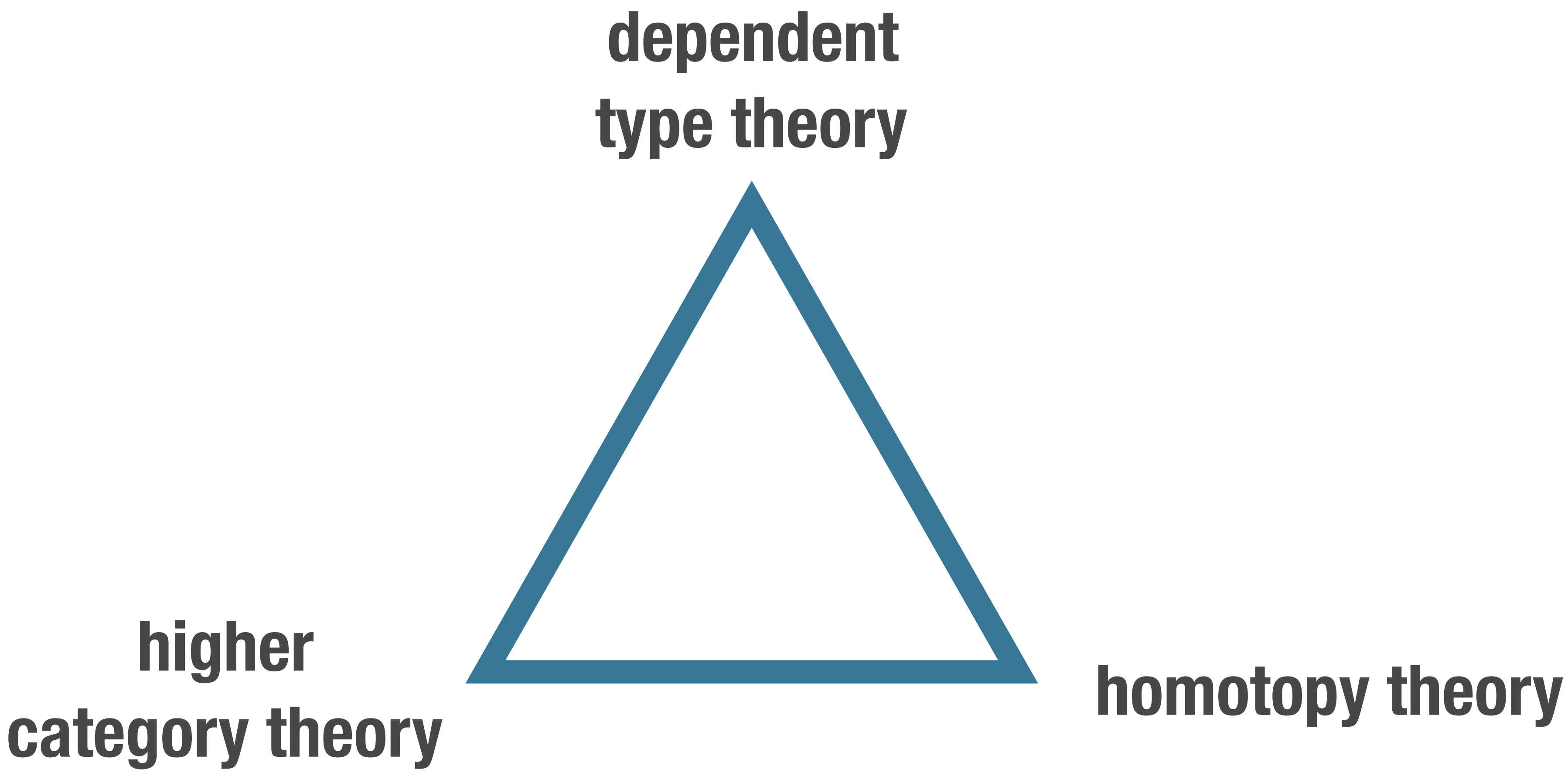
In a world where all functions secretly **do** something…

- get “code for free” / generic programs

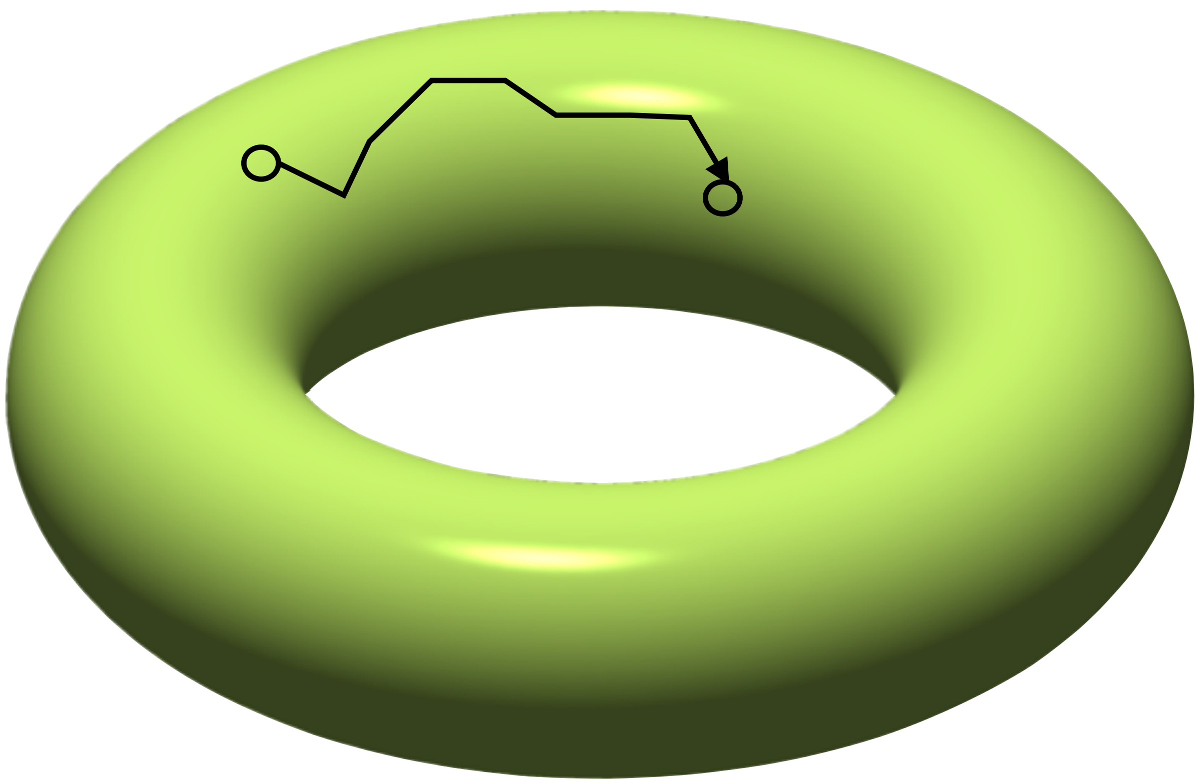
In a world where all functions secretly **do** something…

- get “code for free” / generic programs
- can add new principles that depend on them

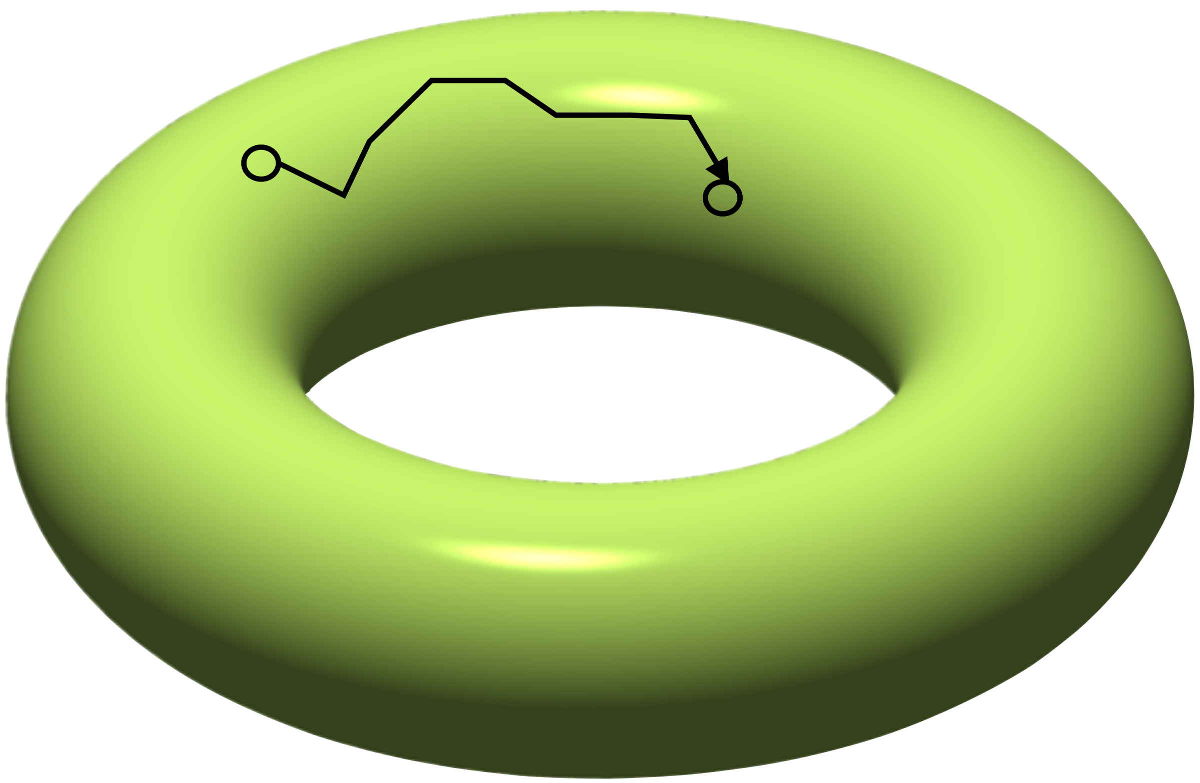
Homotopy type theory



In a world where types are spaces

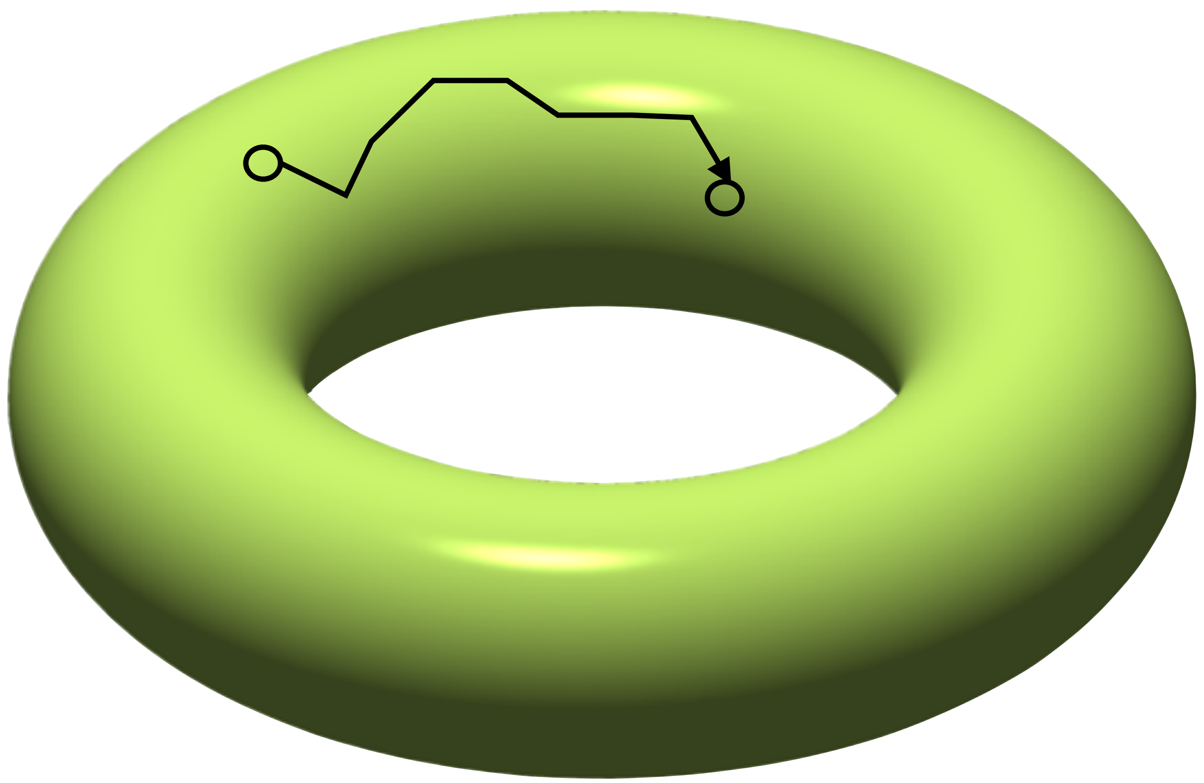


In a world where types are spaces



each type is a space,
with points and paths

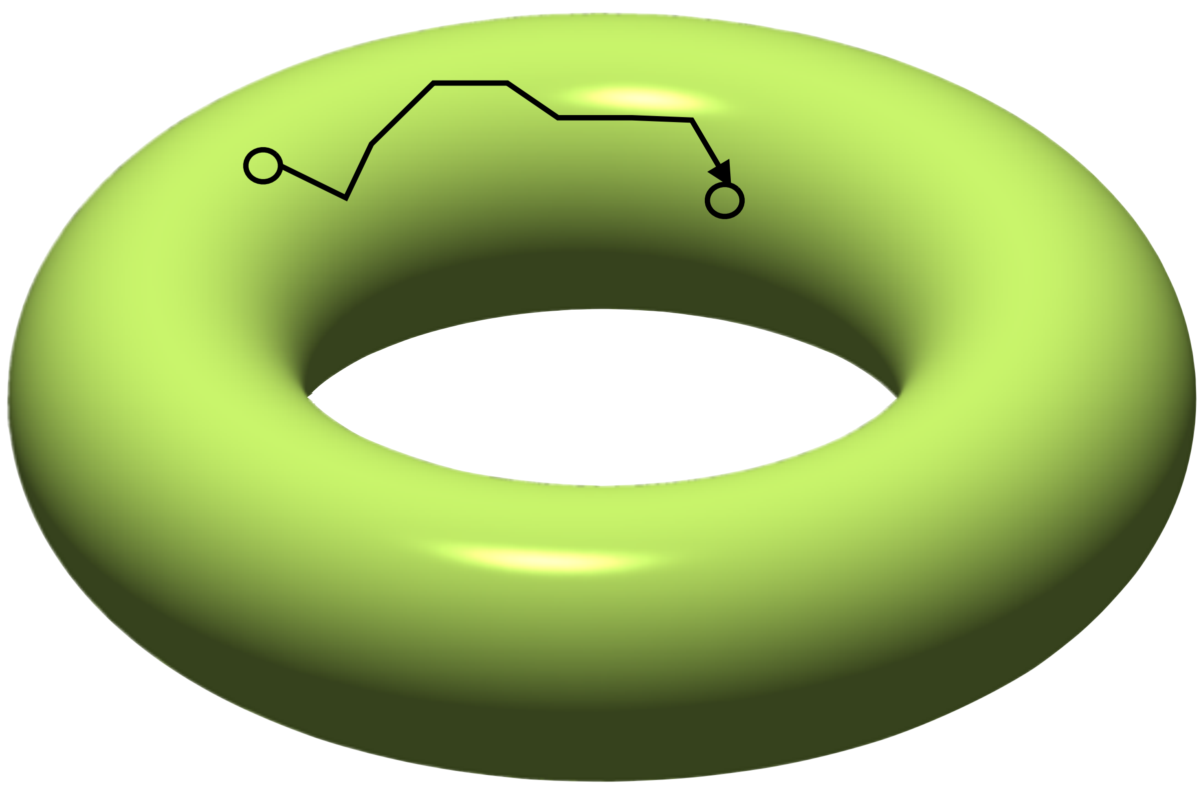
In a world where types are spaces



each type is a space,
with points and paths

programs are points

In a world where types are spaces

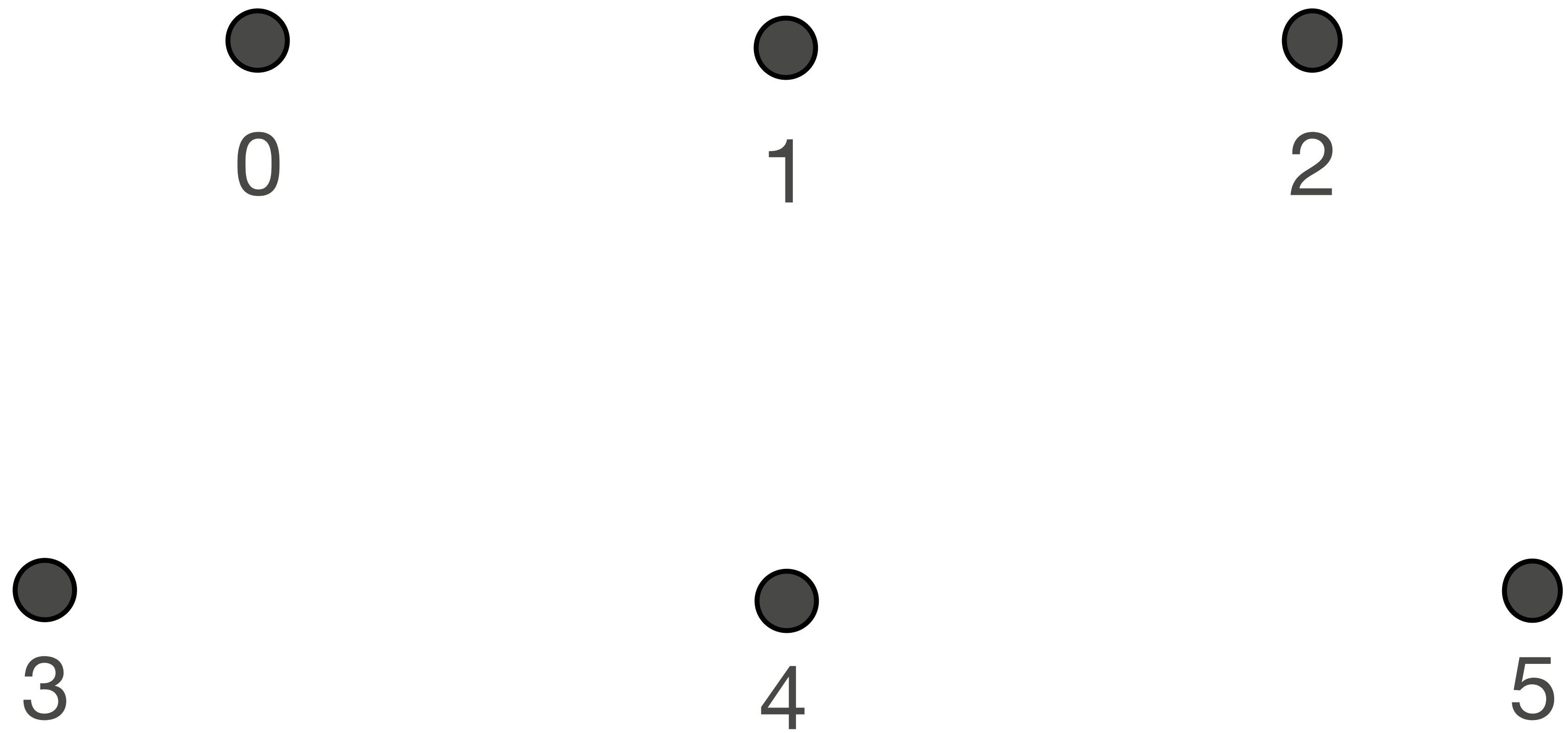


each type is a space, with points and paths

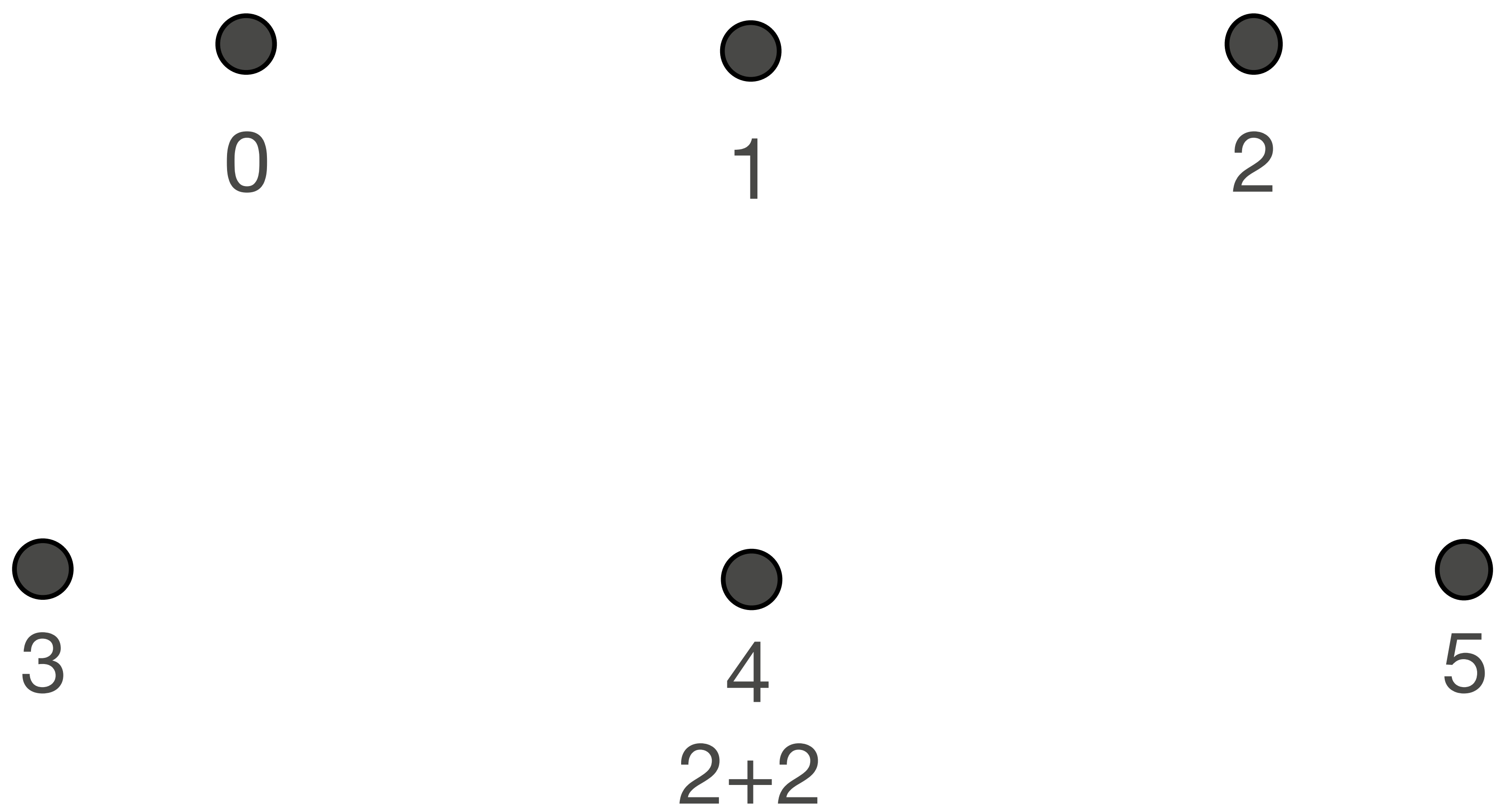
programs are points

points can be "literally the same" or connected by a path

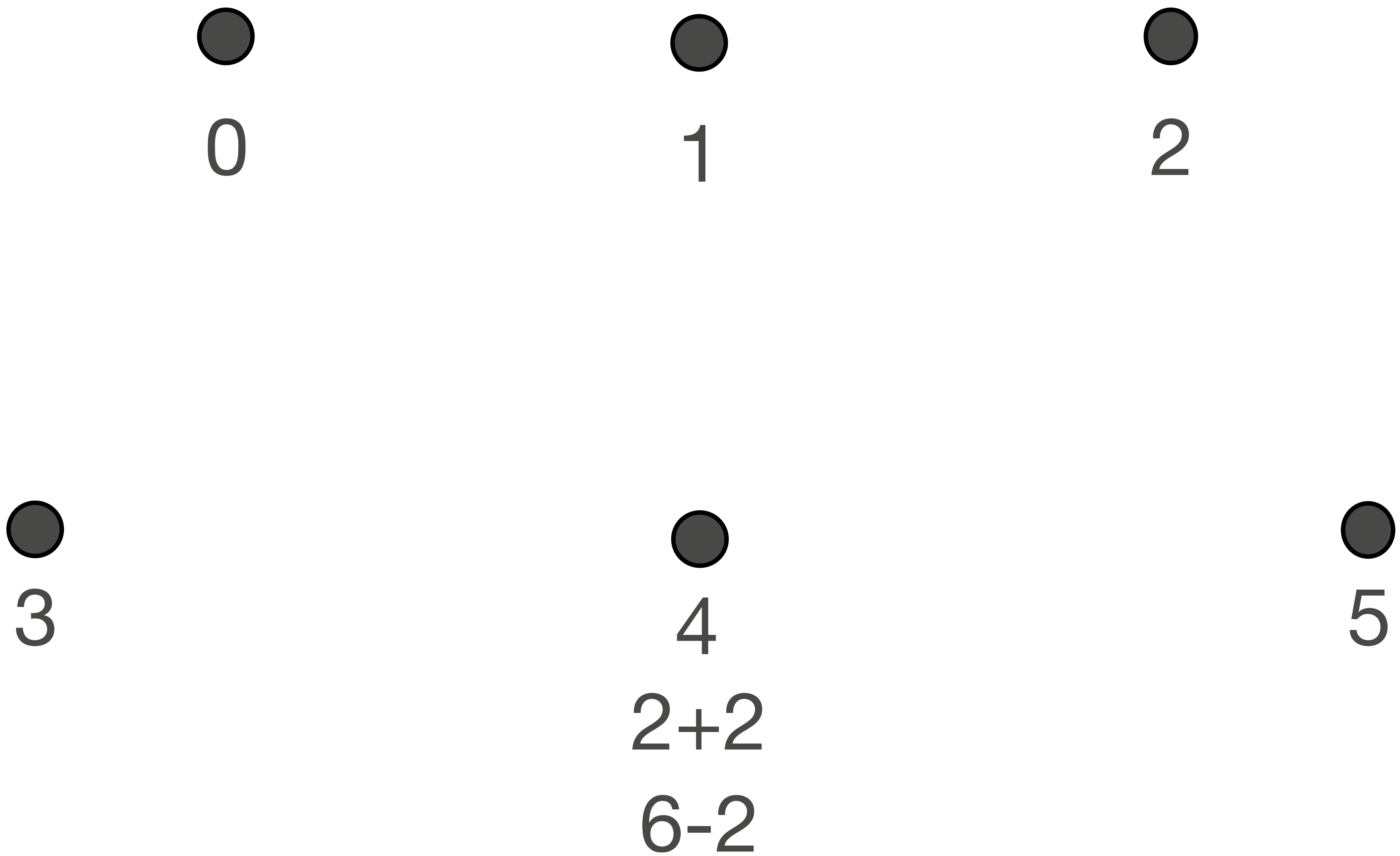
Many types are discrete ($\mathtt{Nat}$)



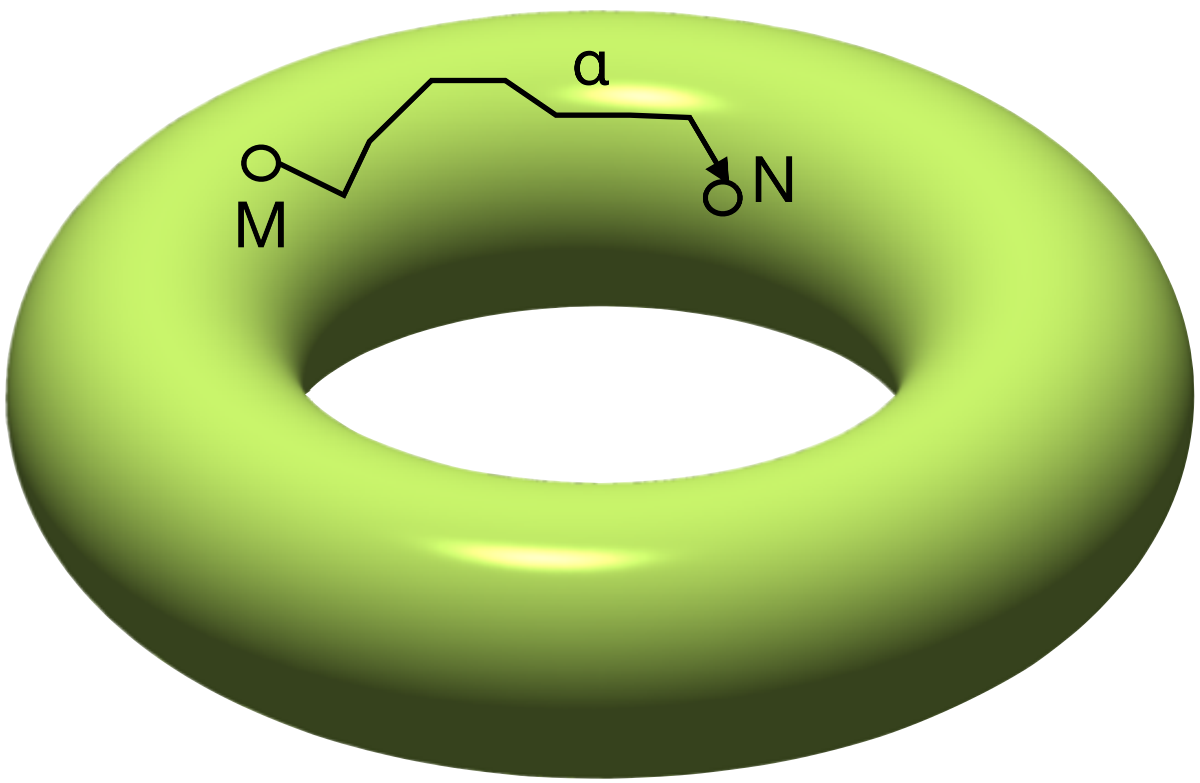
Many types are discrete (`Nat`)



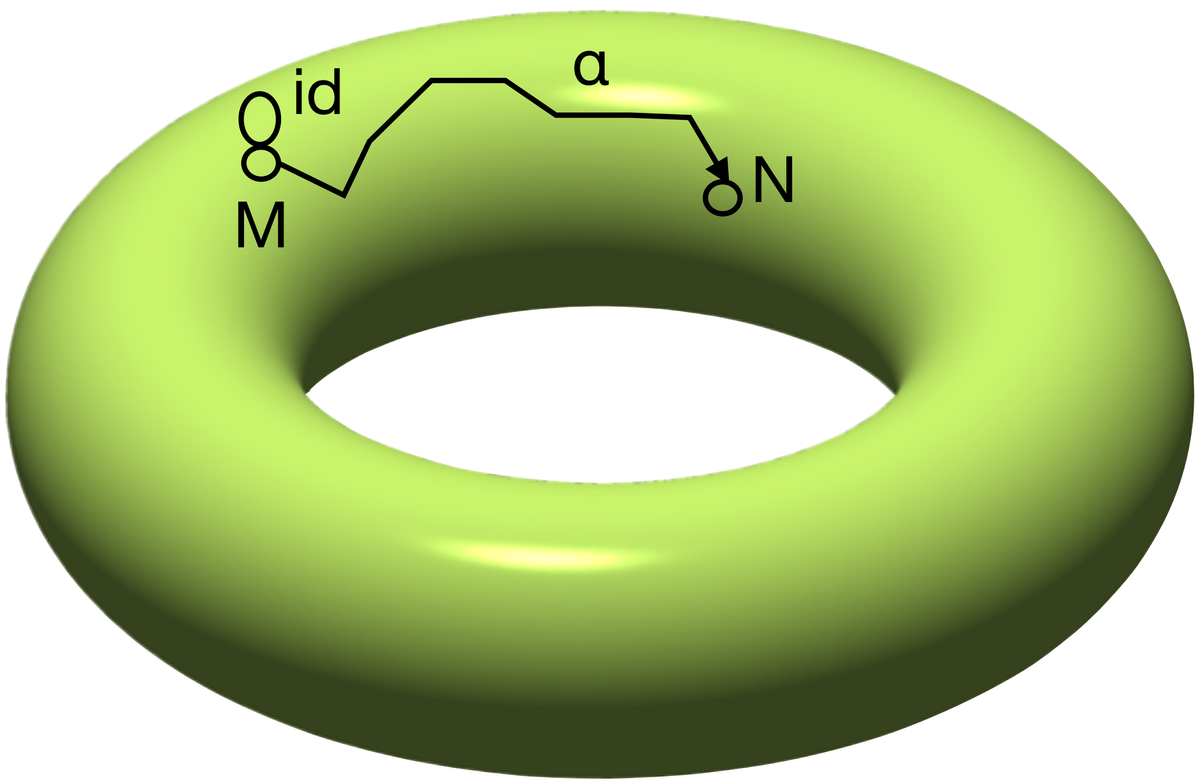
Many types are discrete (`Nat`)



Paths look like equality



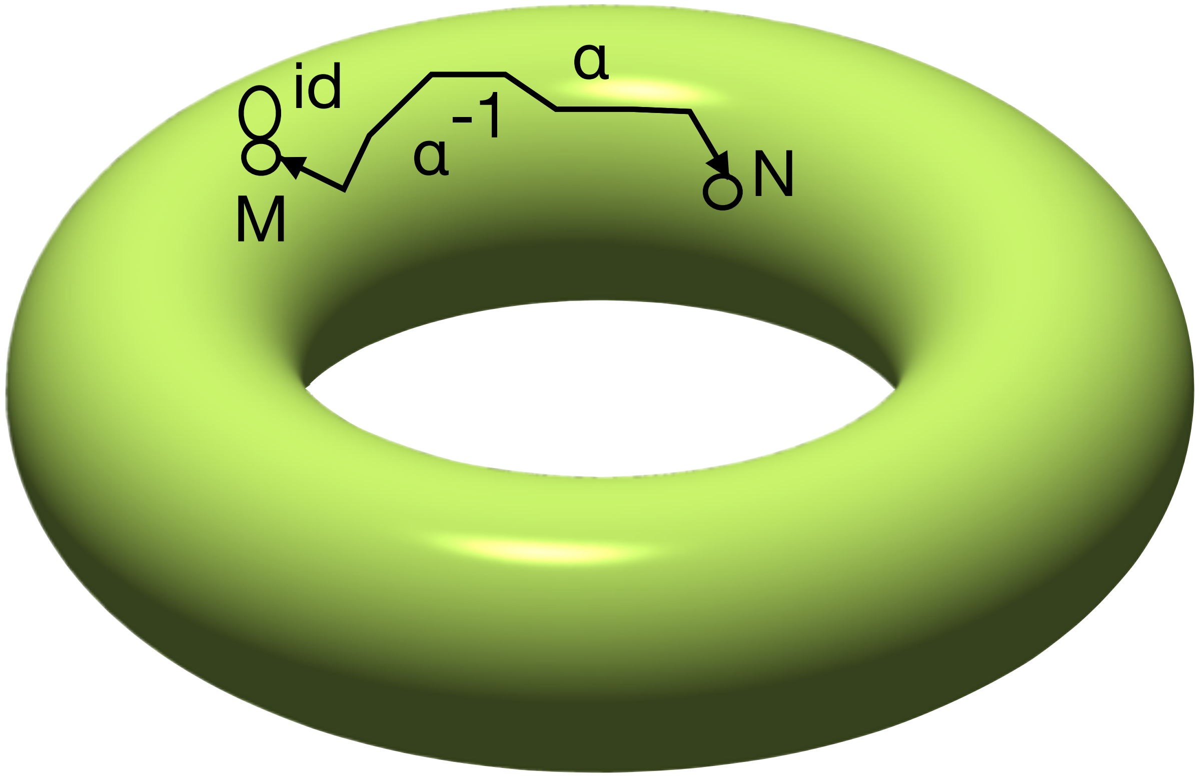
Paths look like equality



reflexivity

```
id : Path M M
```

Paths look like equality



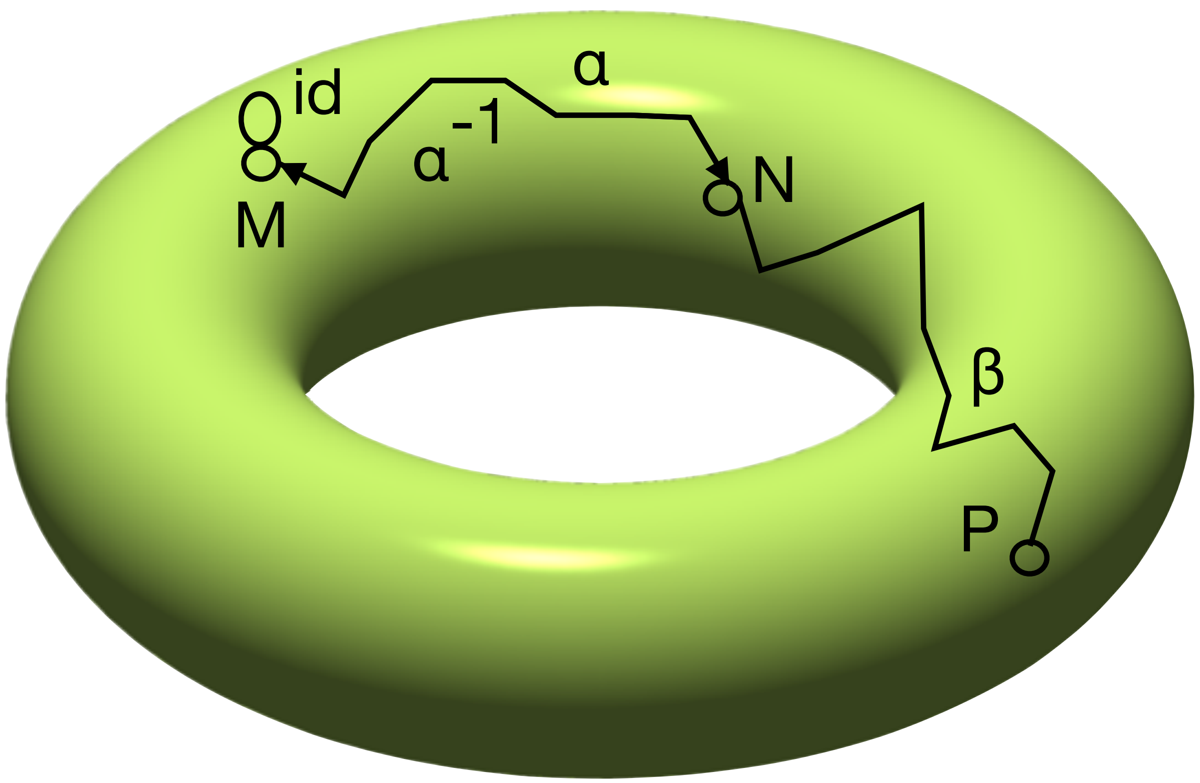
reflexivity

```
id : Path M M
```

symmetry

α^{-1} : Path N M

Paths look like equality



reflexivity

`id : Path M M`

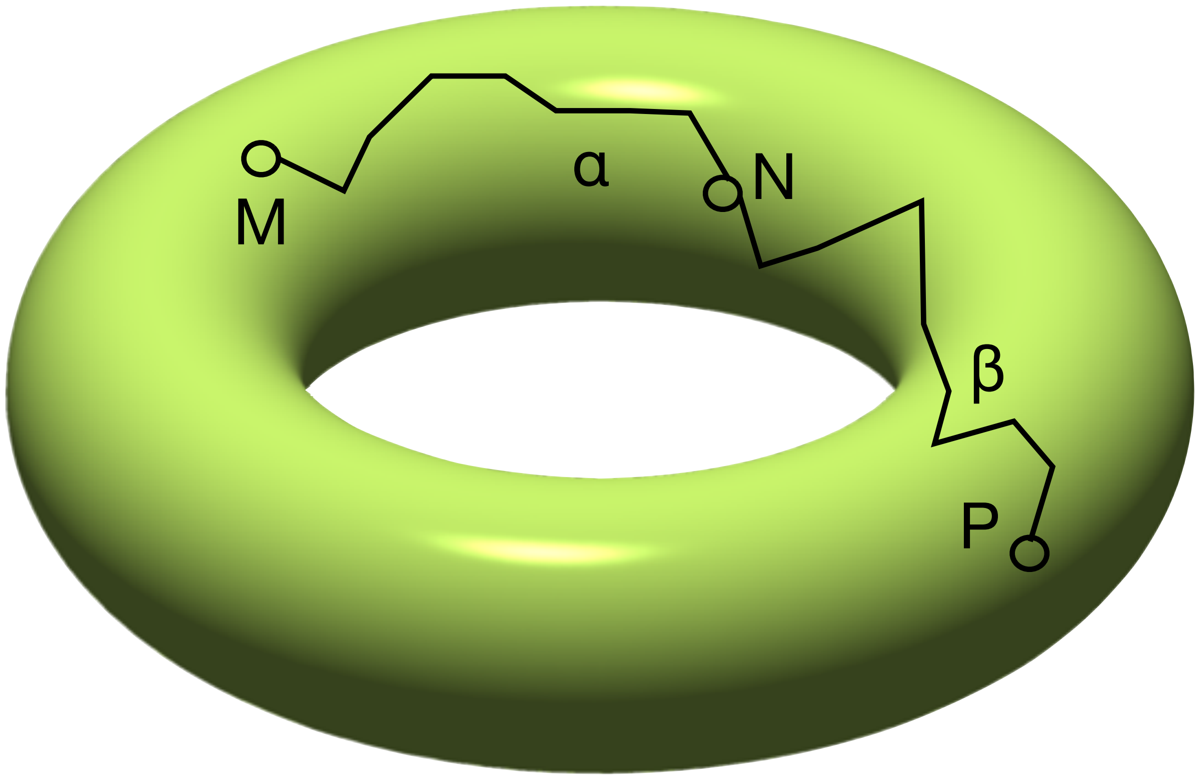
symmetry

α^{-1} `: Path N M`

transitivity

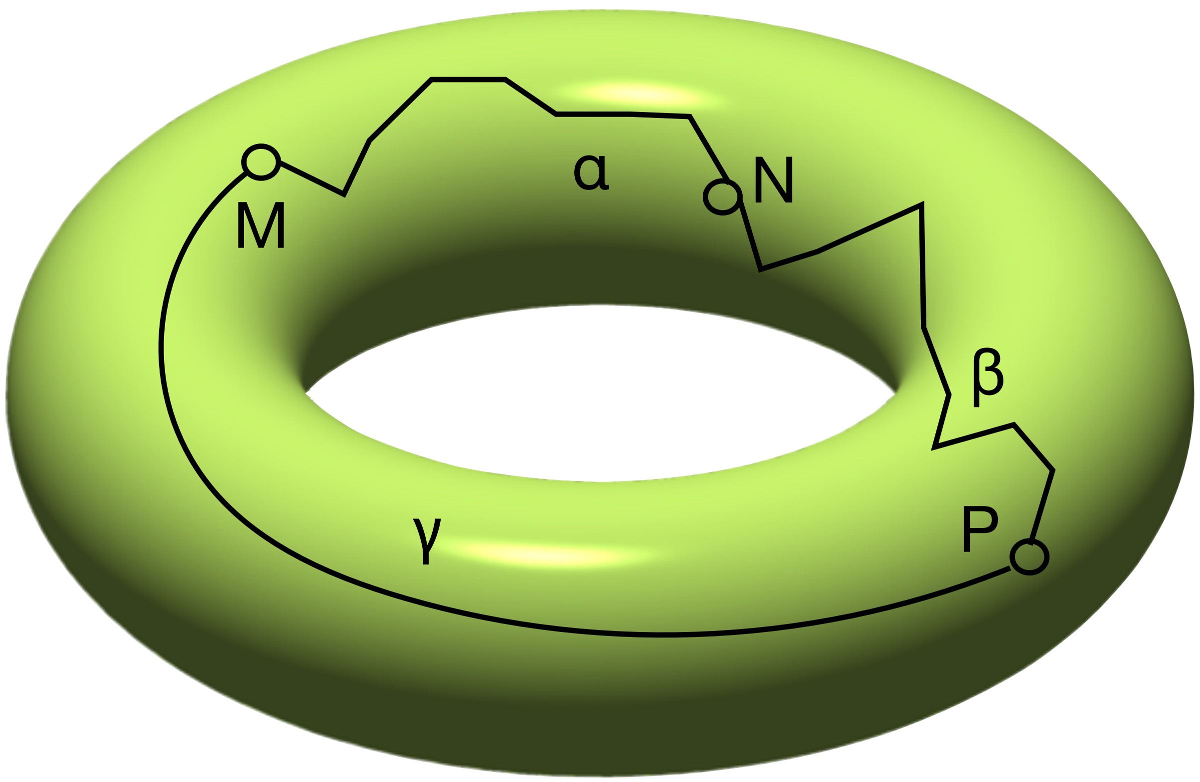
$\beta \circ \alpha$ `: Path M P`

But are *data*



`β ○ α : Path M P`

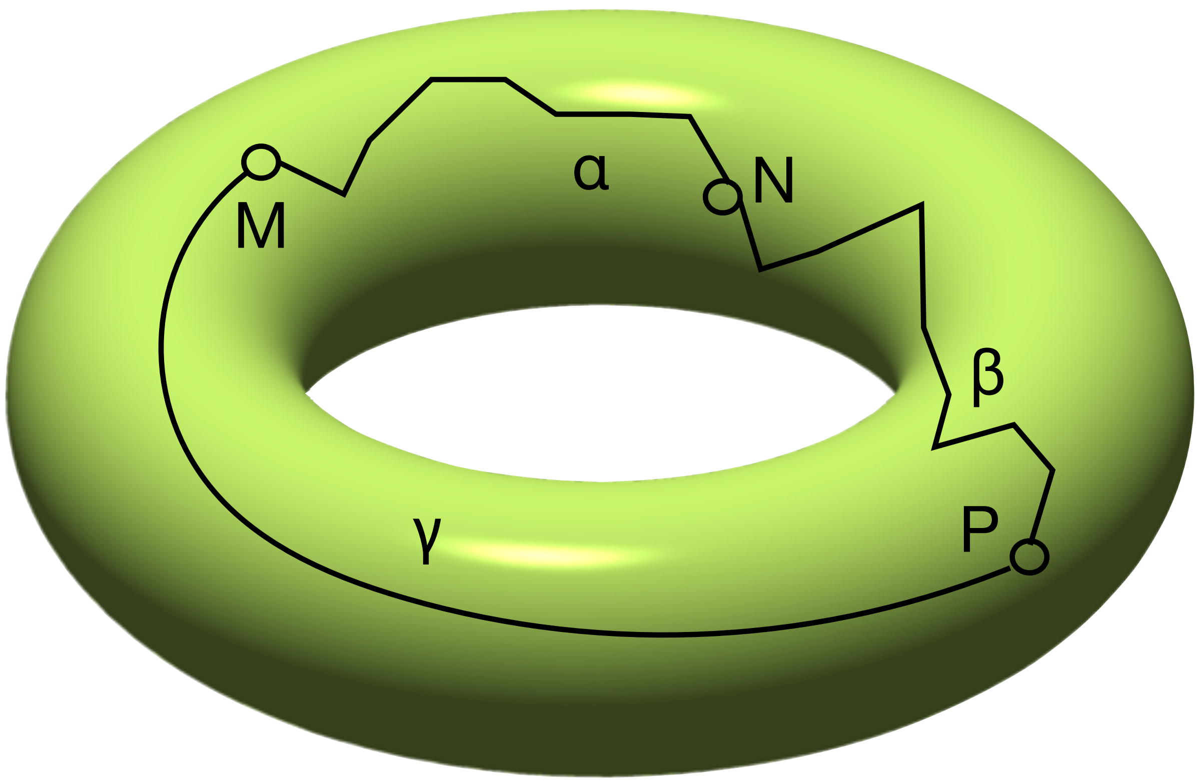
But are *data*



$$\beta \circ \alpha : \mathrm{Path}\ M\ P$$

$$\gamma : \mathrm{Path}\ M\ P$$

But are *data*

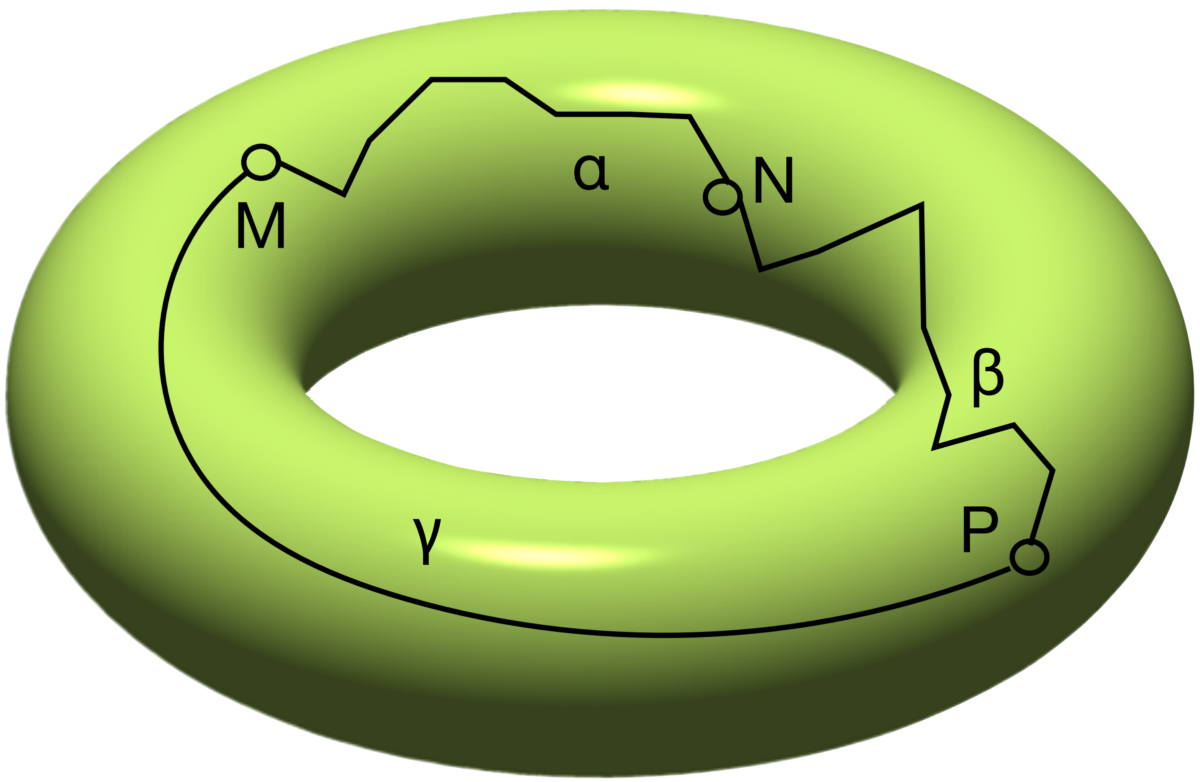


$\beta \circ \alpha$: Path M P

γ : Path M P

$(\beta \circ \alpha) \neq \gamma$

But are *data*



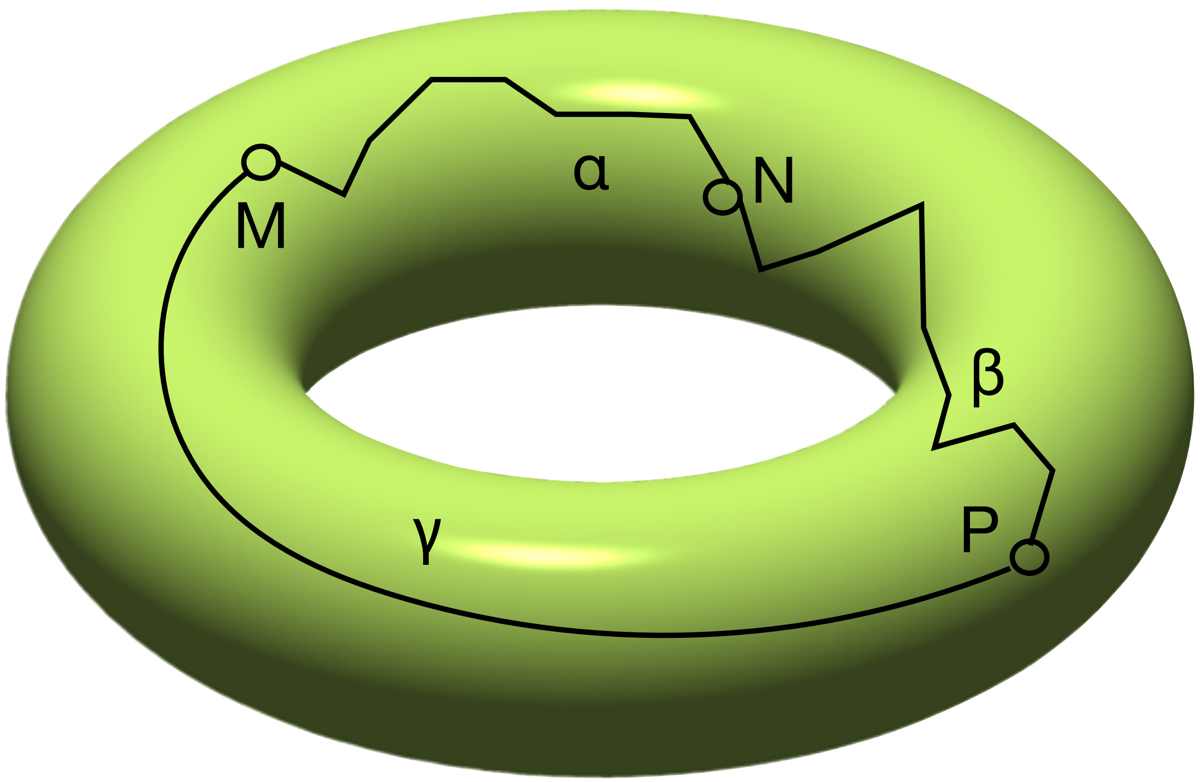
$\beta \circ \alpha : \mathsf{Path}\ M\ P$

$\gamma : \mathsf{Path}\ M\ P$

$(\beta \circ \alpha) \neq \gamma$

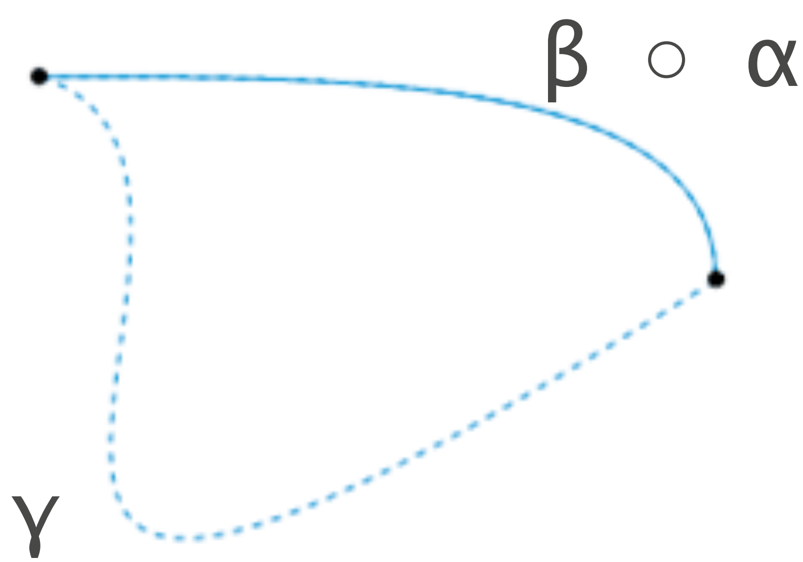
$\neg\ \mathsf{Path}_{\mathsf{Path}\ M\ P}\ (\beta \circ \alpha)\ \gamma$

But are *data*



$$\beta \circ \alpha : \mathsf{Path}\ M\ P$$

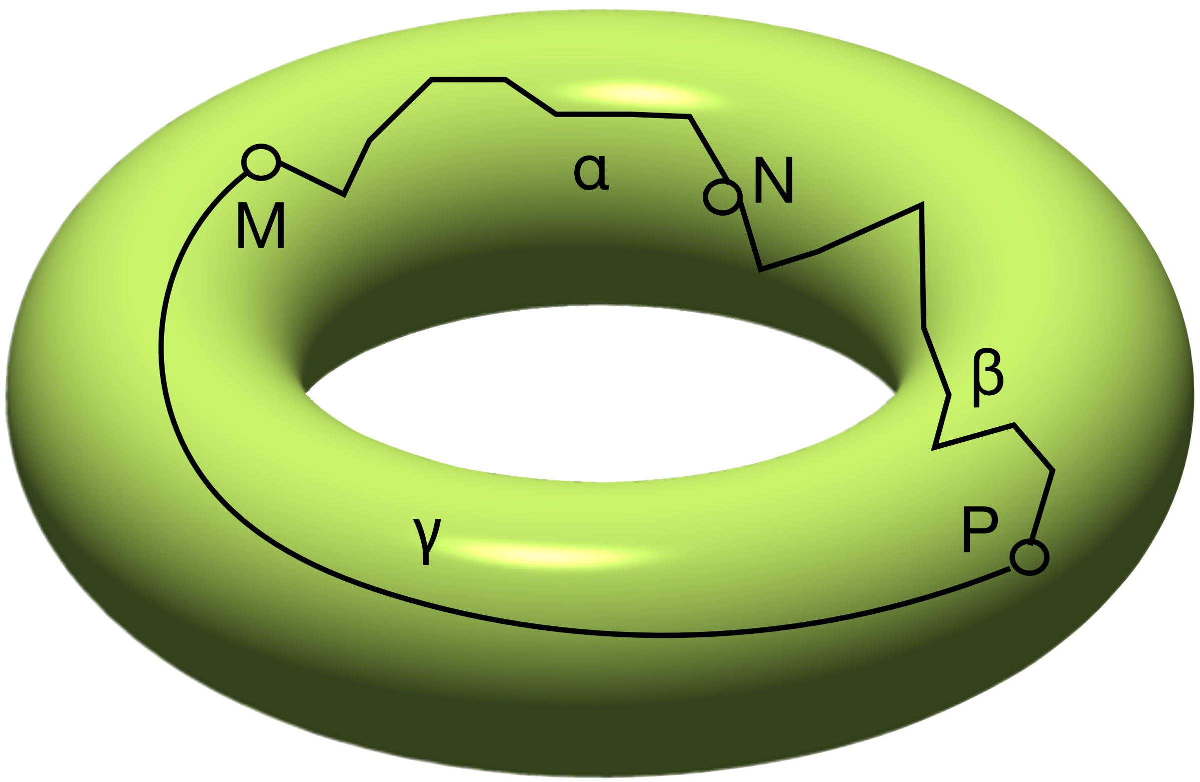
$$\gamma : \mathsf{Path}\ M\ P$$



$$(\beta \circ \alpha) \neq \gamma$$

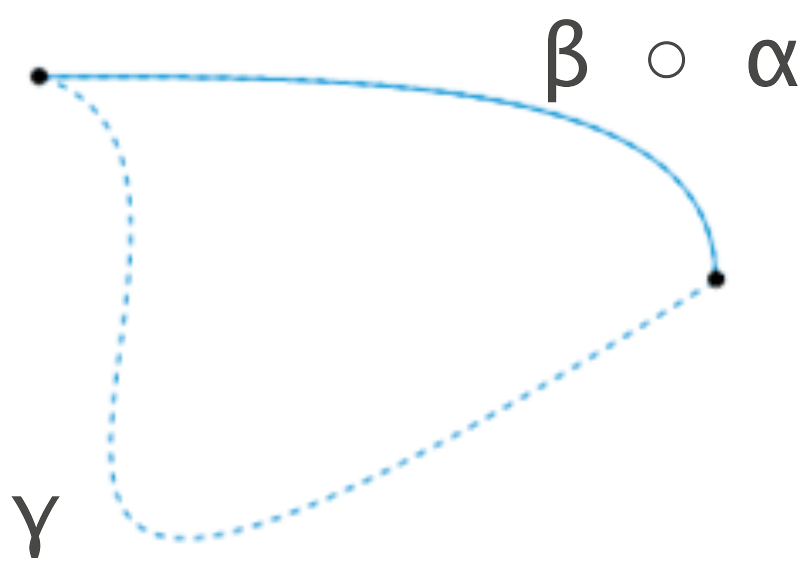
$$\neg\ \mathsf{Path}_{\mathsf{Path}\ M\ P}\ (\beta \circ \alpha)\ \gamma$$

But are *data*



$\beta \circ \alpha : \mathsf{Path}\ M\ P$

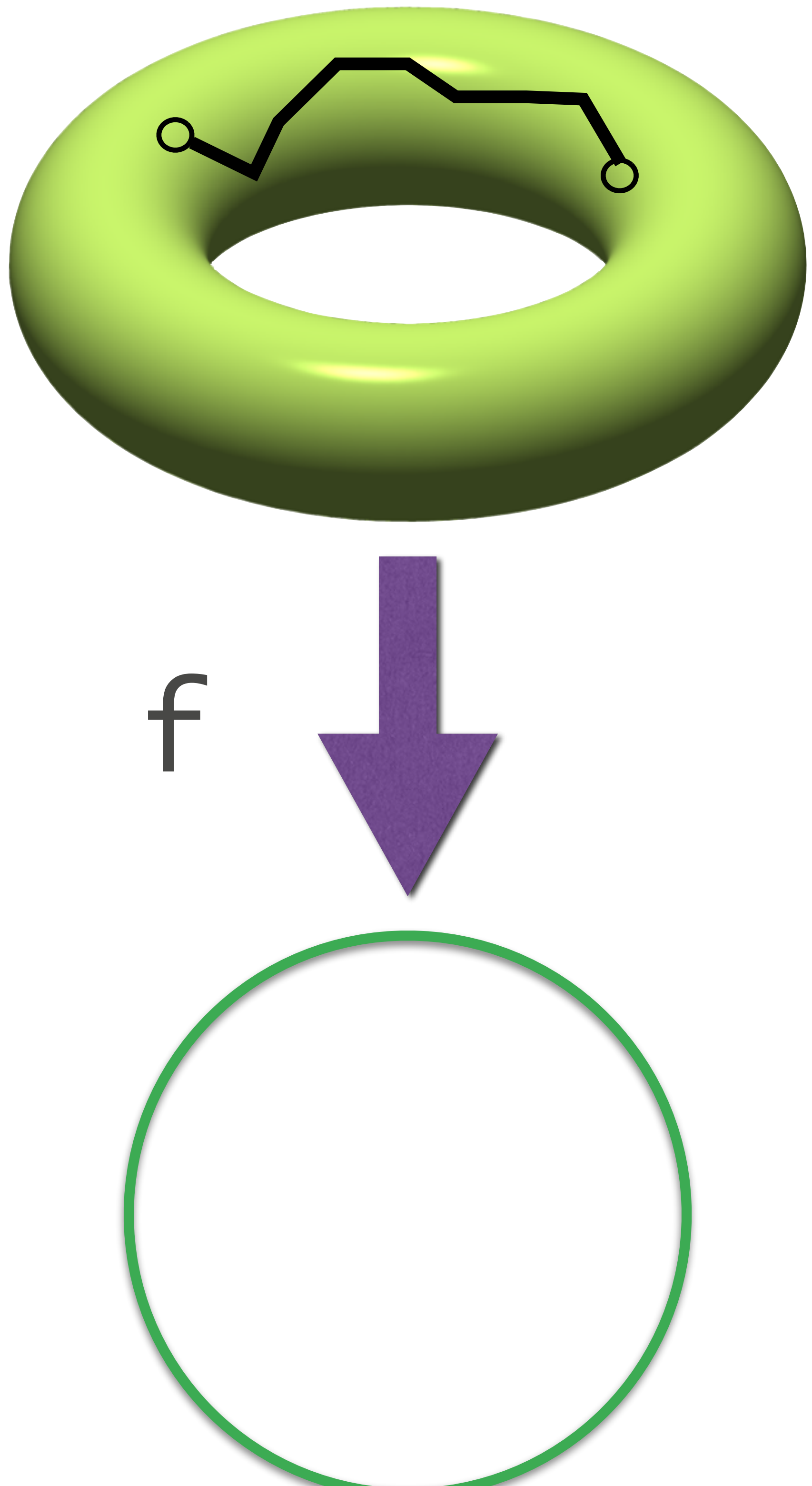
$\gamma : \mathsf{Path}\ M\ P$



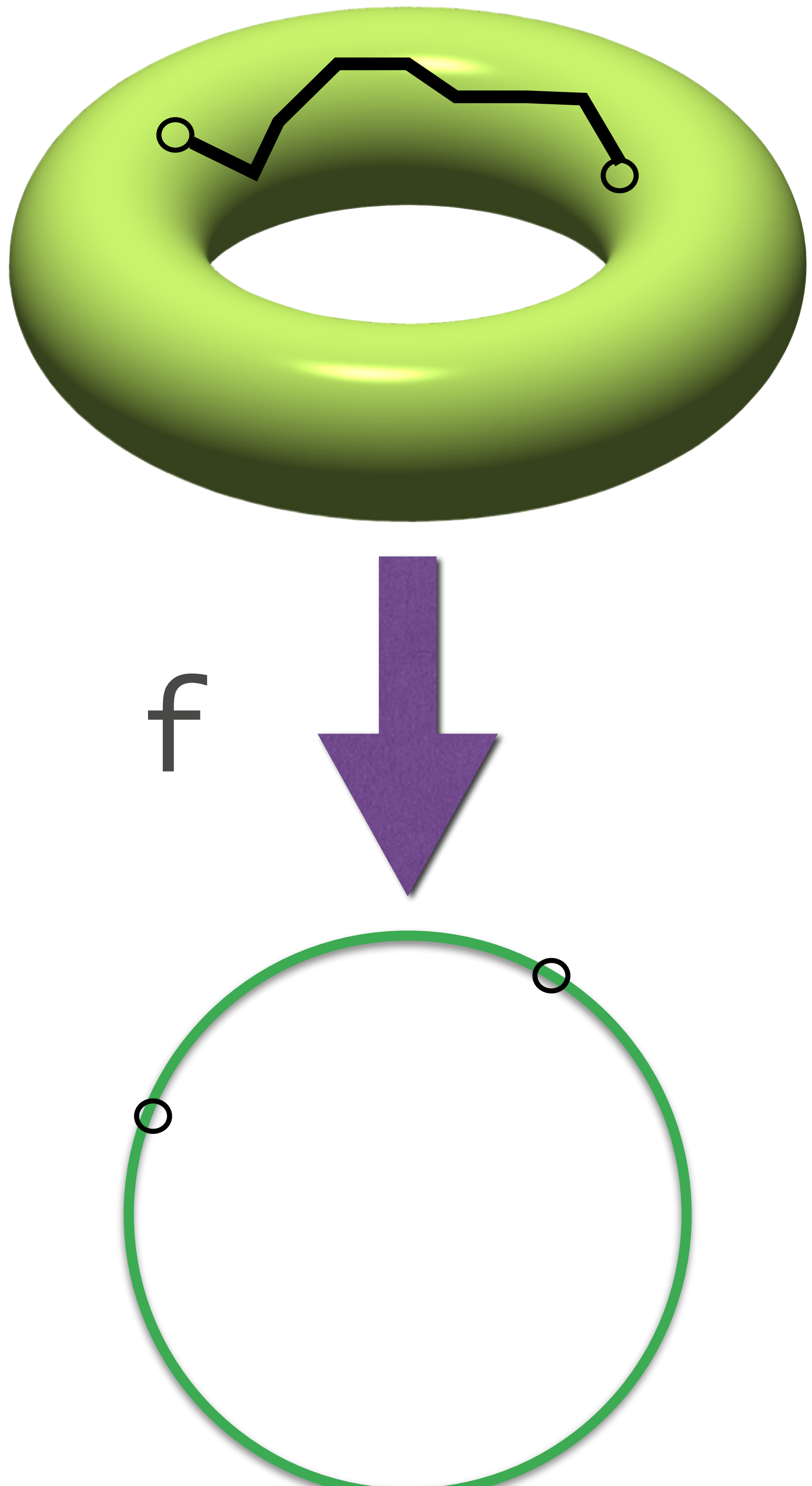
$(\beta \circ \alpha) \neq \gamma$

$\neg\ \mathsf{Path}_{\mathsf{Path}\ M\ P}\ (\beta \circ \alpha)\ \gamma$

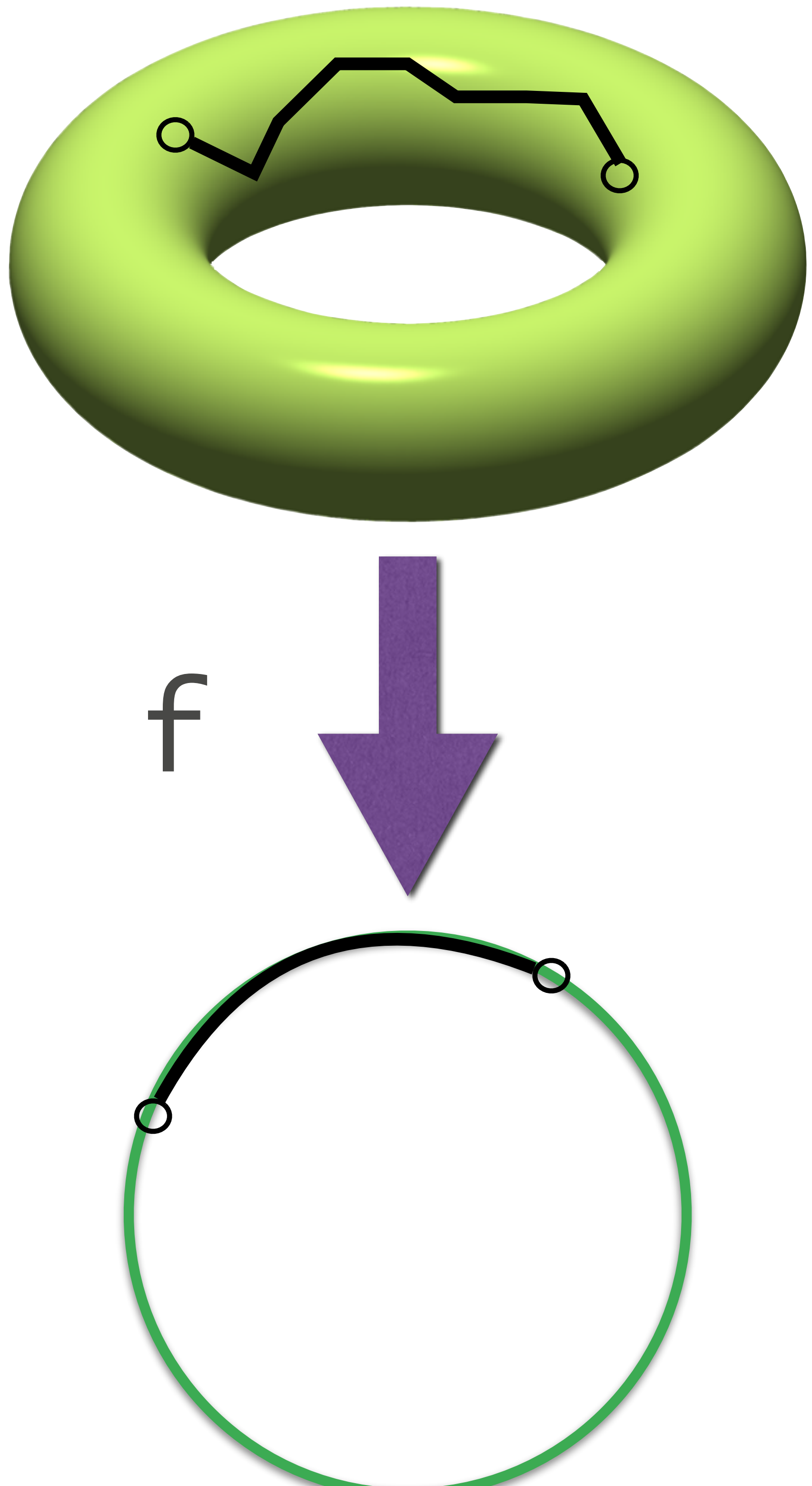
Functions “secretly” act on paths



Functions “secretly” act on paths

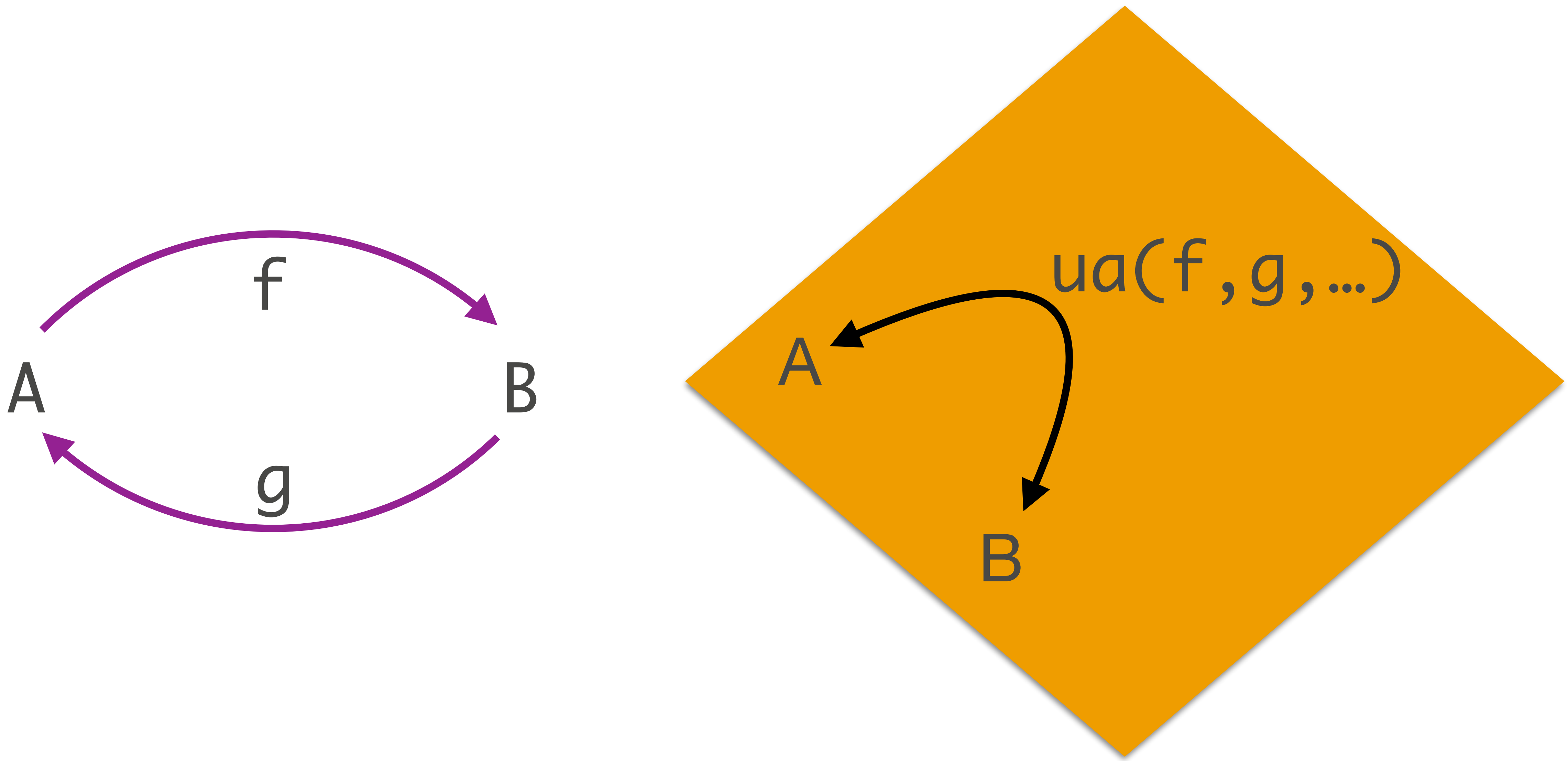


Functions “secretly” act on paths




```
Path x y
   →
Path f(x) f(y)
```

Voevodsky's univalence axiom



bijections induce paths between types

Monad interface (classic)

[Godemont,Moggi,Wadler]

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
```

Monad interface (classic)

[Godemont,Moggi,Wadler]

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

Applicative interface

[McBride,Patterson]

```
record Applicative (T : Type → Type) : Type where
  field
    pure      : ∀ {A} → A → T A
    _<*>_     : ∀ {A B} → T (A → B) → T A → T B
```

Applicative interface

[McBride,Patterson]

```
record Applicative (T : Type → Type) : Type where
  field
    pure      : ∀ {A} → A → T A
    _<*>_     : ∀ {A B} → T (A → B) → T A → T B
```

effects influence value but not structure

Applicative interface

[McBride,Patterson]

```
record Applicative (T : Type → Type) : Type where
  field
    pure        : ∀ {A} → A → T A
    _<*>_       : ∀ {A B} → T (A → B) → T A → T B
    pure-id     : ∀ {A} {c : T A} → pure (\ x → x) <*> c == c
    pure-comp   : ∀ {A B C} {f : T (A → B)} {g : T (B → C)} {c : T A}
                → ((pure _o_ <*> g) <*> f) <*> c == g <*> (f <*> c)
    apply-pure : ∀ {A B} {f : A → B} {a : A}
                → pure f <*> pure a == pure (f a)
    apply-to-pure : ∀ {A B} {f : T (A → B)} {a : A}
                  → f <*> (pure a) == pure (λ f₁ → f₁ a) <*> f
```

effects influence value but not structure

Monad interface (new)

```
record App⇒Monad (T : Type → Type) : Type where

    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

Monad interface (new)

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
     return-pure : ∀ {A} {a : A} → pure a == return a
     <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                 → f <*> a == ( f >>= λ f' →
                                a >>= λ a' →
                                return (f' a'))
```

```
 instance Monad Maybe where
          return a = Some a
        None >>= k = None
    (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
 instance Monad Maybe where
          return a = Some a
        None >>= k = None
    (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
 instance App⇒Monad Maybe where

          return a = Some a
        None >>= k = None
    (Some x) >>= k = k x
           pure a = return a
         f <*> a = f >>= λf’ → a >>= λ a’ → return (f’ a’)

sequenceM : ∀ {T A} {App⇒Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
instance Monad Maybe where
        return a = Some a
      None >>= k = None
  (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
instance App⇒Monad Maybe where

        return a = Some a
      None >>= k = None
  (Some x) >>= k = k x
         pure a = return a
        f <*> a = f >>= λf’ → a >>= λ a’ → return (f’ a’)

sequenceM : ∀ {T A} {App⇒Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
instance Monad Maybe where
         return a = Some a
       None >>= k = None
   (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
instance App⇒Monad Maybe where

         return a = Some a
       None >>= k = None
   (Some x) >>= k = k x
          pure a = return a
         f <*> a = f >>= λf’ → a >>= λ a’ → return (f’ a’)

sequenceM : ∀ {T A} {App⇒Monad T} → List (T A) → T(List A)

sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

Instance of classic → instance of new

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
    return-pure : ∀ {A} {a : A} → pure a == return a
    <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                → f <*> a == ( f >>= λ f' →
                               a >>= λ a' →
                               return (f' a'))
```

Instance of new → instance of classic

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
    return-pure : ∀ {A} {a : A} → pure a == return a
    <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                → f <*> a == ( f >>= λ f' →
                               a >>= λ a' →
                               return (f' a'))
```

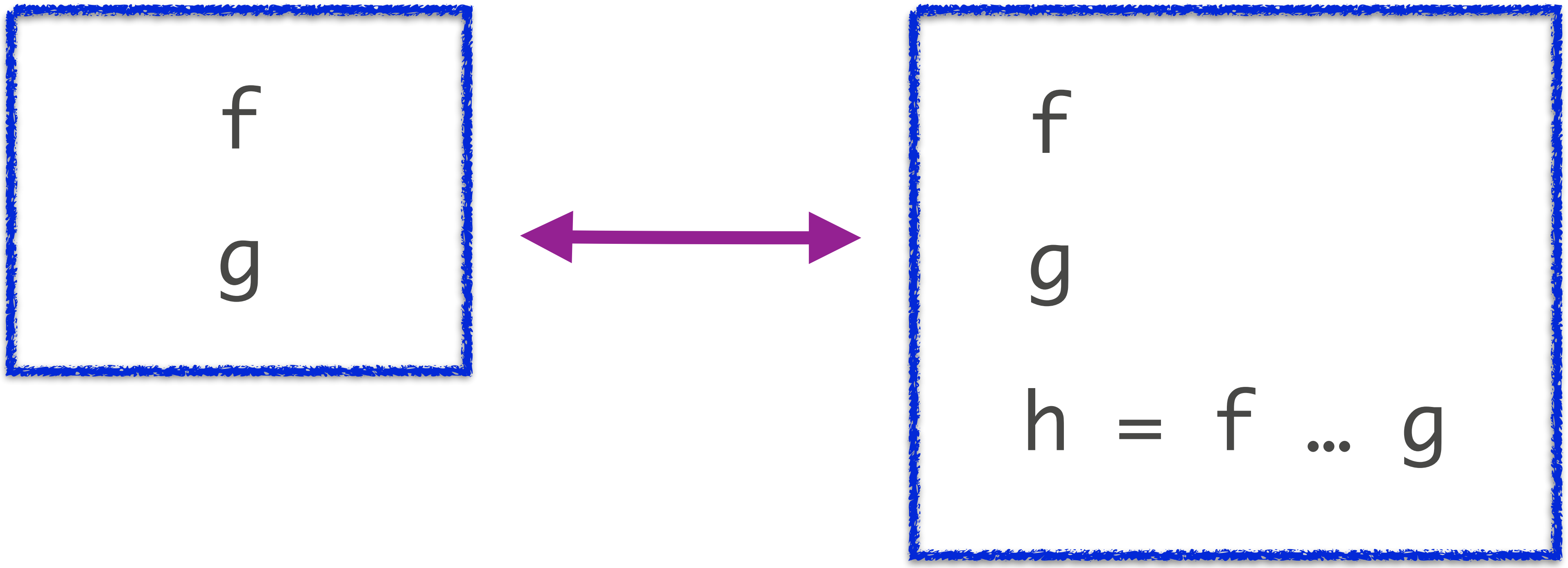
f

g

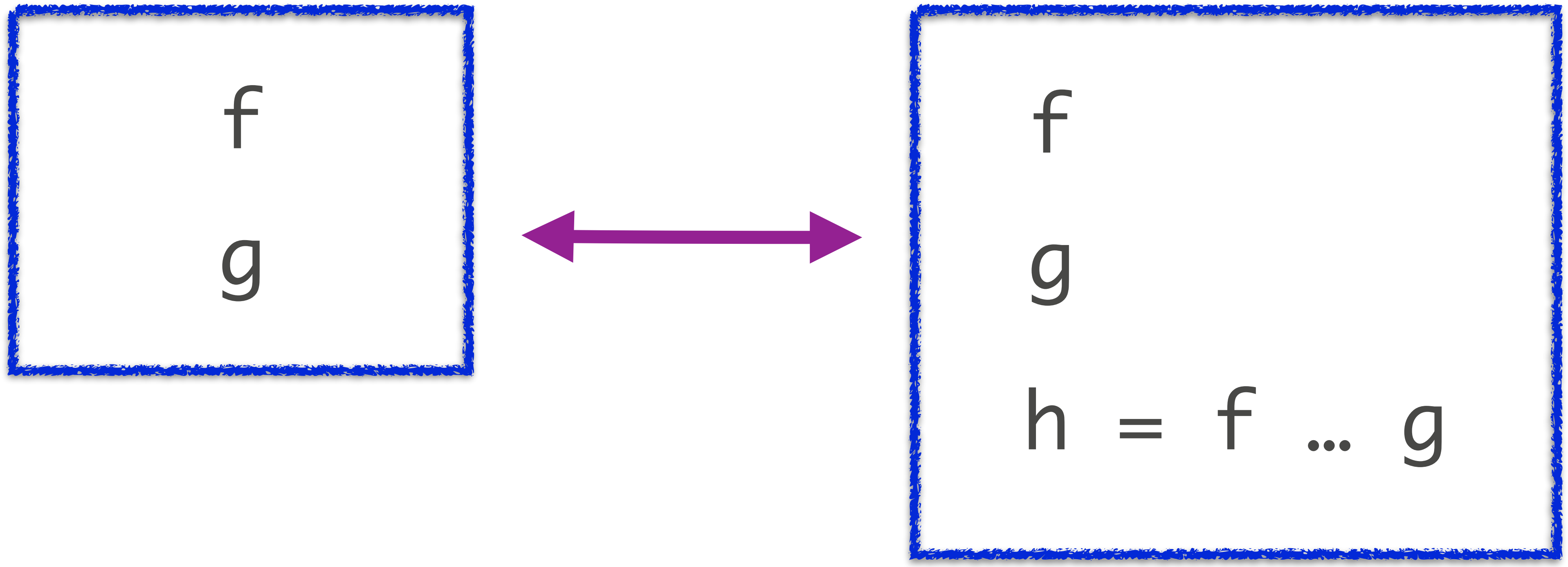
```
f
g
```

```
f
g
h = f … g
```

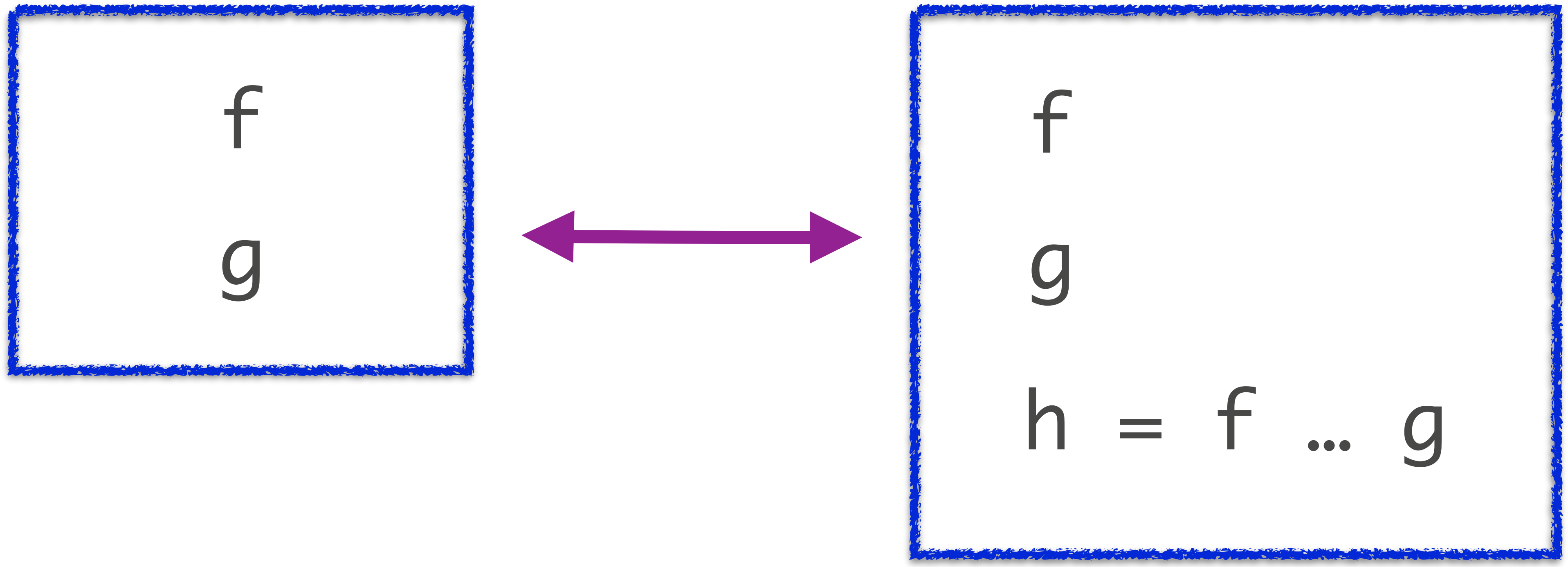
- extend interface with an operation that is determined by the others (convenience, efficiency)



- extend interface with an operation that is determined by the others (convenience, efficiency)
- the (default implementation, forget)-bijection can be used to *dynamically* convert between them

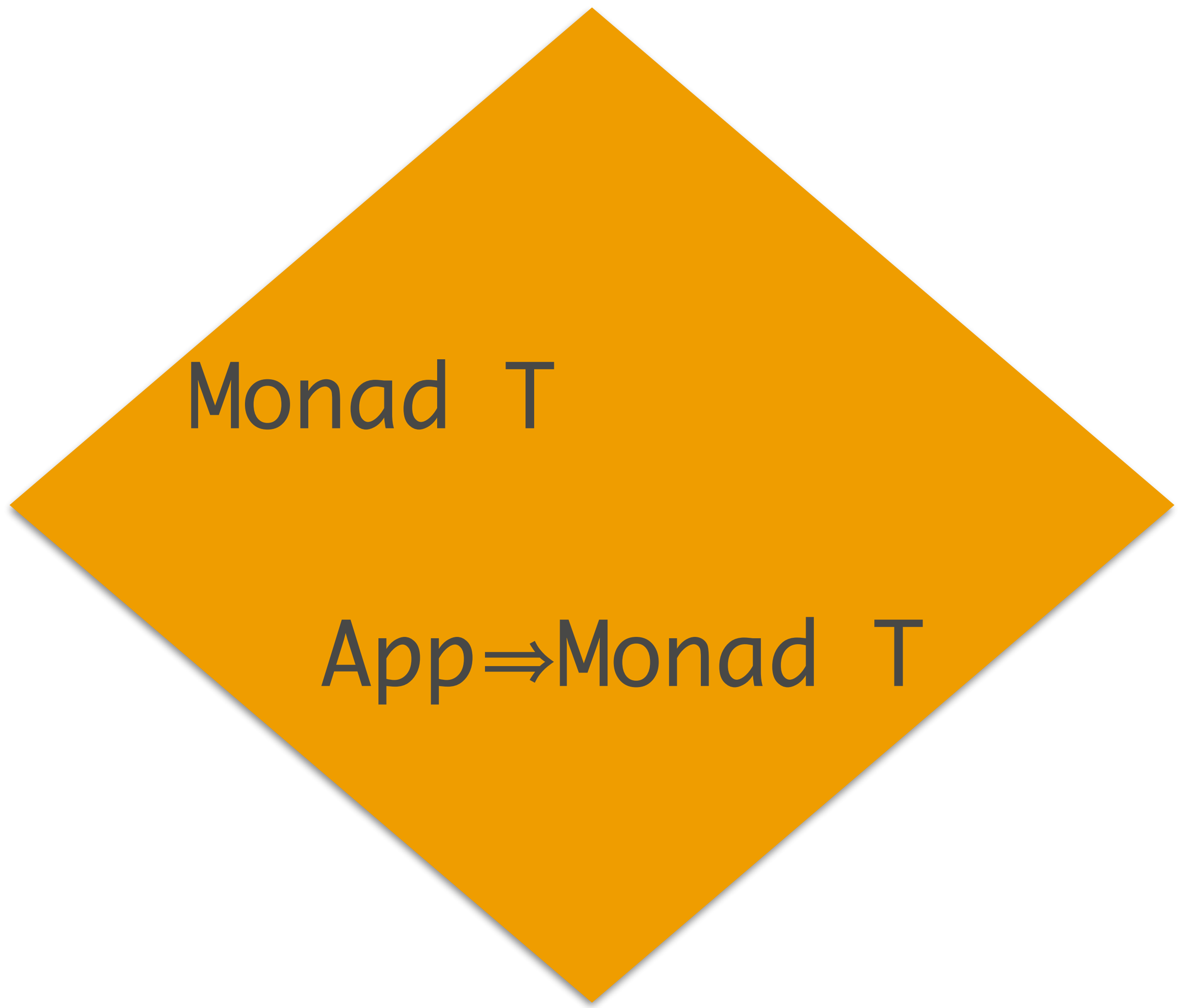


- extend interface with an operation that is determined by the others (convenience, efficiency)
- the (default implementation, forget)-bijection can be used to *dynamically* convert between them
- it’s “obvious” how to apply this in context

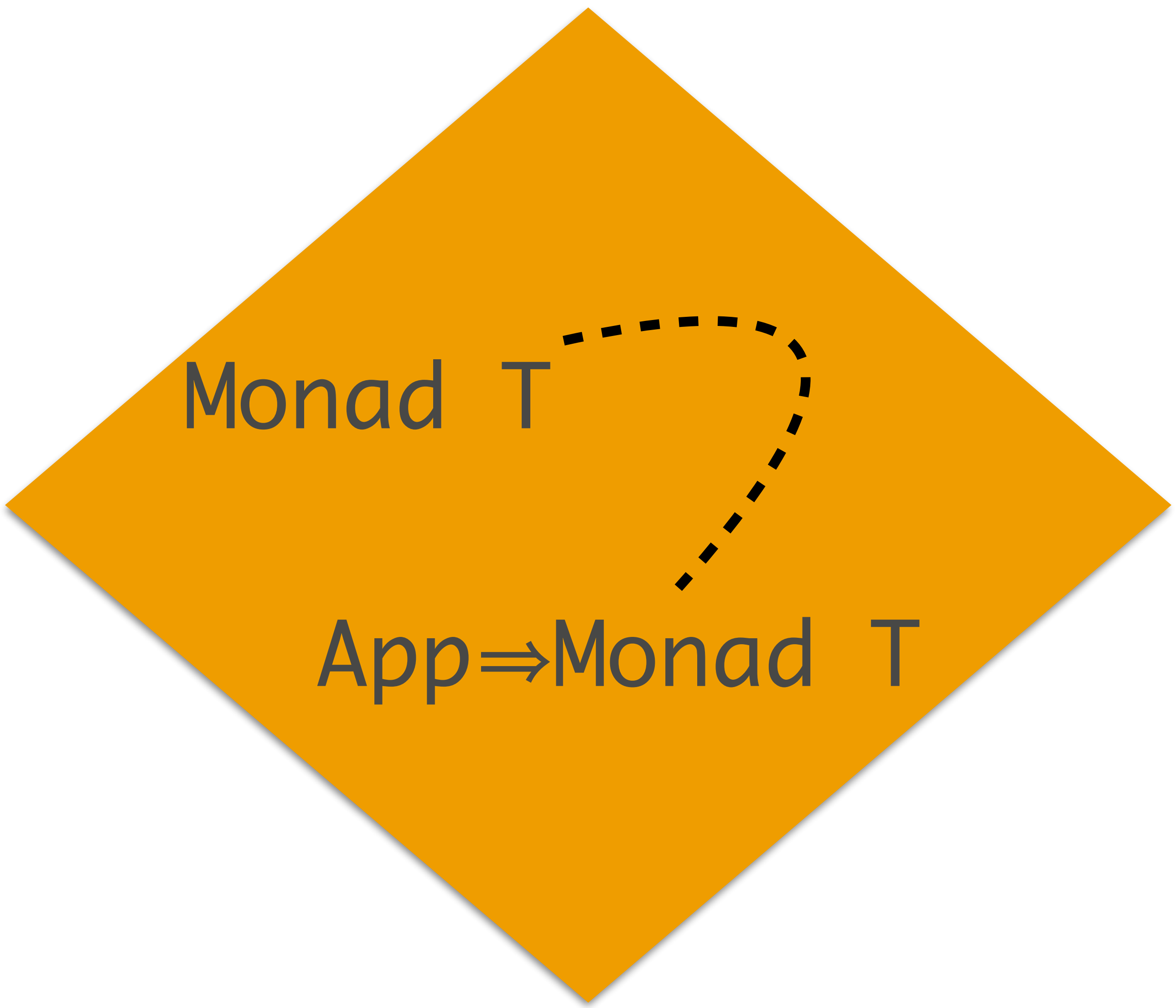


- extend interface with an operation that is determined by the others (convenience, efficiency)
- the (default implementation, forget)-bijection can be used to *dynamically* convert between them
- it's "obvious" how to apply this in context
- partially evaluate to modify source code

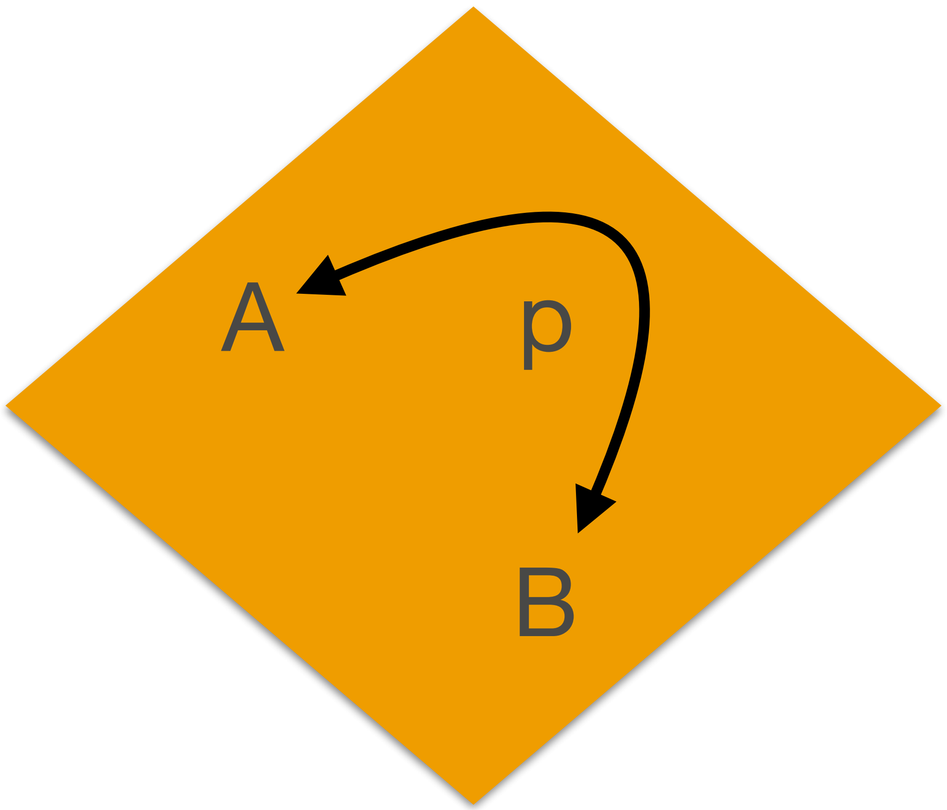
Paths between types



Paths between types

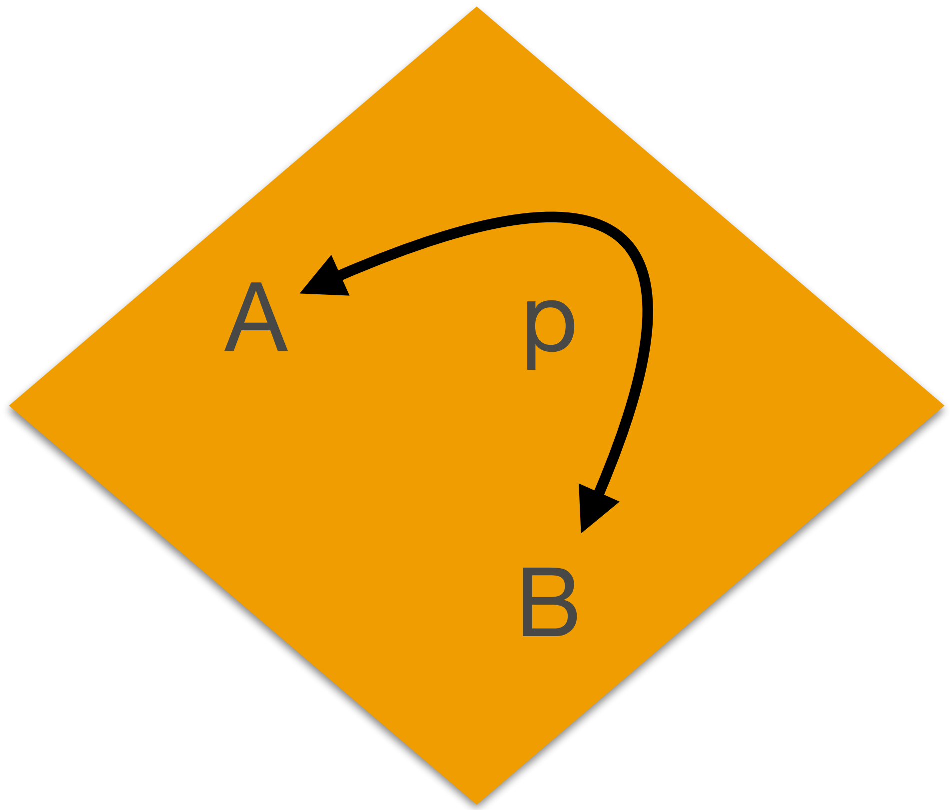


Path-related types do **not** have same elements

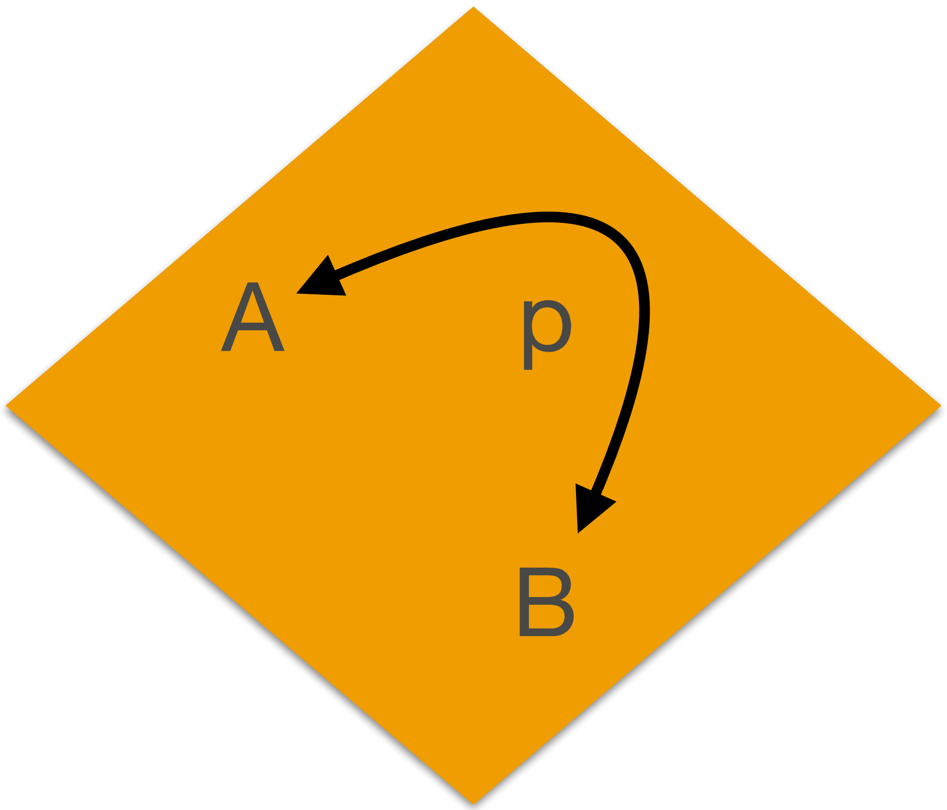


Path-related types do **not** have same elements

but paths between types induce **bijections**

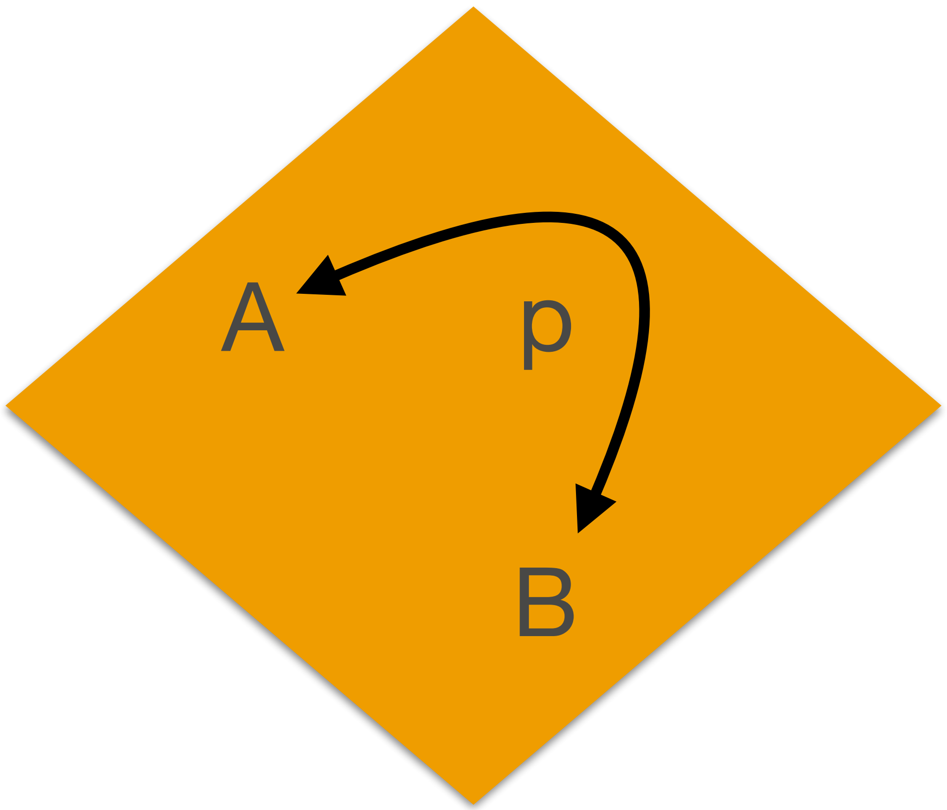


Path-related types do **not** have same elements but paths between types induce **bijections**



`coe p : A → B`

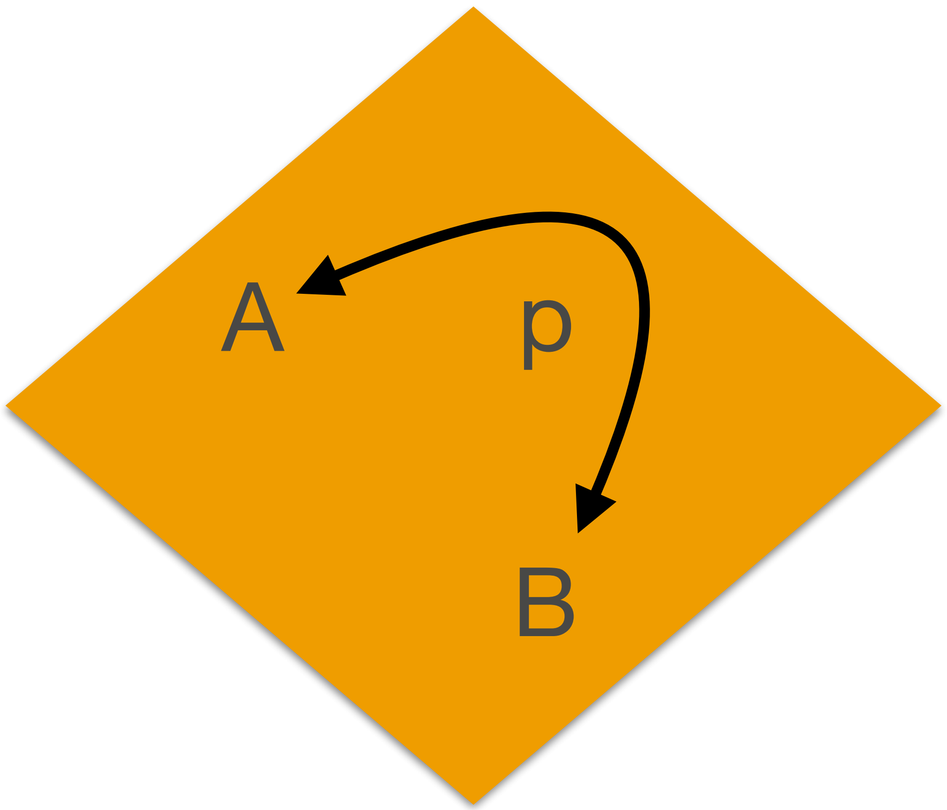
Path-related types do **not** have same elements but paths between types induce **bijections**



$\texttt{coe p : A} \to \texttt{B}$

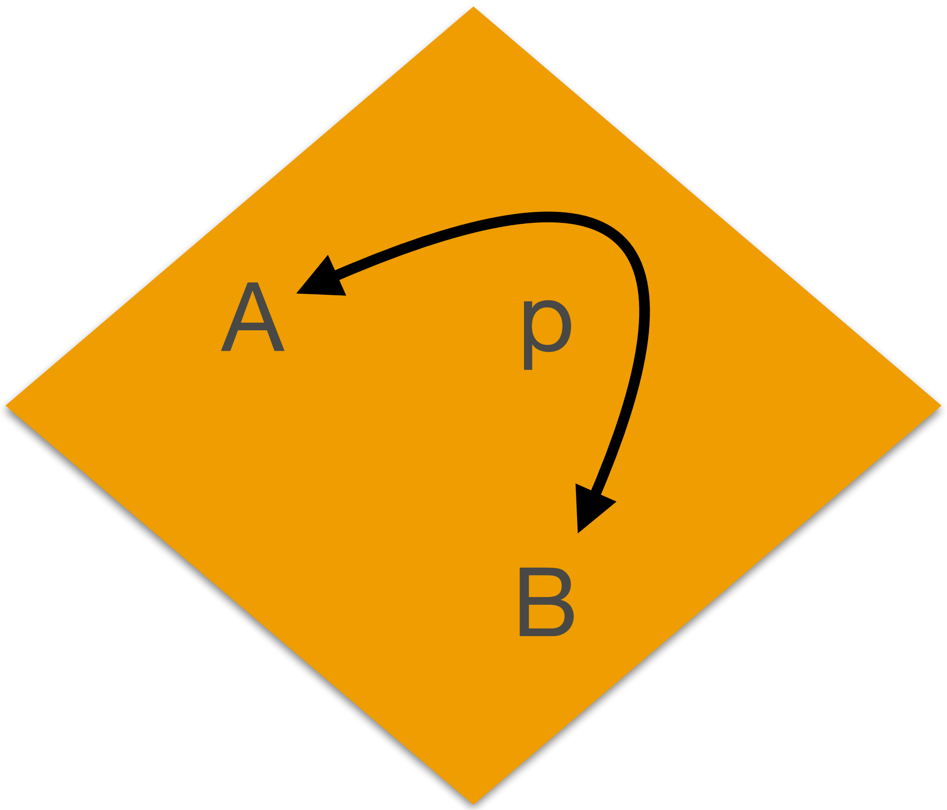
$\texttt{coe p}^{-1}\ \texttt{: B} \to \texttt{A}$

Path-related types do **not** have same elements but paths between types induce **bijections**




```
coe p : A → B

coe p⁻¹ : B → A

(mutually inverse)
```

Path-related types do **not** have same elements but paths between types induce **bijections**



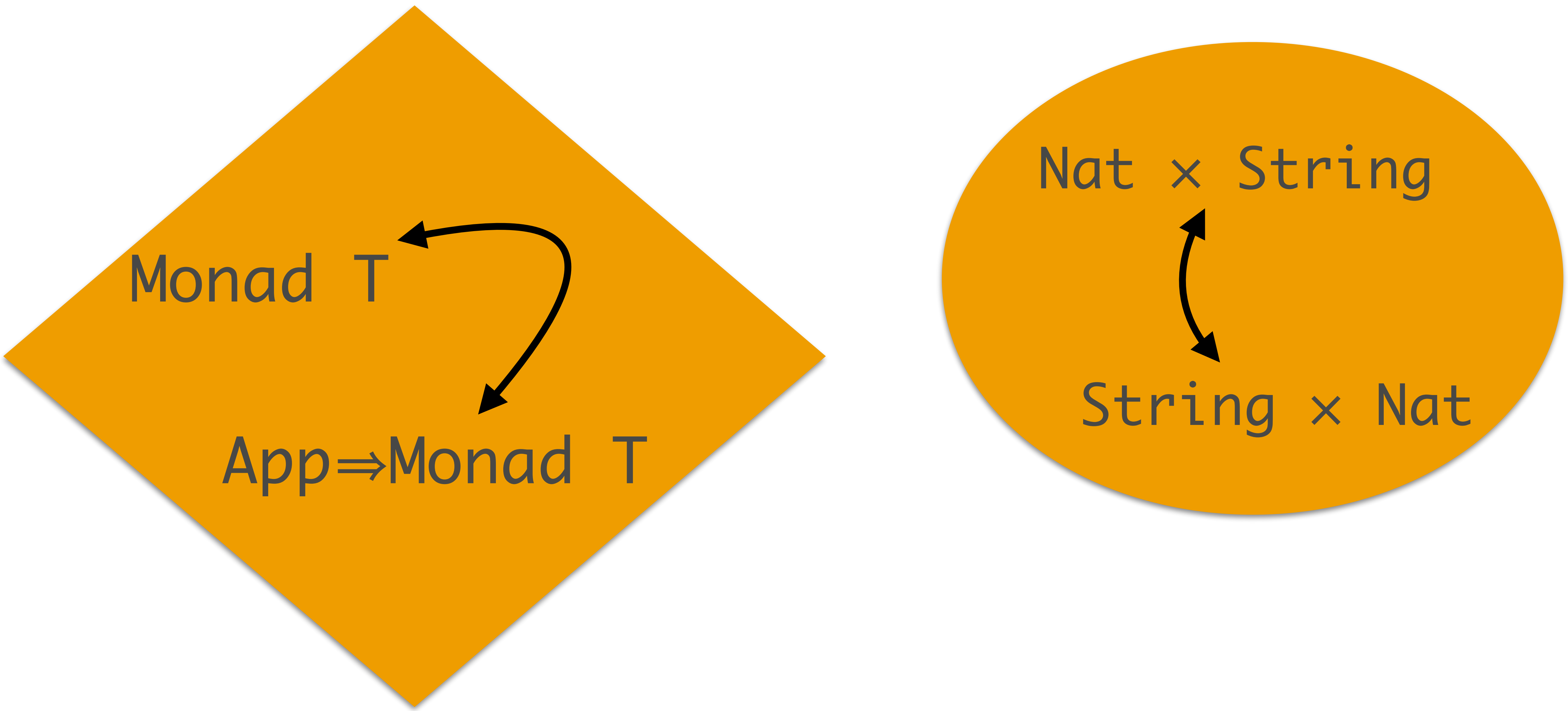
coe p : A → B

coe p^{-1} : B → A

(mutually inverse)

moving along a path might do some work

Voevodsky's univalence axiom

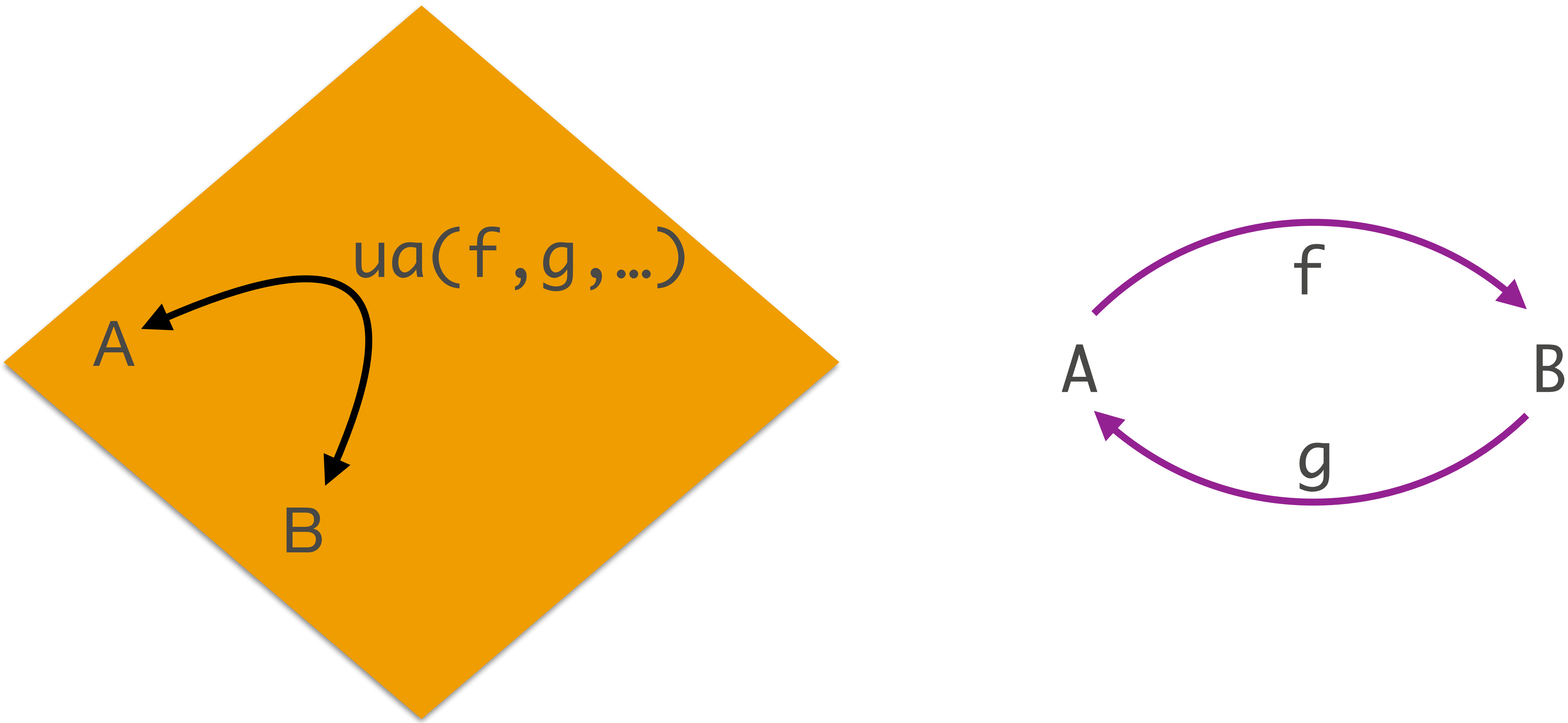


bijections induce paths between types*

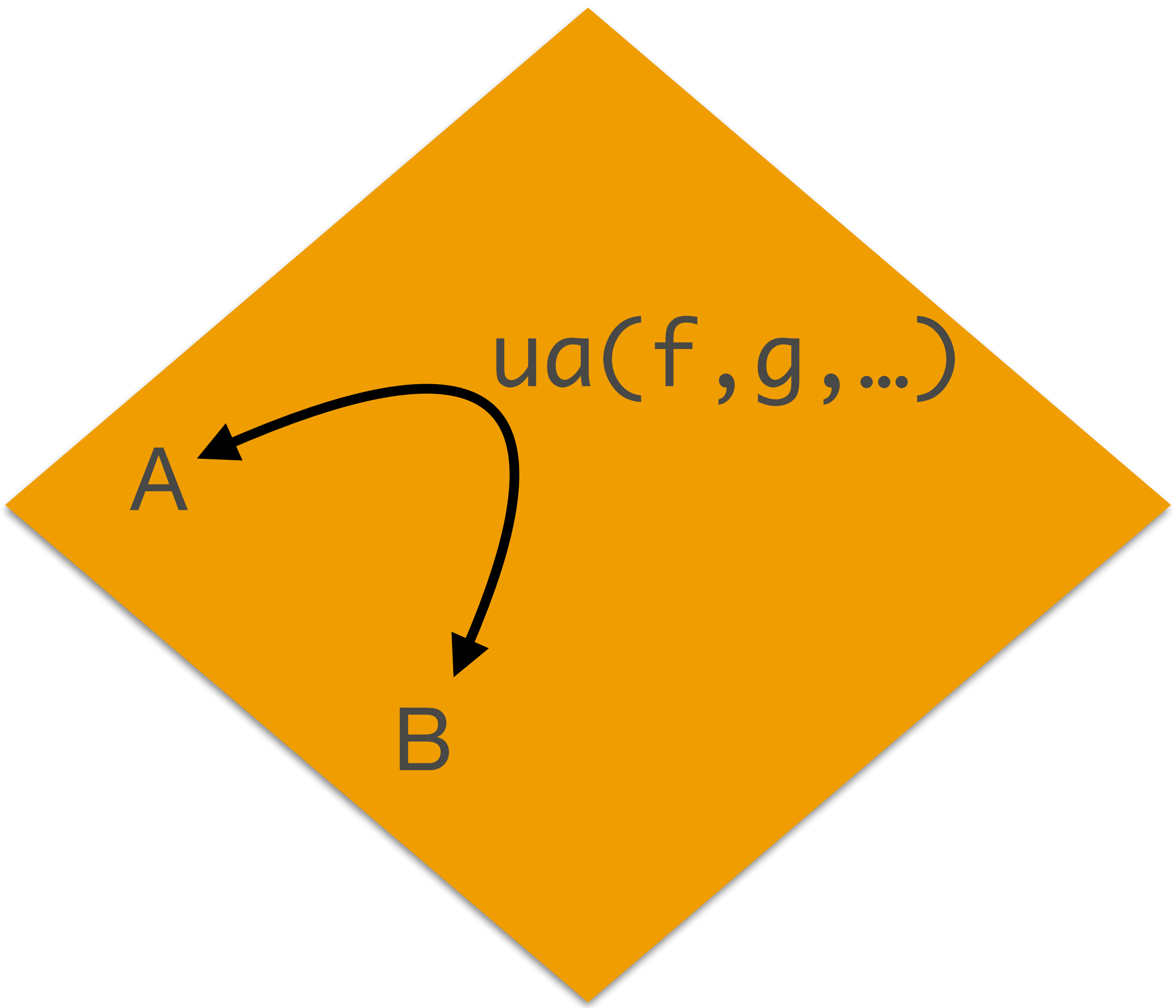
Coercing along univalence

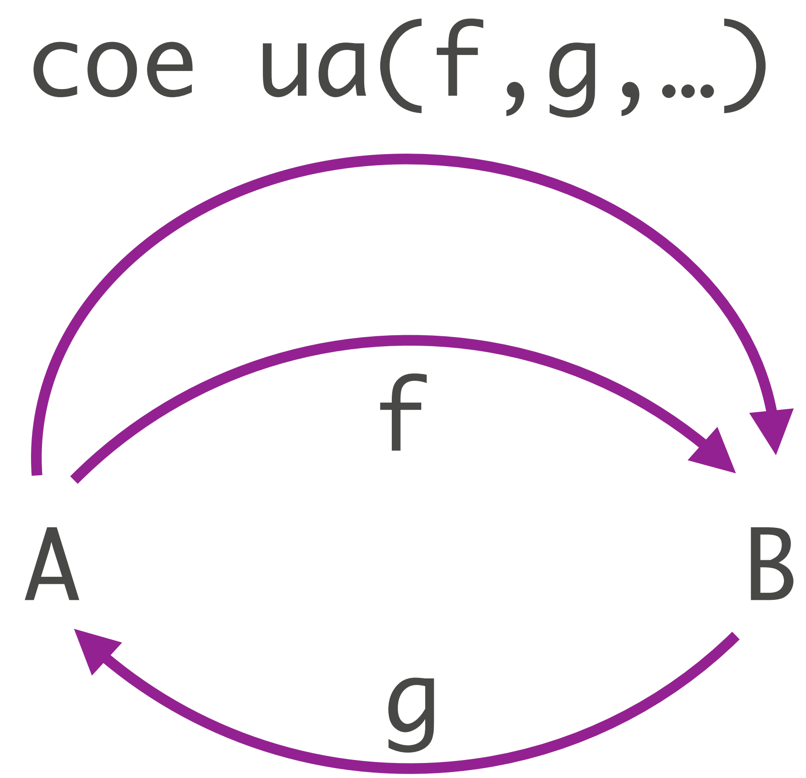
$$\begin{tikzcd} A \arrow[r, bend left, "f"] & B \arrow[l, bend left, "g"] \end{tikzcd}$$

Coercing along univalence

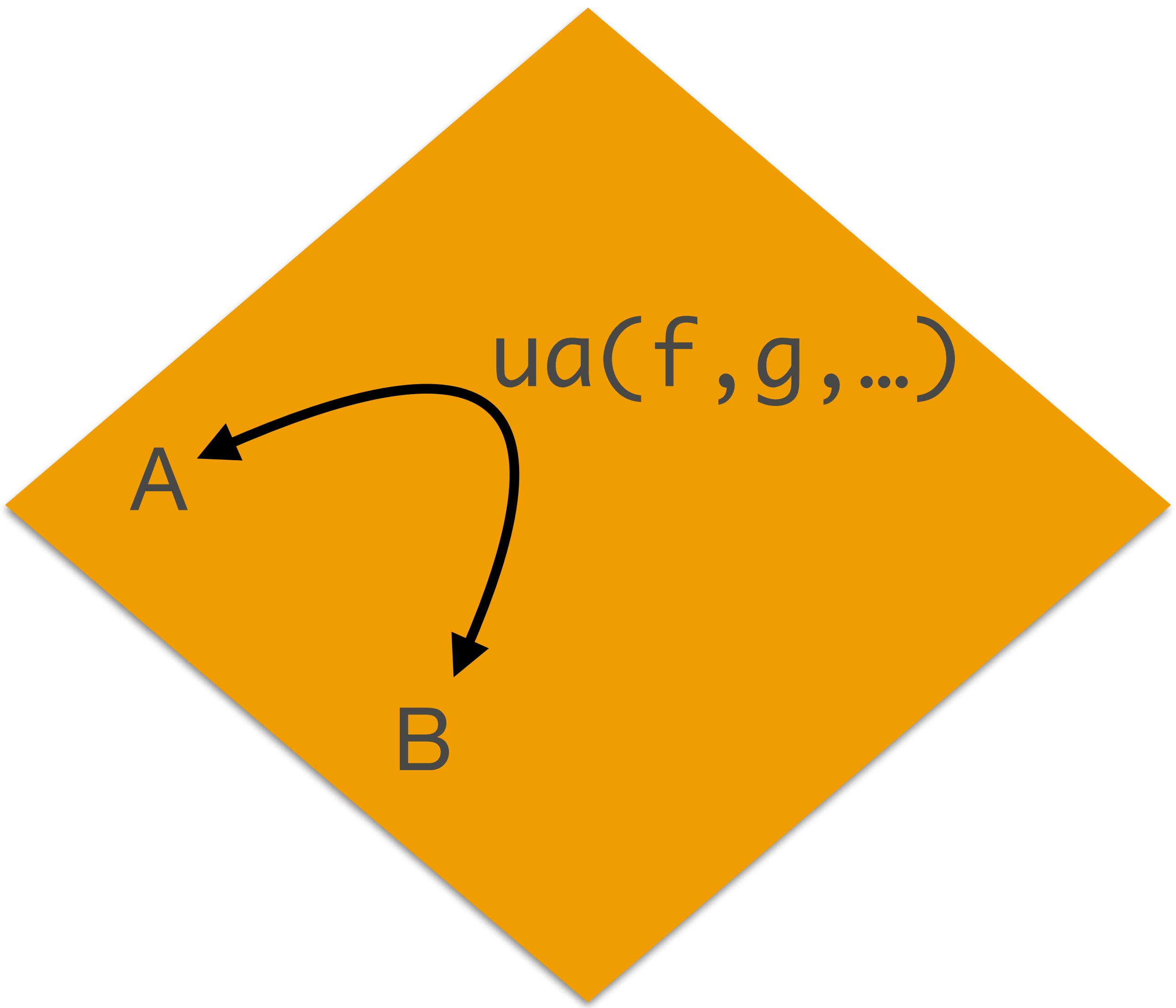


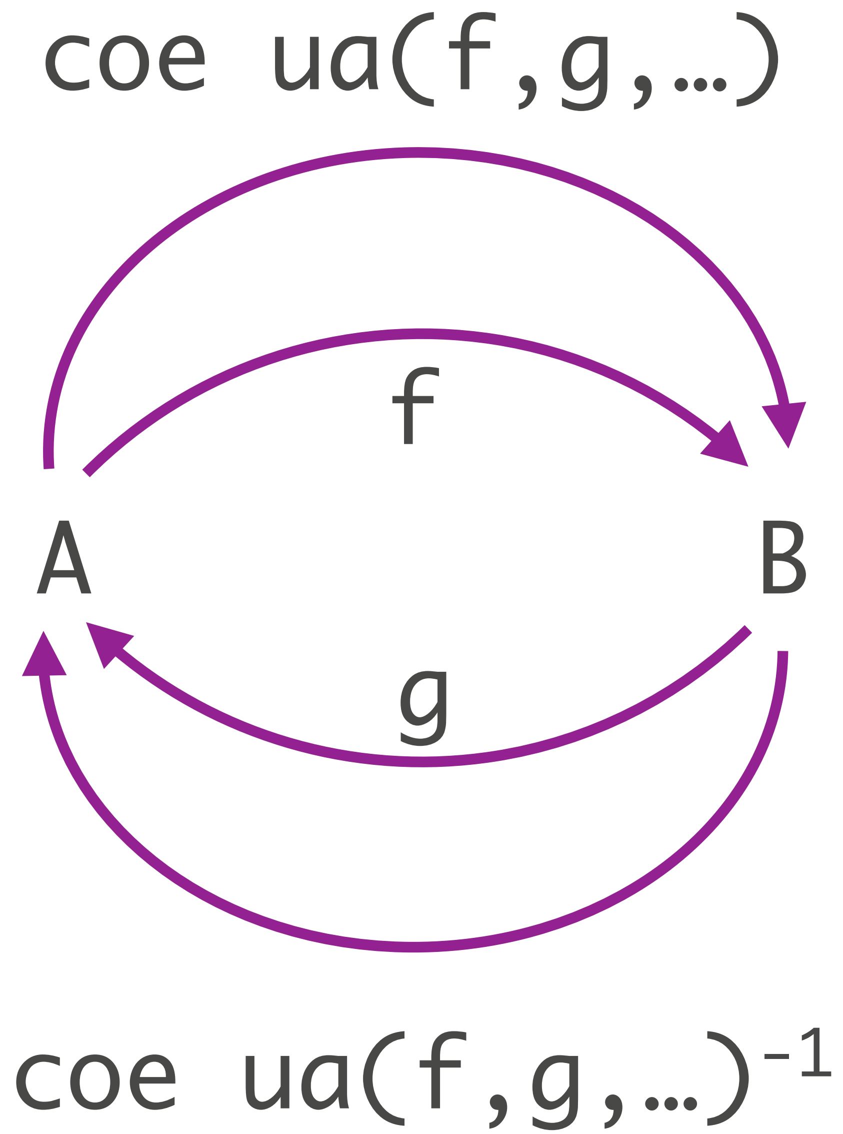
Coercing along univalence



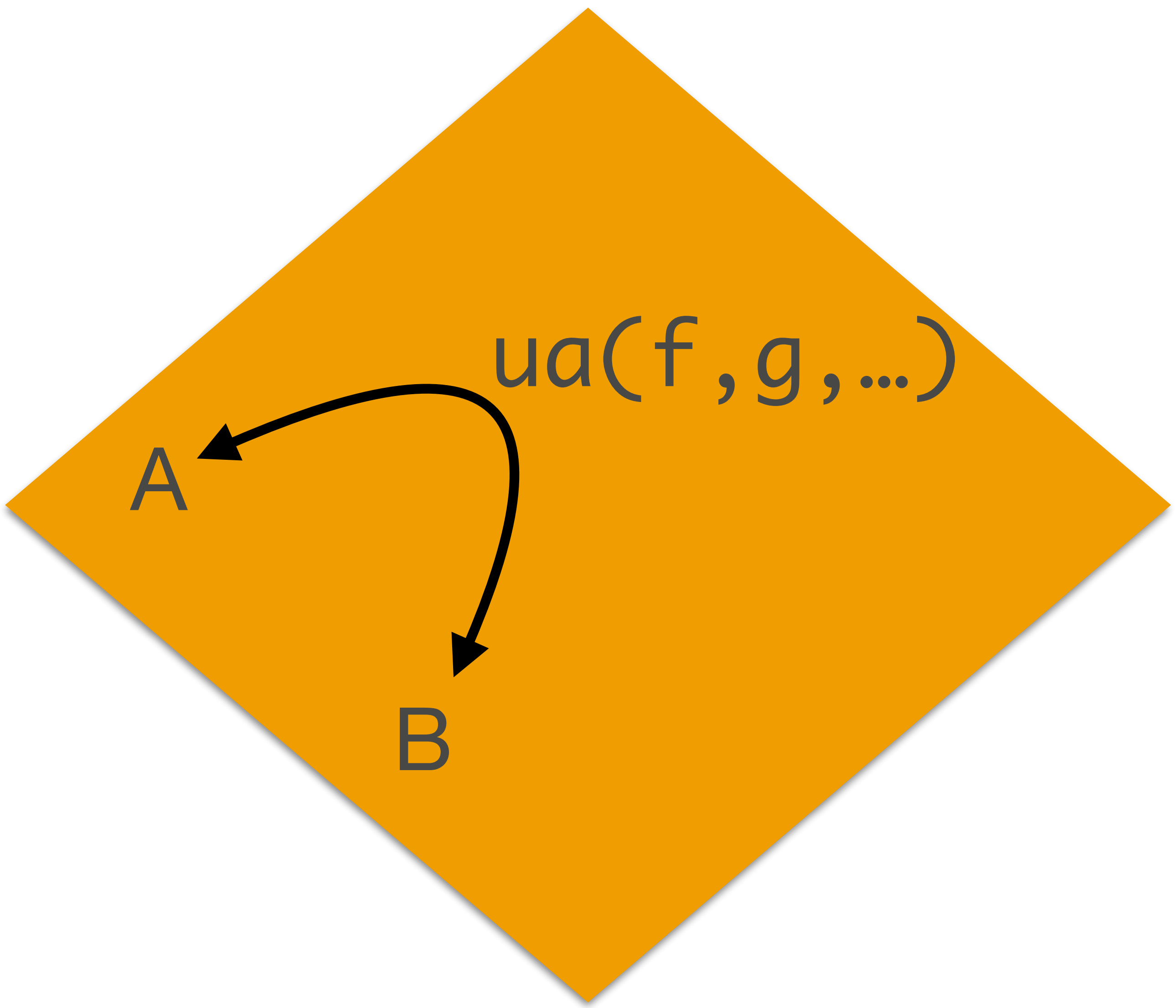


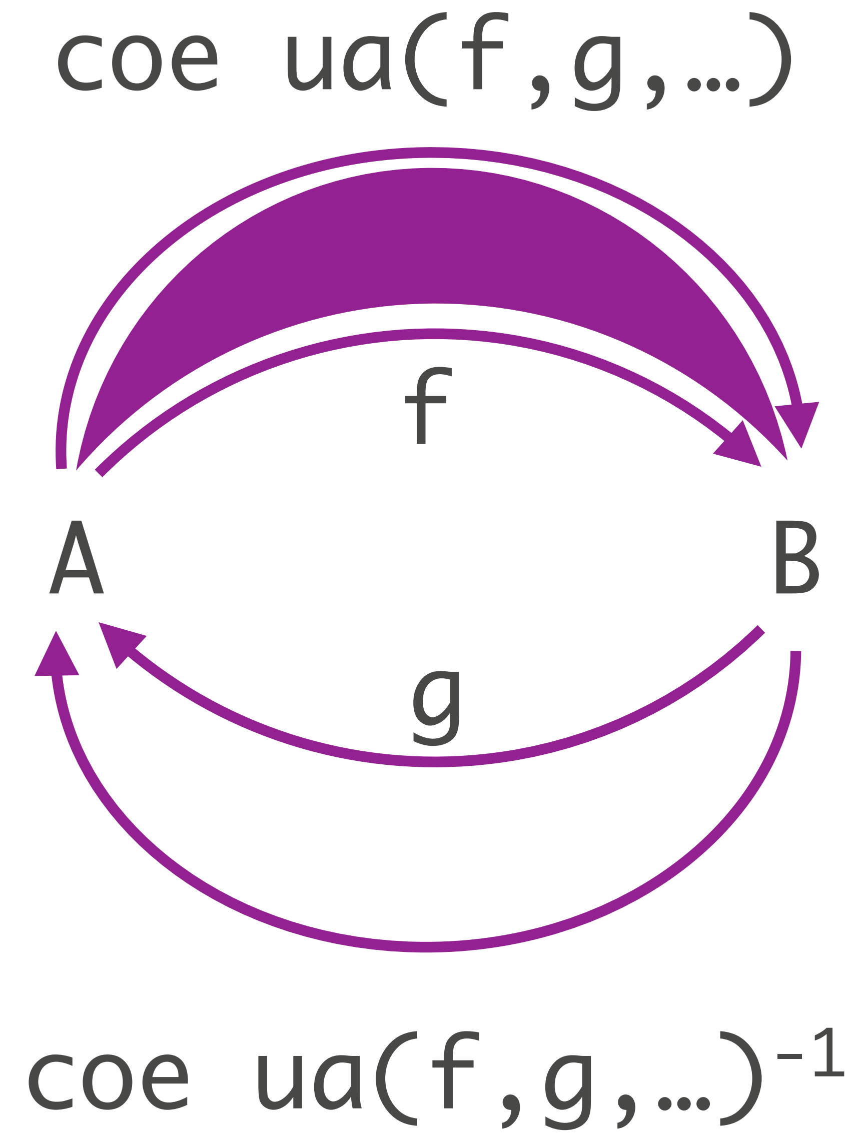
Coercing along univalence



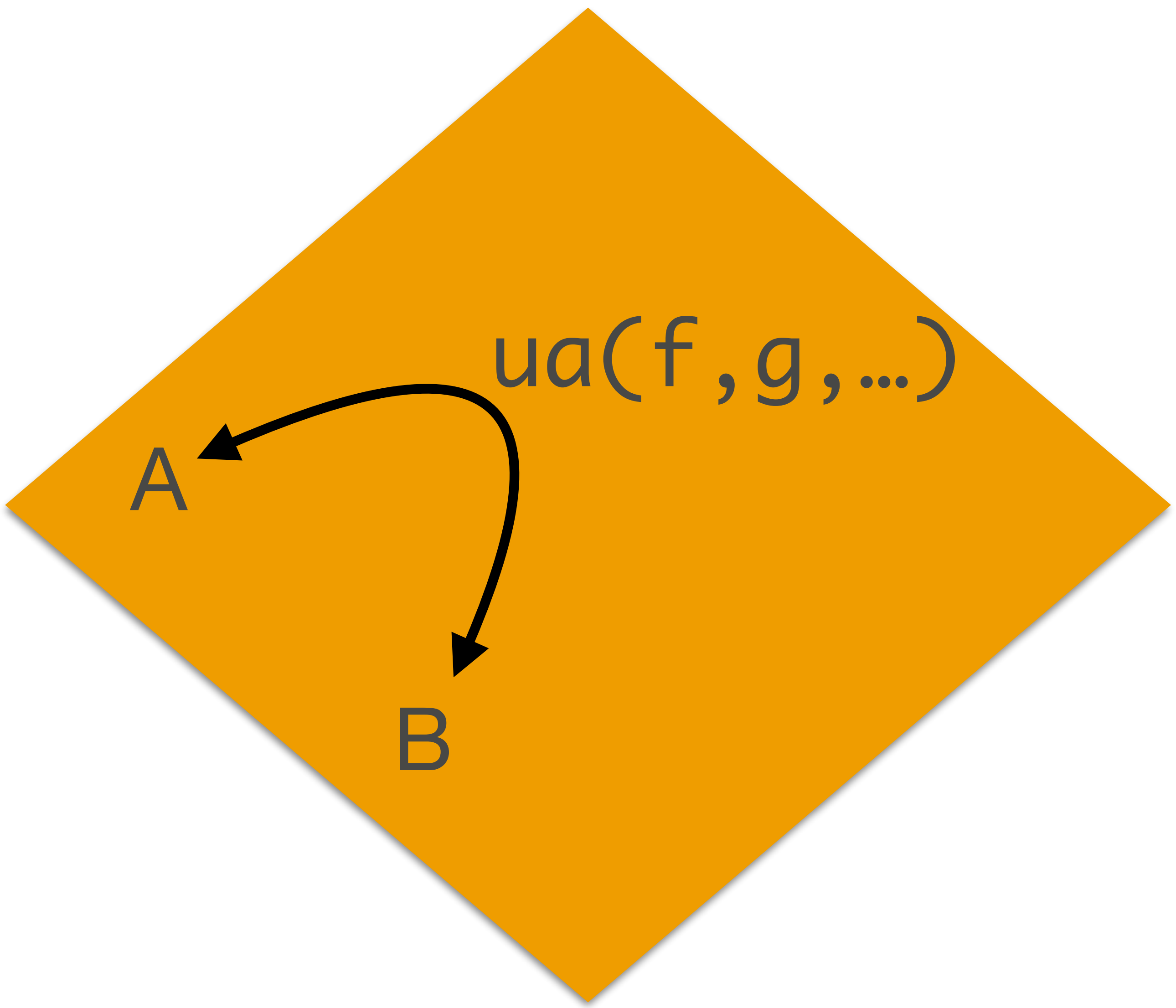


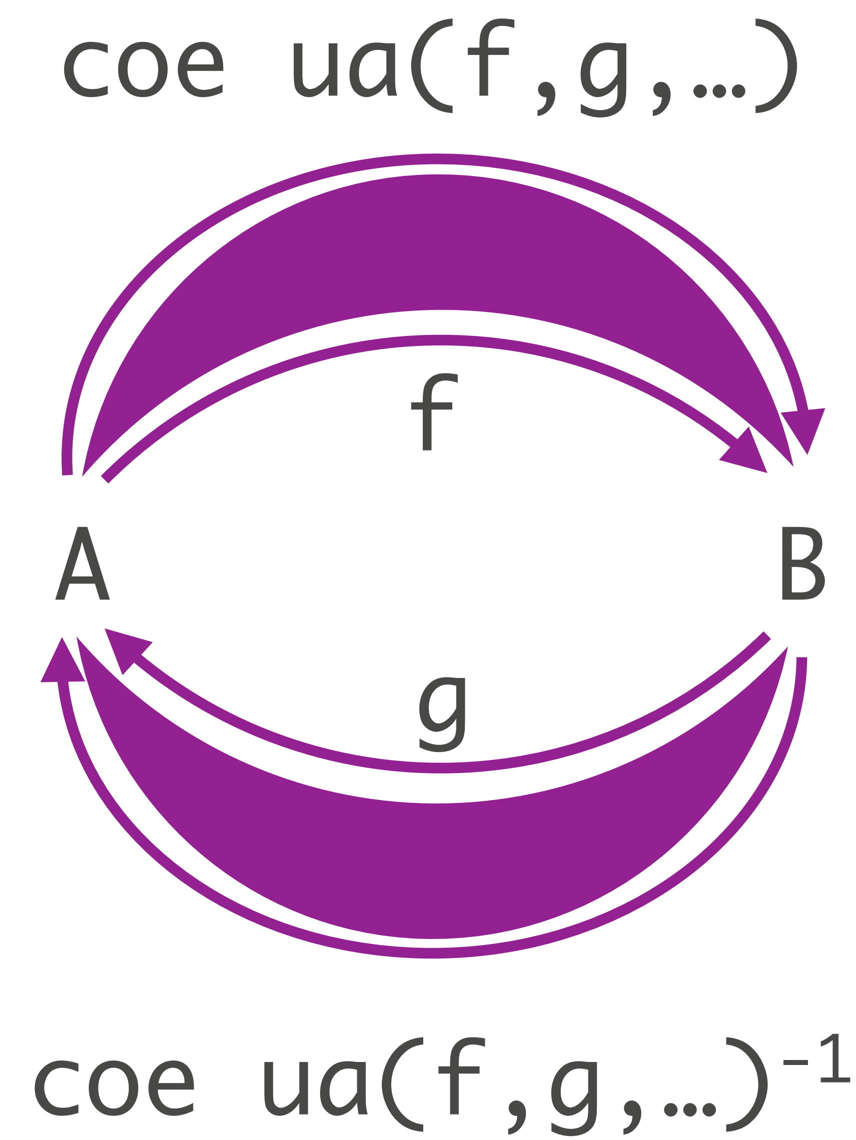
Coercing along univalence





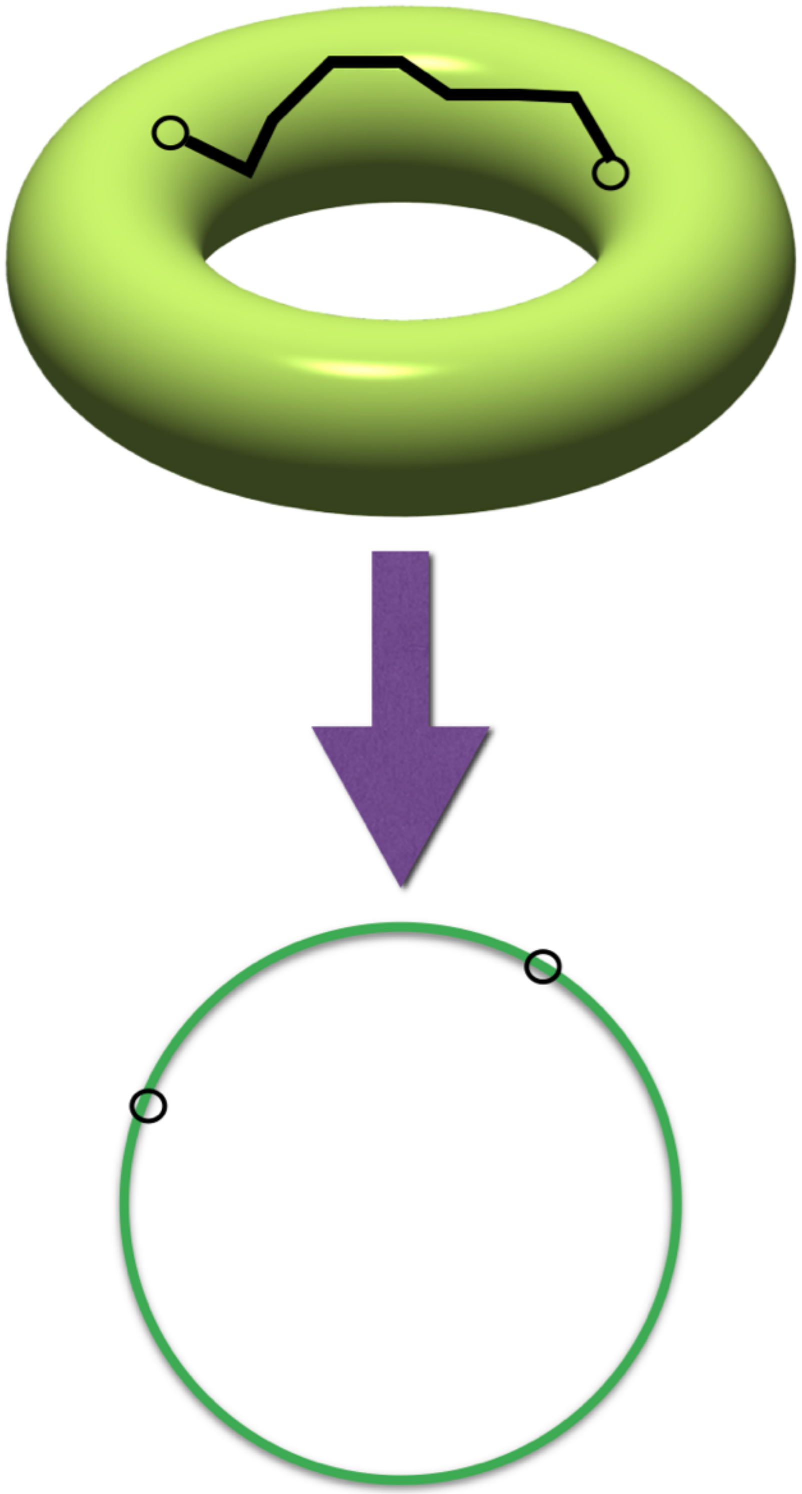
Coercing along univalence



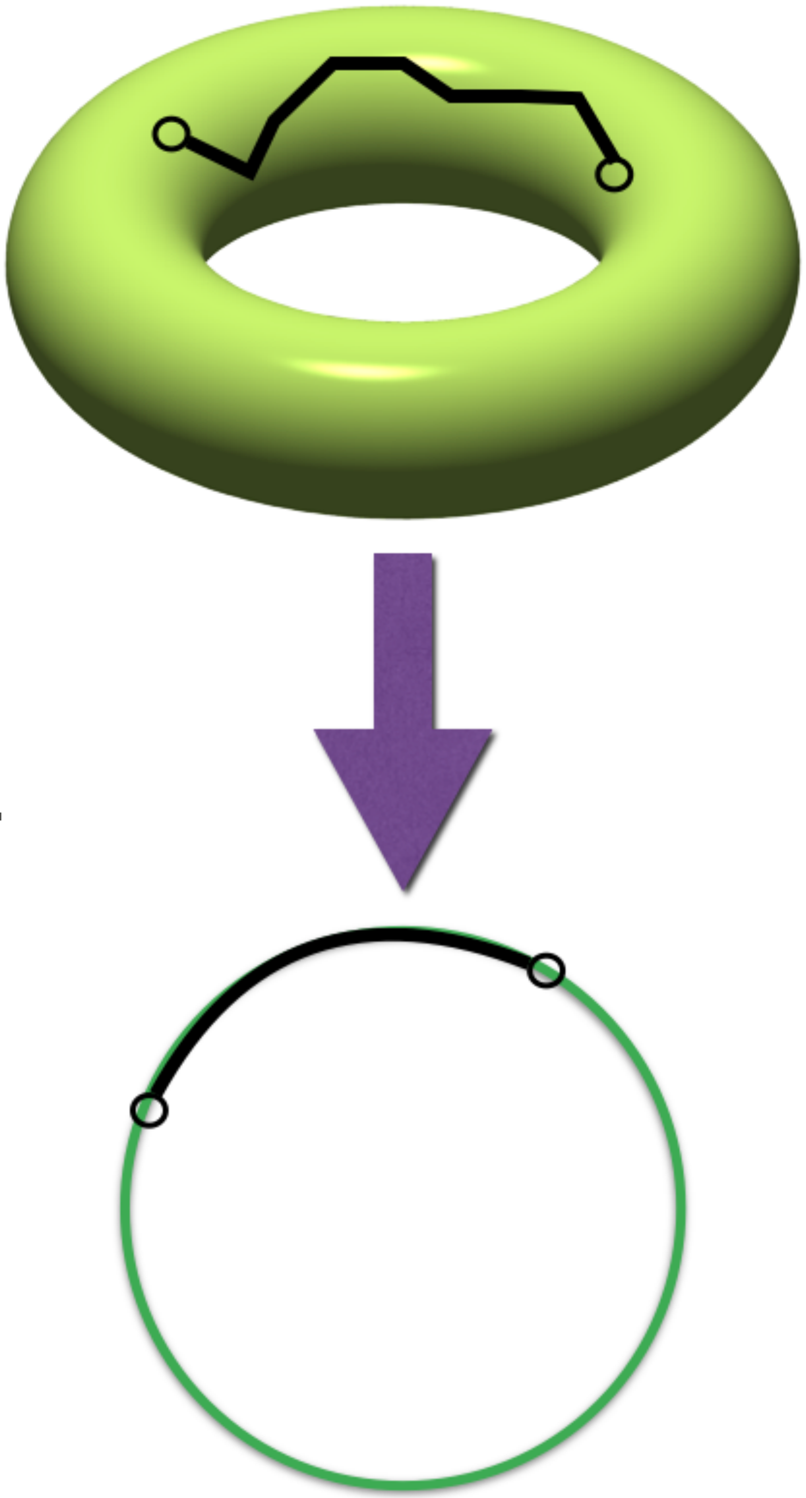


Type constructors act on points

$$\frac{\texttt{A type} \quad \texttt{B type}}{\texttt{A} \to \texttt{B type}}$$

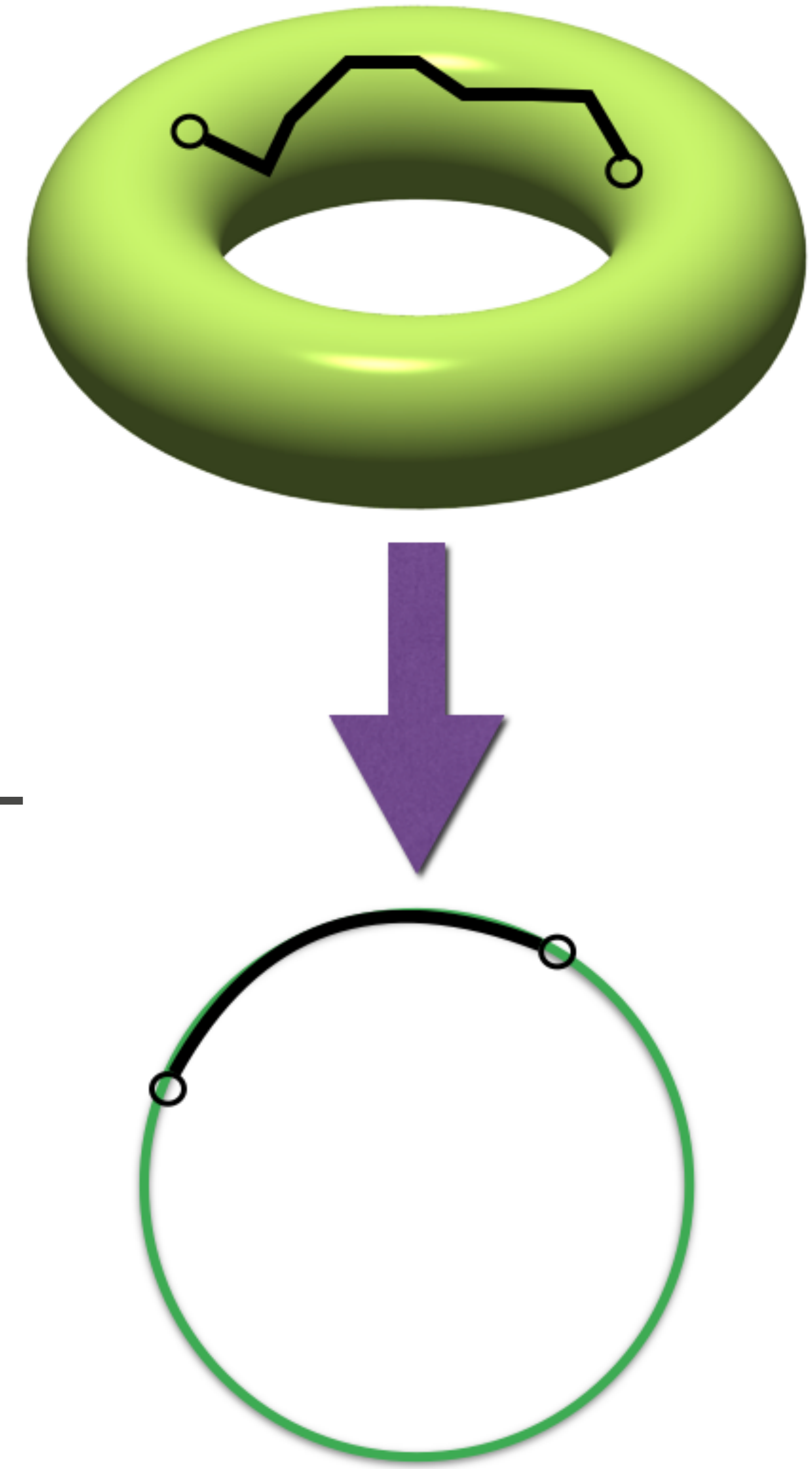


And “secretly” act on paths



$$\frac{\alpha : \text{Path } A\ A' \qquad \beta : \text{Path } B\ B'}{\alpha \to \beta : \text{Path } (A \to B)\ (A' \to B')}$$

And “secretly” act on paths



$$\dfrac{\alpha : \mathrm{Path}\ A\ A' \qquad \beta : \mathrm{Path}\ B\ B'}{\alpha \to \beta : \mathrm{Path}\ (A \to B)\ (A' \to B')}$$

$$\mathrm{coe}\ (\alpha \to \beta)\ (h : A \to B) = \mathrm{coe}\ \beta \circ h \circ \mathrm{coe}\ \alpha^{-1}$$

```
 instance Monad Maybe where
          return a = Some a
        None >>= k = None
    (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

ua(d)

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
    return-pure : ∀ {A} {a : A} → pure a == return a
    <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                → f <*> a == ( f >>= λ f' →
                               a >>= λ a' →
                               return (f' a'))
```

```
C : ((Type → Type) → Type) → Type
C Mon = {instance  : Mon Maybe,
         sequenceM : ∀ {T A} {Mon T}
                   → List (T A) → T(List A)}
```

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

ua(d)

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
    return-pure : ∀ {A} {a : A} → pure a == return a
    <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                → f <*> a == ( f >>= λ f' →
                               a >>= λ a' →
                               return (f' a'))
```

```
C : ((Type → Type) → Type) → Type
C Mon = {instance  : Mon Maybe,
         sequenceM : ∀ {T A} {Mon T}
                   → List (T A) → T(List A)}
```

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

ua(d)

C

C[Monad]

C[ua(d)]

C[App⇒Monad]

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
    return-pure : ∀ {A} {a : A} → pure a == return a
    <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                → f <*> a == ( f >>= λ f' →
                               a >>= λ a' →
                               return (f' a'))
```

```
C : ((Type → Type) → Type) → Type
C Mon = {instance : Mon Maybe,
         sequenceM : ∀ {T A} {Mon T}
                       → List (T A) → T(List A)}
```

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

ua(d)

C

C[Monad]

C[ua(d)]

C[App⇒Monad]

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
    return-pure : ∀ {A} {a : A} → pure a == return a
    <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                → f <*> a == ( f >>= λ f' →
                               a >>= λ a' →
                               return (f' a'))
```

```
C : ((Type → Type) → Type) → Type
C Mon = {instance : Mon Maybe,
         sequenceM : ∀ {T A} {Mon T}
                   → List (T A) → T(List A)}
```

```
record Monad (T : Type → Type) : Type where
  field
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
```

ua(d)

C

C[Monad]

C[ua(d)]

C[App⇒Monad]

```
record App⇒Monad (T : Type → Type) : Type where
    AT : Applicative T
    return : ∀ {A} → A → T A
    _>>=_  : ∀ {A B} → T A → (A → T B) → T B
    lunit  : ∀ {A B} {a : A} {f : A → T B} → (return a >>= f) == f a
    runit  : ∀ {A} {c : T A} → (c >>= return) == c
    assoc  : ∀ {A B C} {c : T A} {f : A → T B} {g : B → T C}
           → ((c >>= f) >>= g) == c >>= (λ x → f x >>= g)
    return-pure : ∀ {A} {a : A} → pure a == return a
    <*>-ap      : ∀ {A B} {f : T (A → B)} {a : T A}
                → f <*> a == ( f >>= λ f' →
                               a >>= λ a' →
                               return (f' a'))
```

```
instance Monad Maybe where
          return a = Some a
       None >>= k = None
   (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

C[Monad]

coe C[ua(d)]

C[App⇒Monad]

```
instance Monad Maybe where
        return a = Some a
      None >>= k = None
  (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

C[Monad]

coe C[ua(d)]

```
instance App⇒Monad Maybe where

        return a = Some a
      None >>= k = None
  (Some x) >>= k = k x
         pure a = Some a
        f <*> a = f >>= λf’ → a >>= λ a’ → return (f’ a’)

sequenceM : ∀ {T A} {App⇒Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

C[App⇒Monad]

```
instance Monad Maybe where
          return a = Some a
       None >>= k = None
   (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

coe ua(d)

C[Monad]

coe C[ua(d)]

C[App⇒Monad]

```
instance App⇒Monad Maybe where
          return a = Some a
       None >>= k = None
   (Some x) >>= k = k x
           pure a = Some a
         f <*> a = f >>= λf’ → a >>= λ a’ → return (f’ a’)

sequenceM : ∀ {T A} {App⇒Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

```
instance Monad Maybe where
         return a = Some a
      None >>= k = None
  (Some a) >>= k = k a

sequenceM : ∀ {T A} {Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

C[Monad]

coe C[ua(d)]

coe ua(d)$^{-1}$

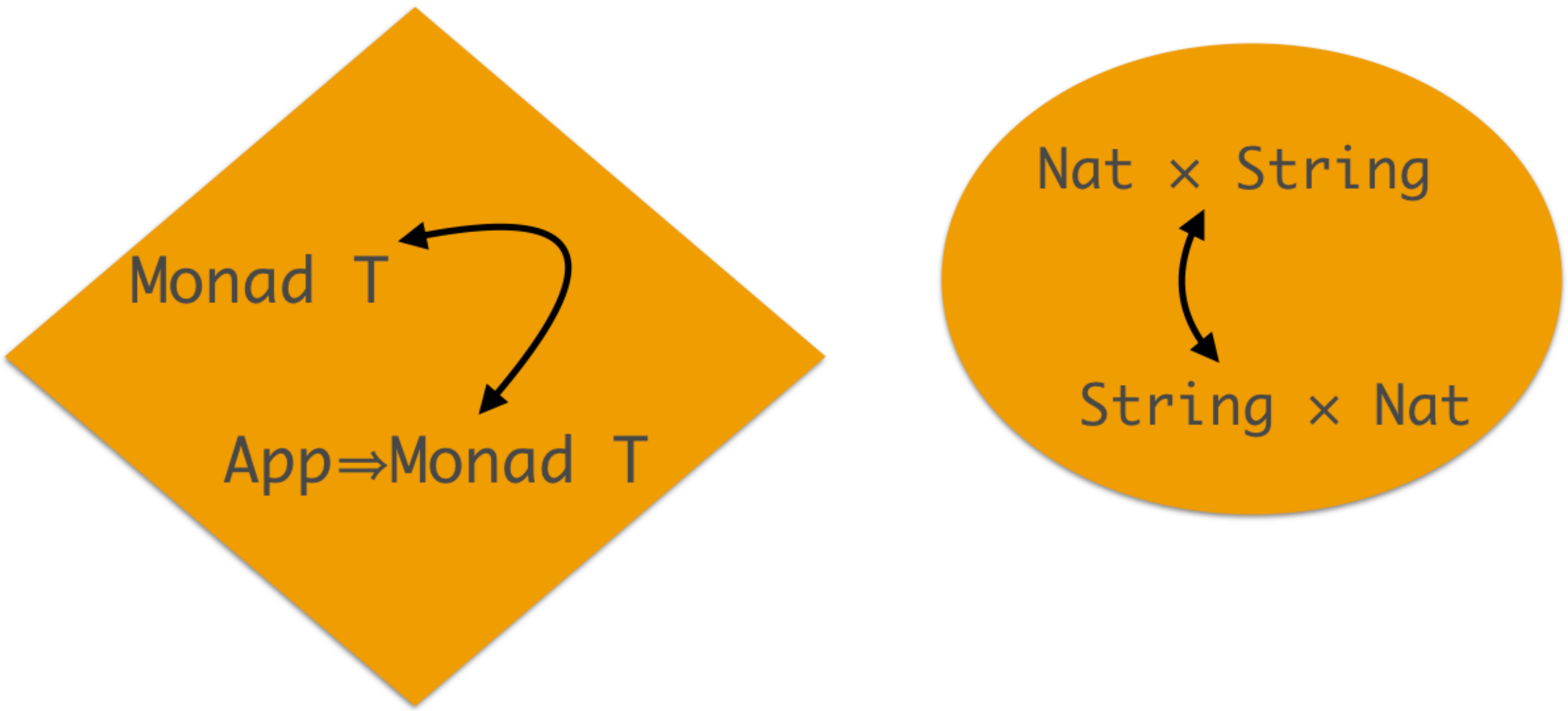
```
instance App⇒Monad Maybe where
         return a = Some a
      None >>= k = None
  (Some x) >>= k = k x
          pure a = Some a
         f <*> a = f >>= λf’→ a >>= λ a’ → return (f’ a’)

sequenceM : ∀ {T A} {App⇒Monad T} → List (T A) → T(List A)
sequenceM [] = return []
sequenceM (x :: xs) = x >>= λ xv →
         (sequenceM xs) >>= λ xsv →
             return (xv :: xsv)
```

C[App⇒Monad]

In a world where all functions secretly **do** something…

- get “code for free” / generic programs
- can add new principles that depend on them

Univalence



in a world where all functions act on paths,
… and paths between types induce bijections
you can allow bijections to induce paths
… and ∴ lift any bijection by a generic program

Which types act on paths?

Works for:

- Π, Σ, $+$, `Path`, (co)inductives

Which types act on paths?

Works for:

- Π, Σ, $+$, $\mathtt{Path}$, (co)inductives

Doesn't work for:

- intersection types $\quad A \cap B$

  made explicit as $\times$ of predicates

- intensional type analysis $\quad \mathtt{case}\ A\ \mathtt{of}$

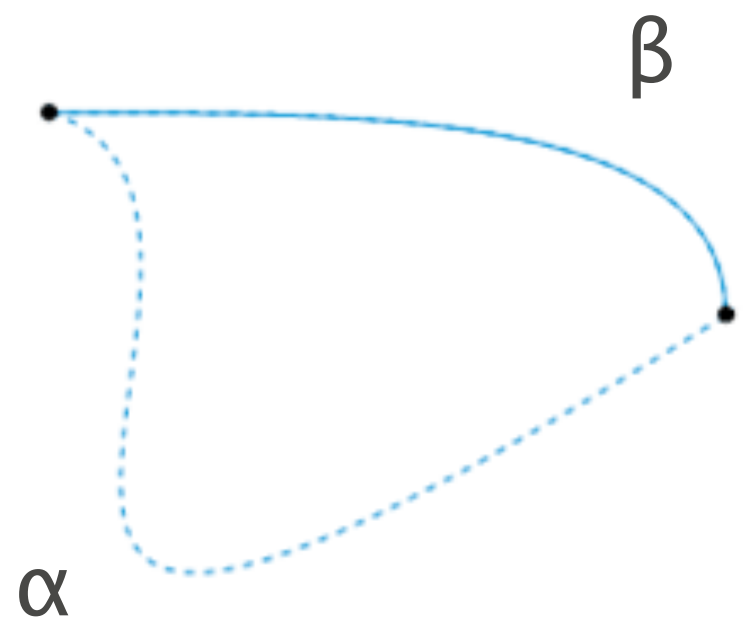
  $\qquad B \times C \Rightarrow \ldots$

  can define non-univalent inductive codes for types

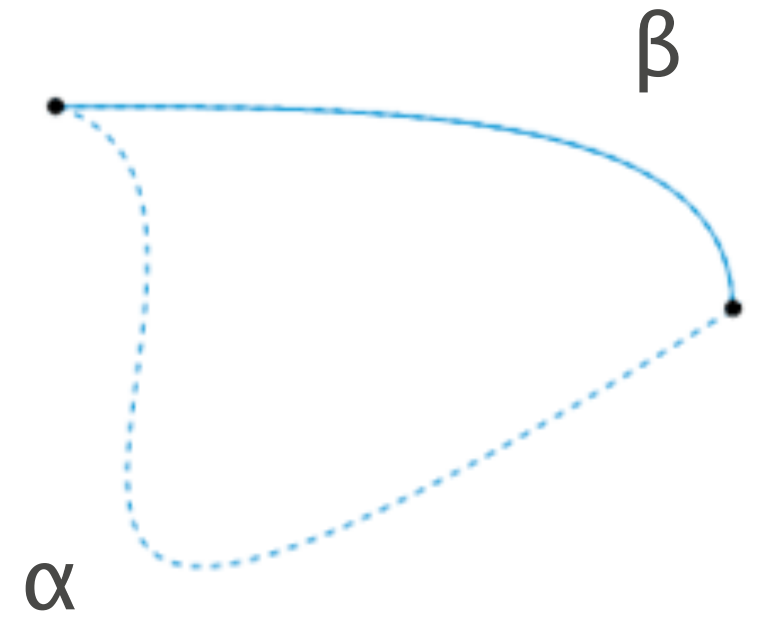
Other sources of bijections

- `List A ≃ Tree/{assoc,unit} A`
- `List` and `Tree/{assoc,unit}` implementations of ordered collections, if coercion of operations agree: `treemap f = fromlist ∘ listmap f ∘ tolist` (parametricity for graphs of bijections)
- `(Σ n:Nat.Vec A n) ≃ List A`
- `Everywhere P xs ≃ (x : A) → x ∈ xs → P x`
- Lots more in libraries/formalizations

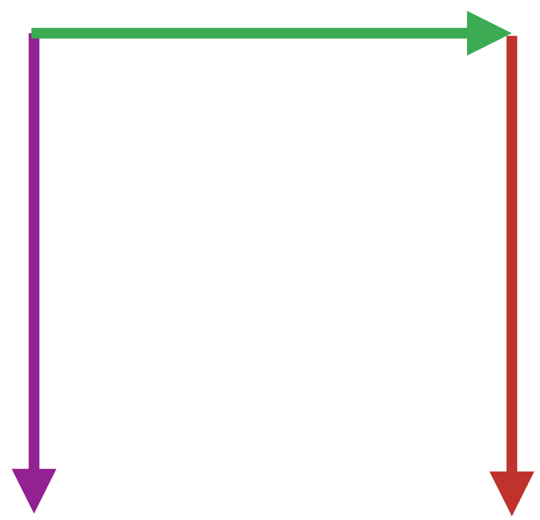
Paths are *data*



Paths are *data*

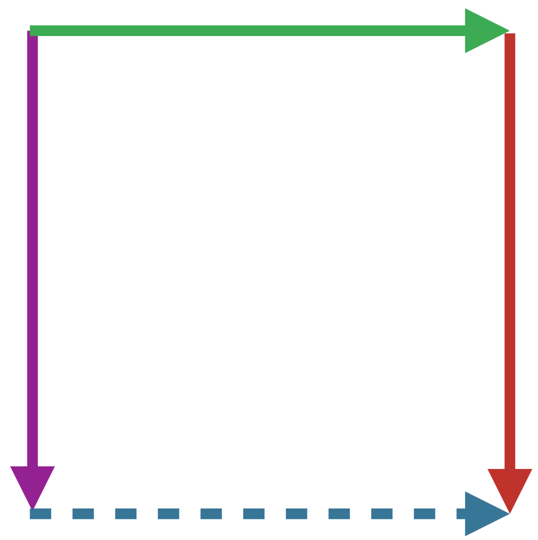


Cubical type theories



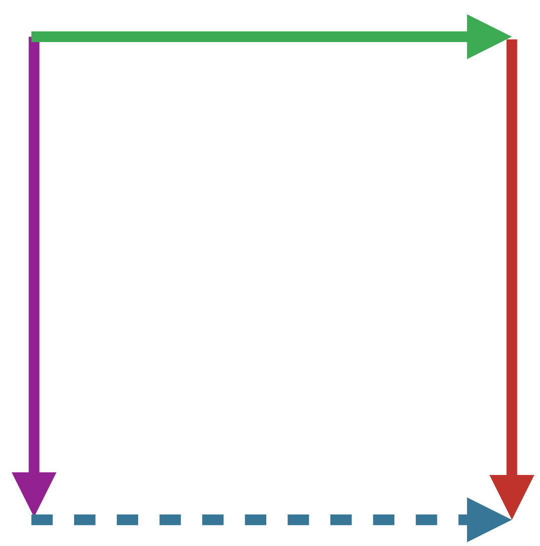
Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg;
Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata;
Angiuli,Harper,Wilson; Pitts,Orton

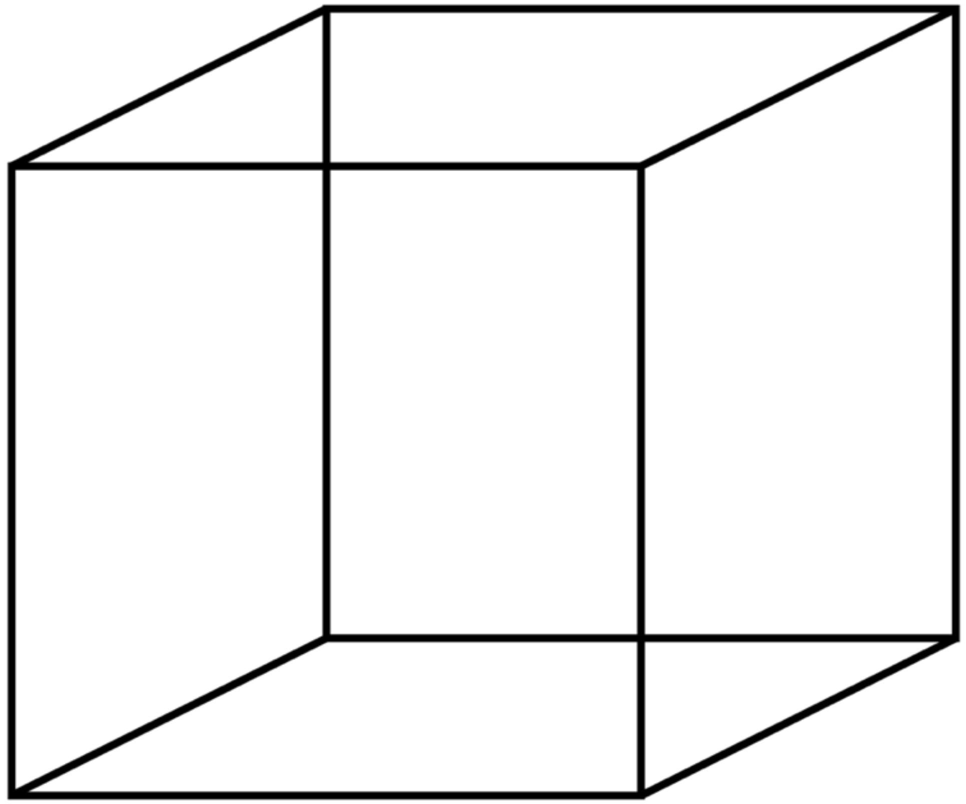
Cubical type theories



Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg; Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata; Angiuli,Harper,Wilson; Pitts,Orton

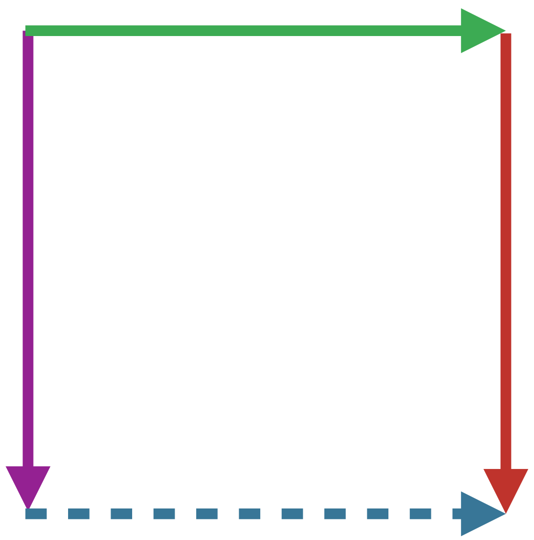
Cubical type theories

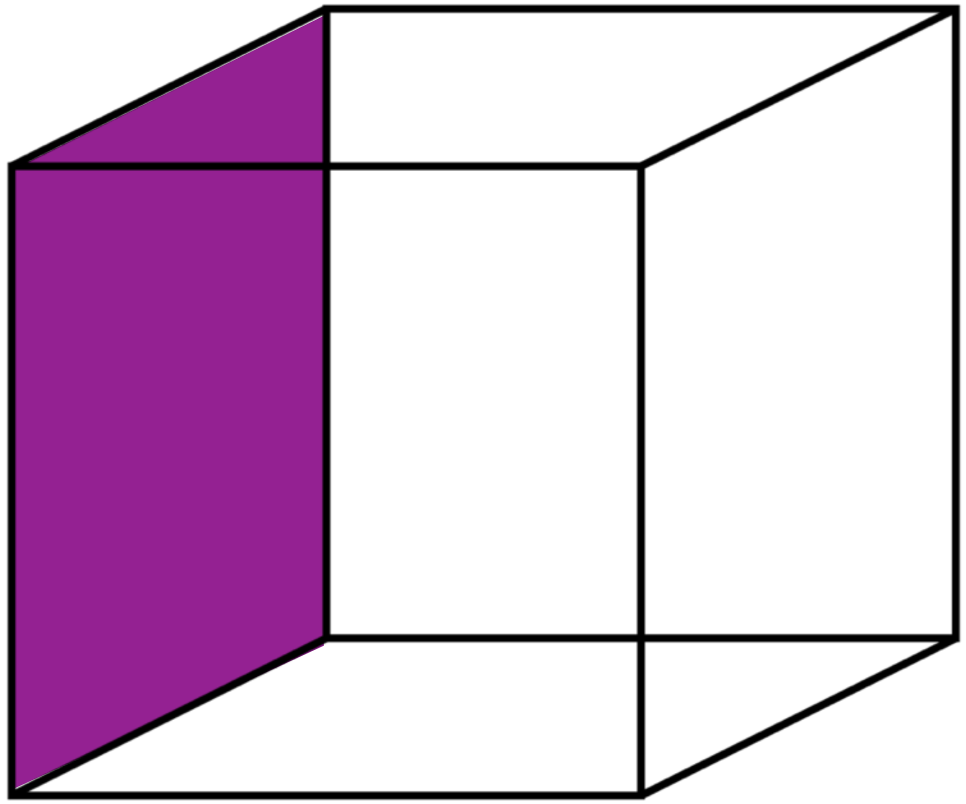




Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg;
Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata;
Angiuli,Harper,Wilson; Pitts,Orton

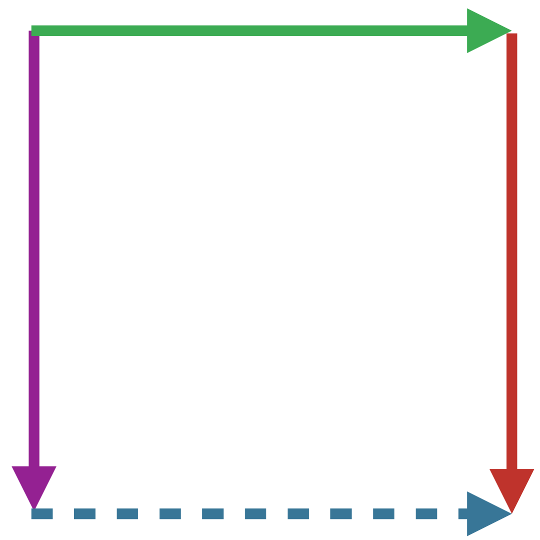
Cubical type theories

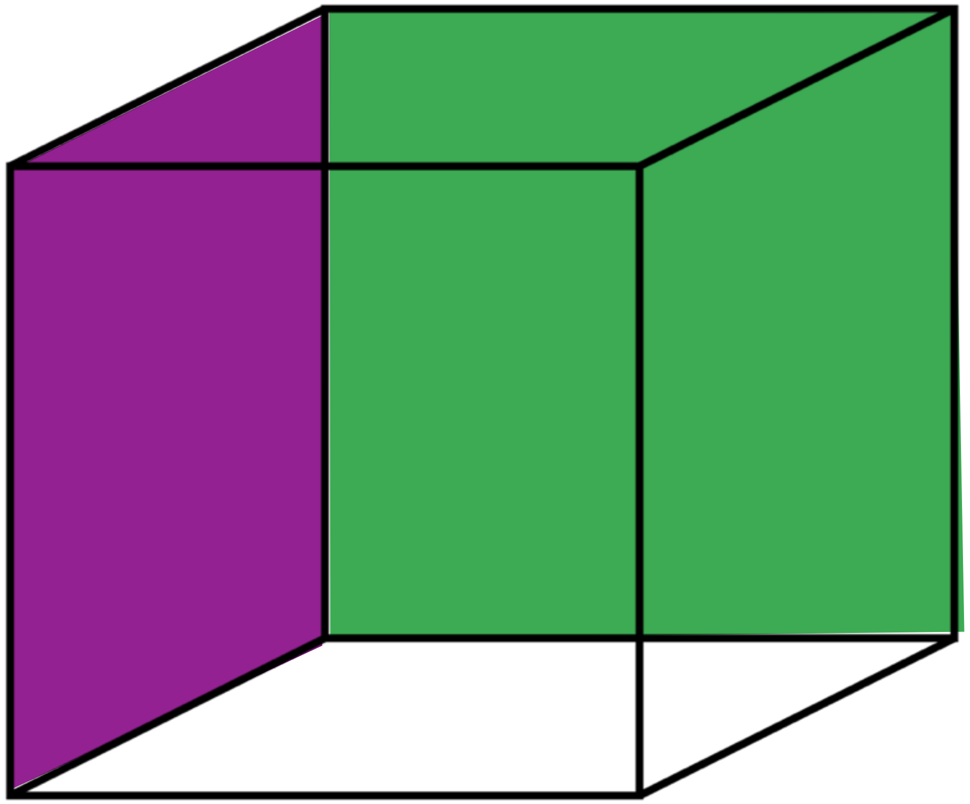




Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg;
Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata;
Angiuli,Harper,Wilson; Pitts,Orton

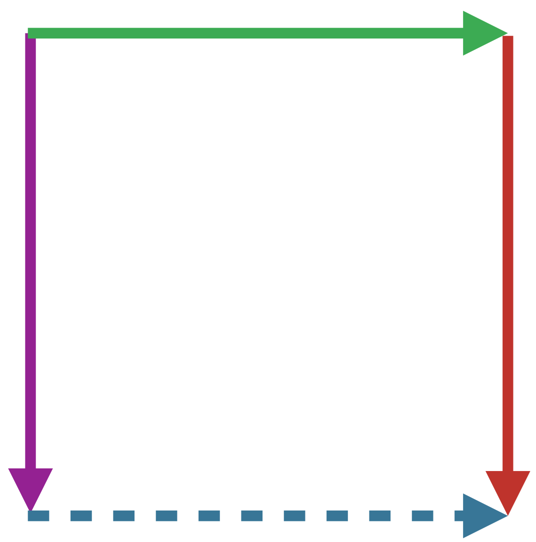
Cubical type theories

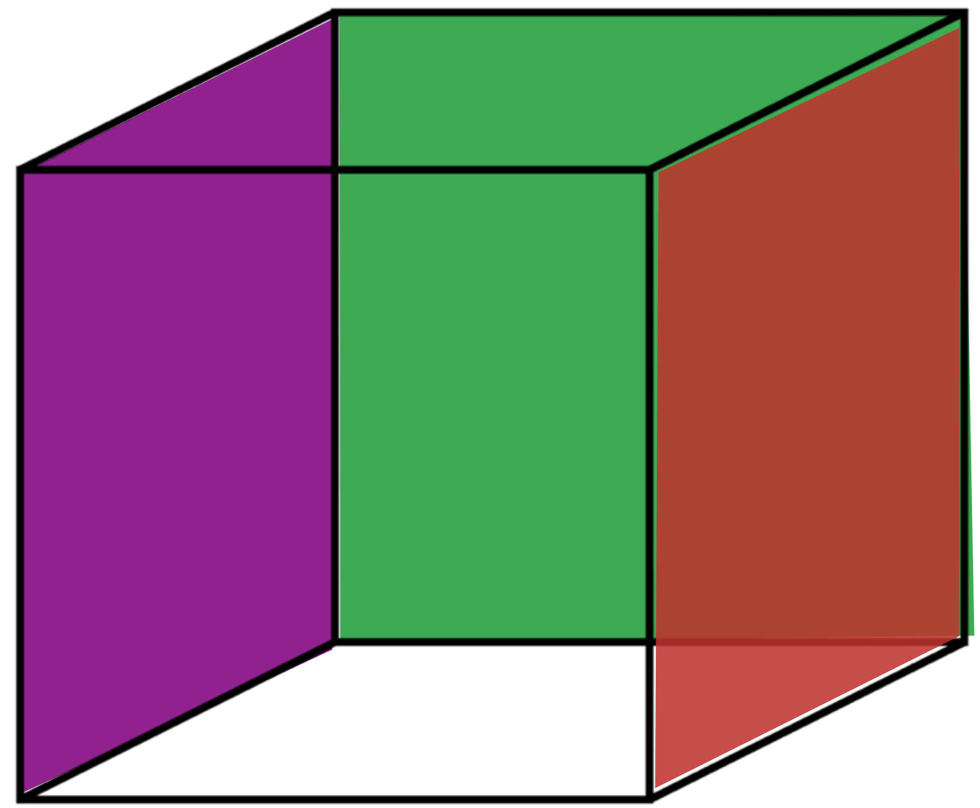




Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg;
Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata;
Angiuli,Harper,Wilson; Pitts,Orton

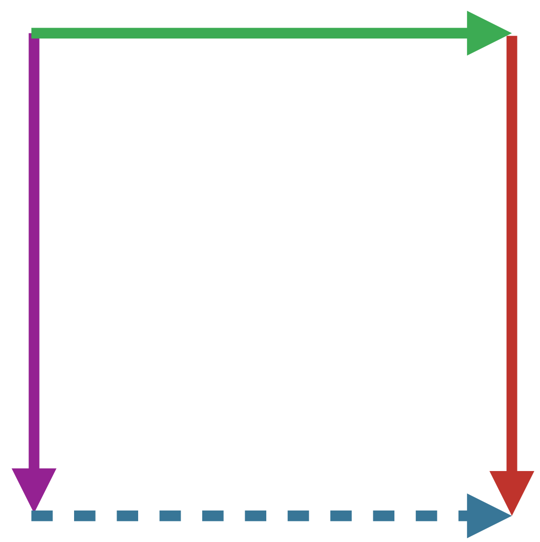
Cubical type theories

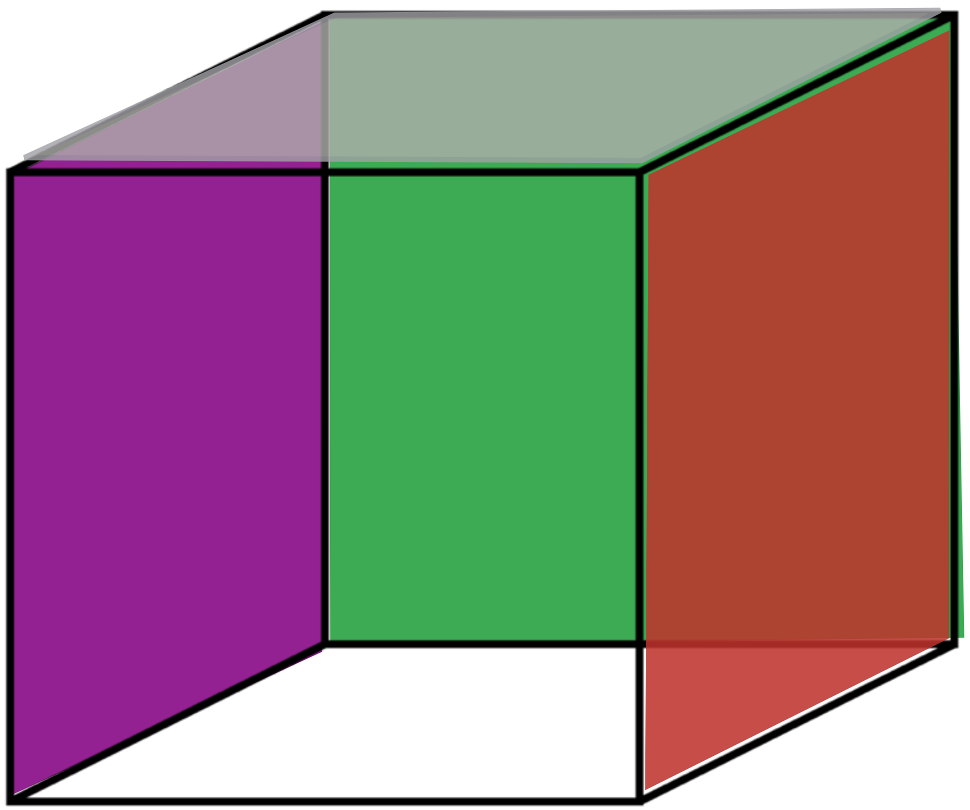




Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg;
Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata;
Angiuli,Harper,Wilson; Pitts,Orton

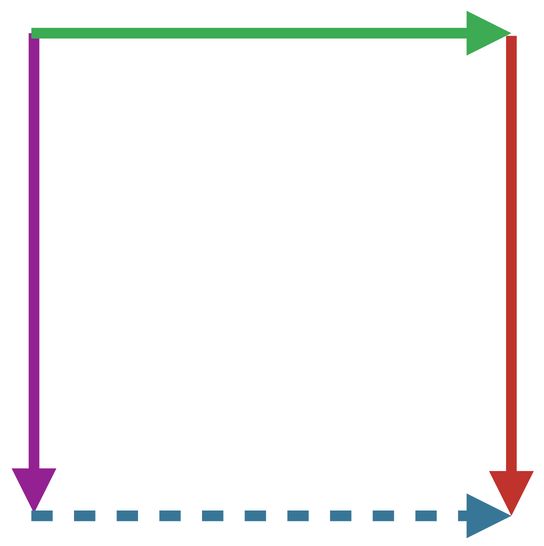
Cubical type theories

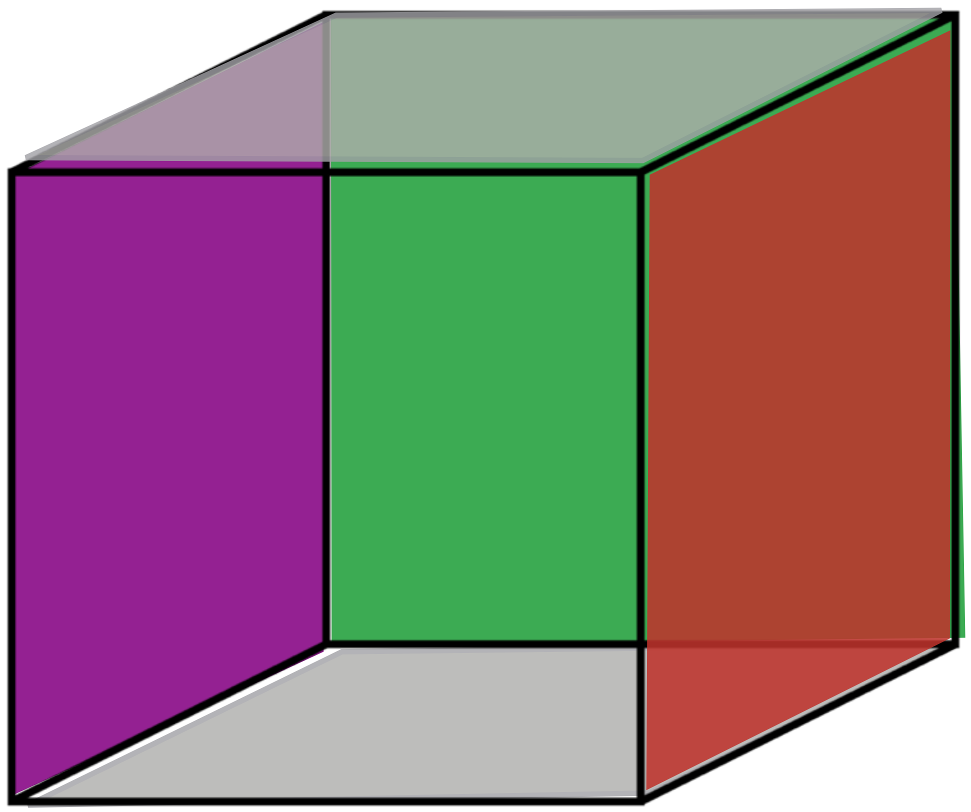




Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg; Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata; Angiuli,Harper,Wilson; Pitts,Orton

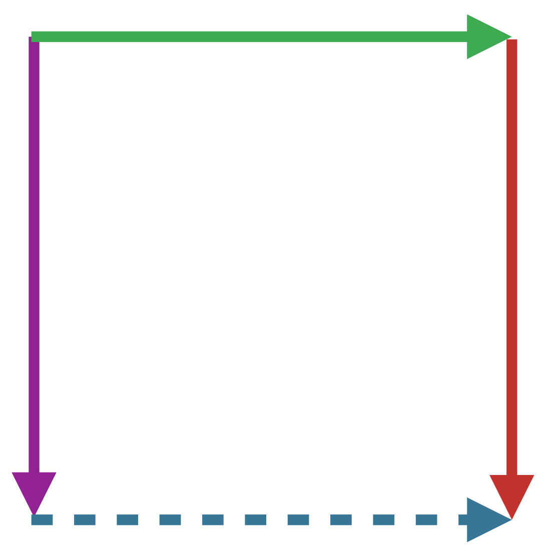
Cubical type theories

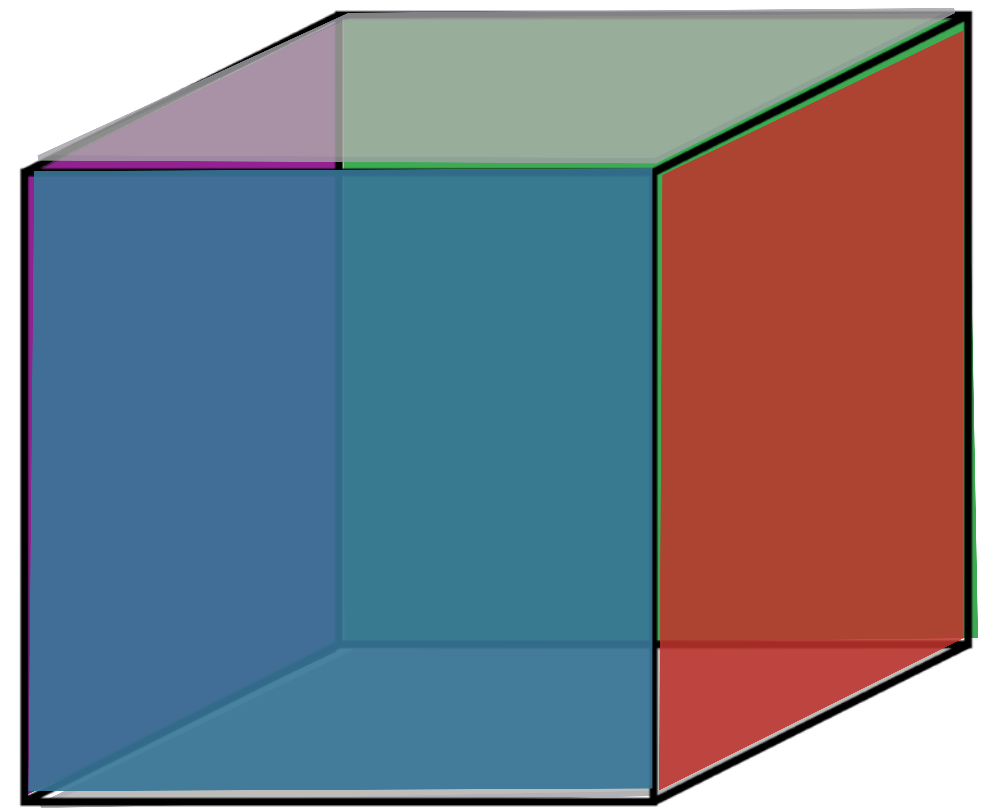




Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg;
Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata;
Angiuli,Harper,Wilson; Pitts,Orton

Cubical type theories



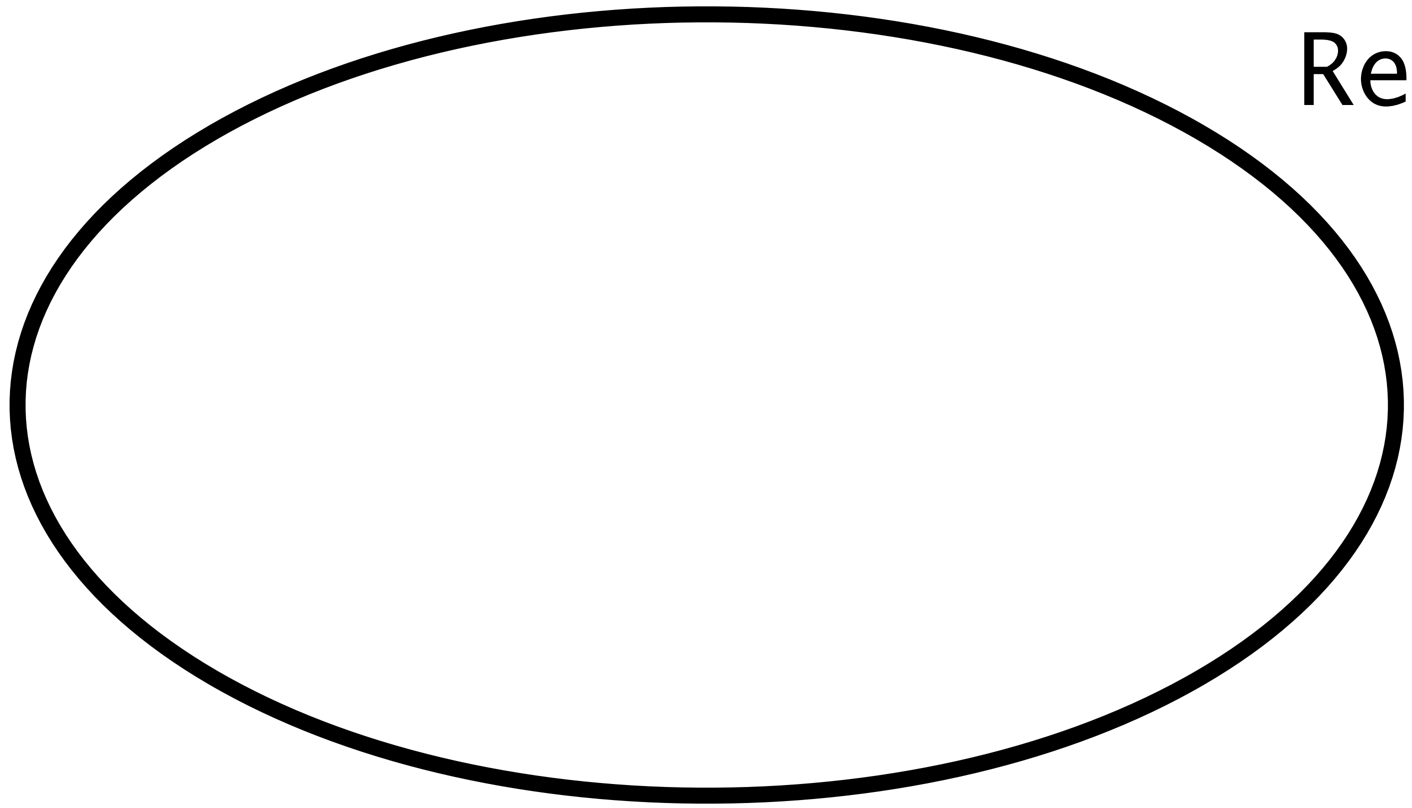


Bezem,Coquand,Huber; Cohen,Coquand,Huber,Mörtberg;
Polonsky; Altenkirch,Kaposi; Isaev; Brunerie,Licata;
Angiuli,Harper,Wilson; Pitts,Orton

Datatypes with paths

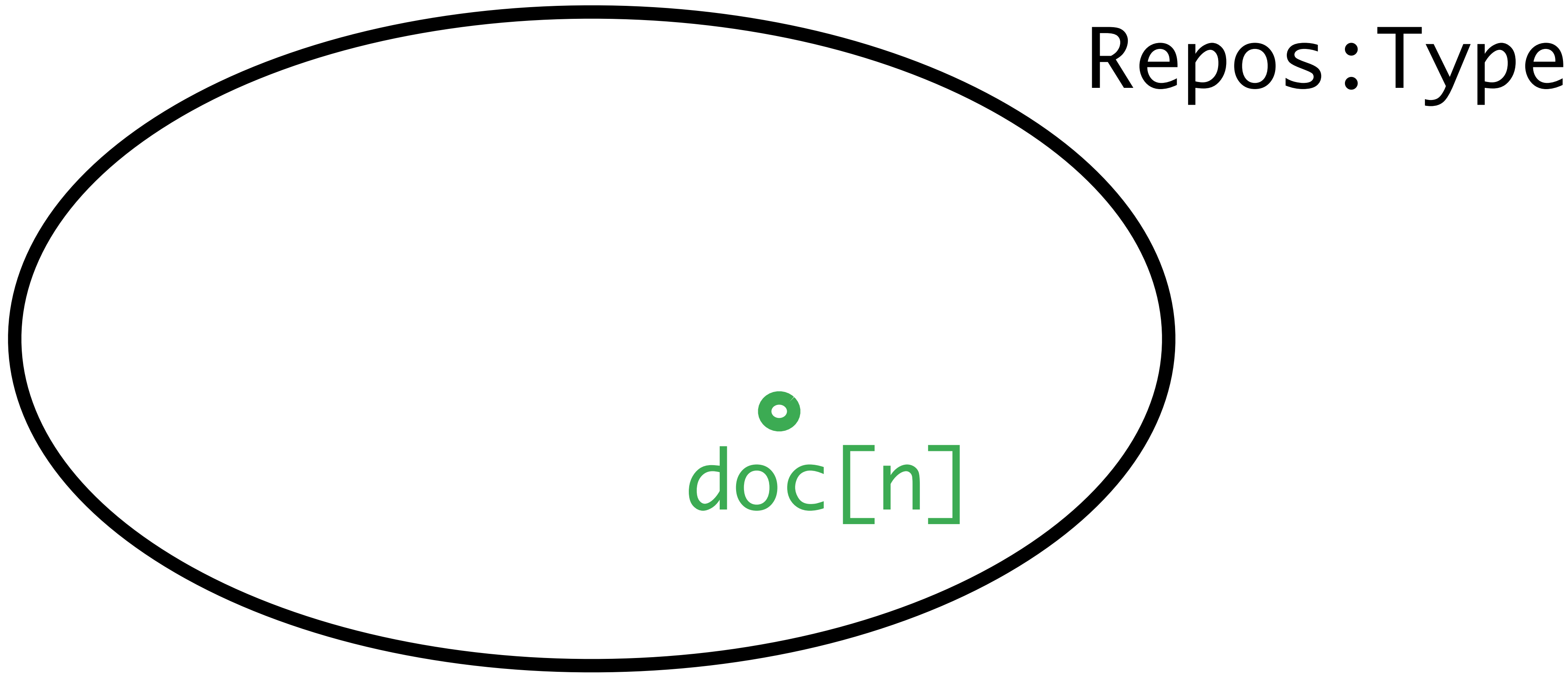
Patches as Paths

[Angiuli,Morehouse, L.,Harper]



Patches as Paths

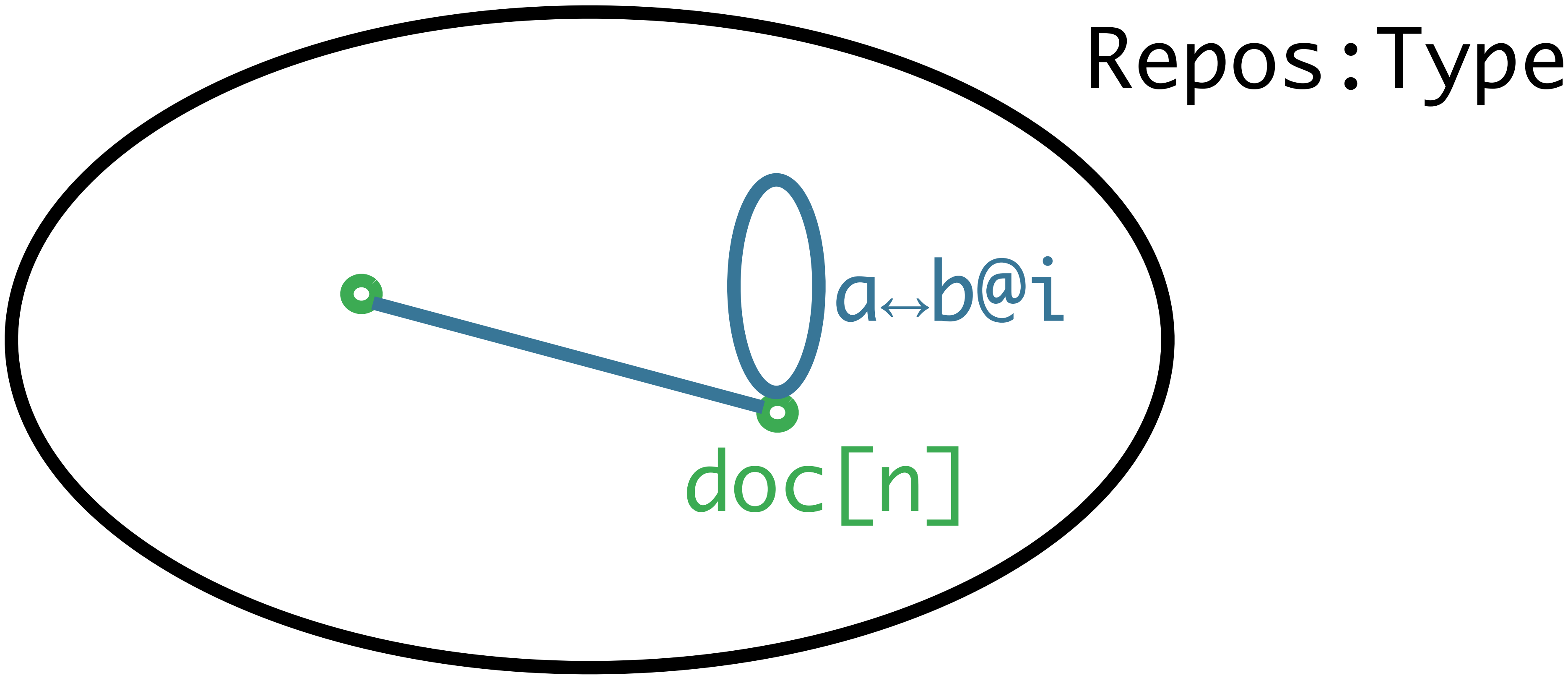
[Angiuli,Morehouse, L.,Harper]



points describe repository contents

Patches as Paths

[Angiuli,Morehouse, L.,Harper]

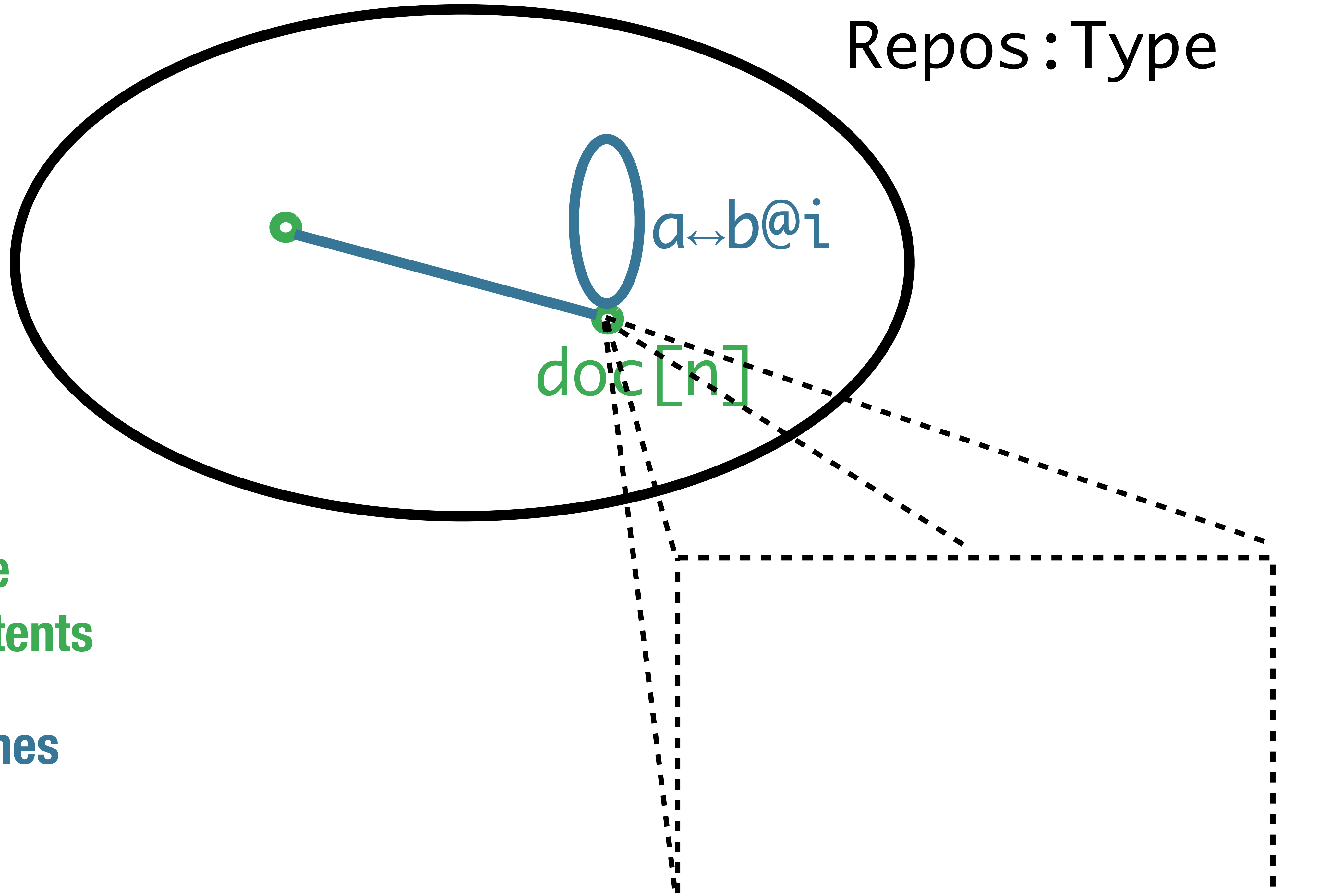


points describe repository contents

paths are patches

Patches as Paths

[Angiuli,Morehouse, L.,Harper]

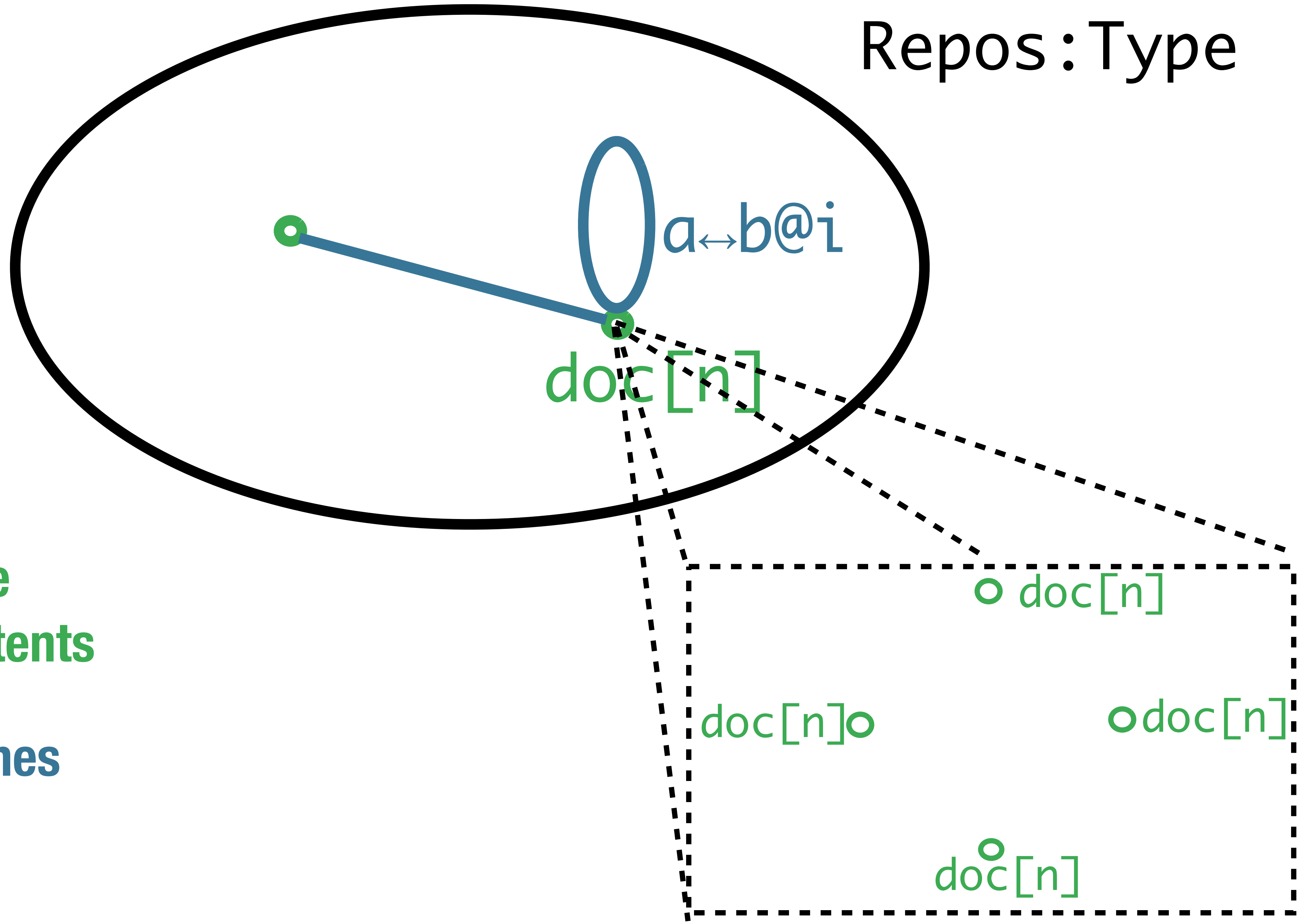


points describe repository contents

paths are patches

Patches as Paths

[Angiuli,Morehouse, L.,Harper]

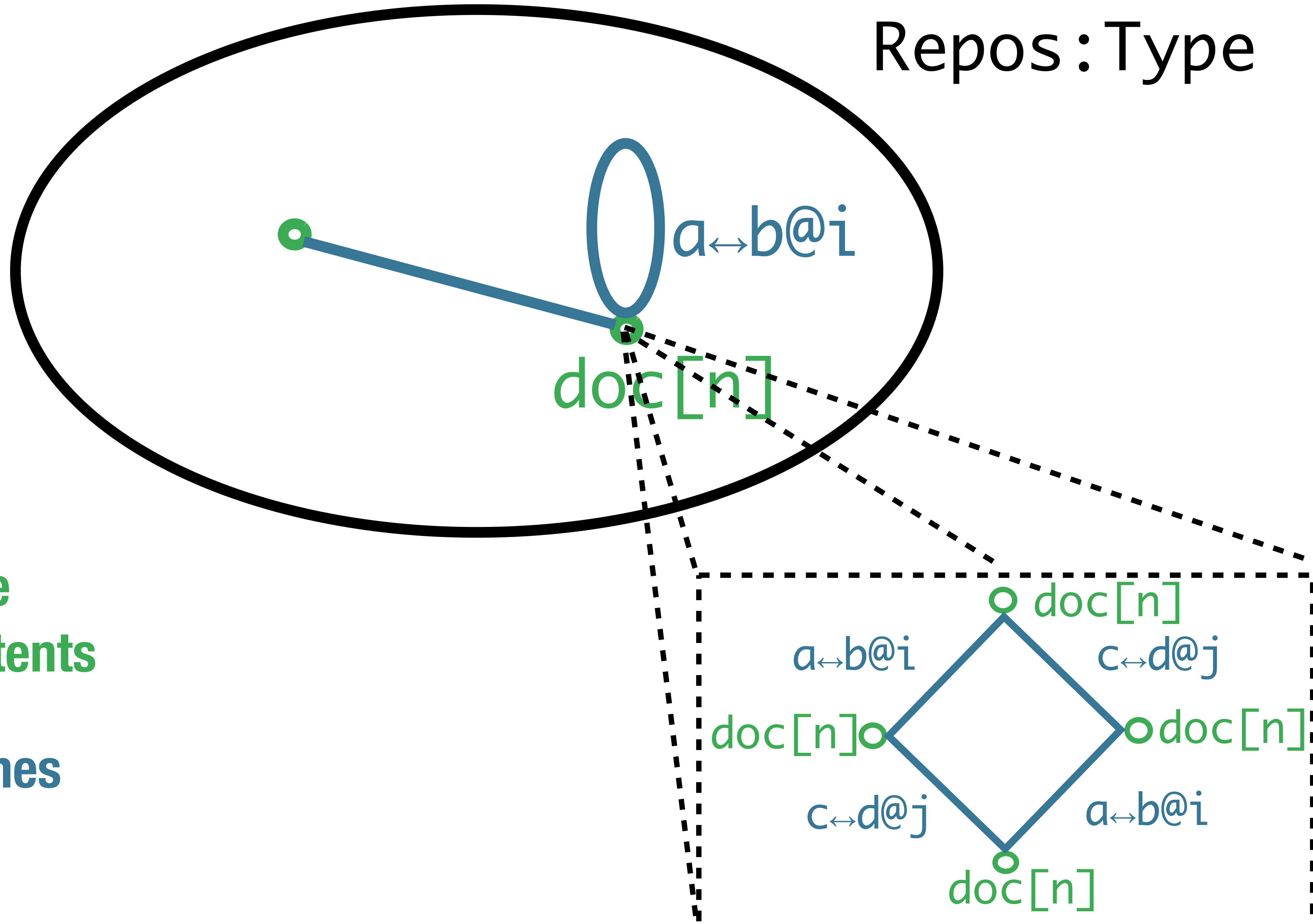


points describe repository contents

paths are patches

Patches as Paths

[Angiuli,Morehouse, L.,Harper]

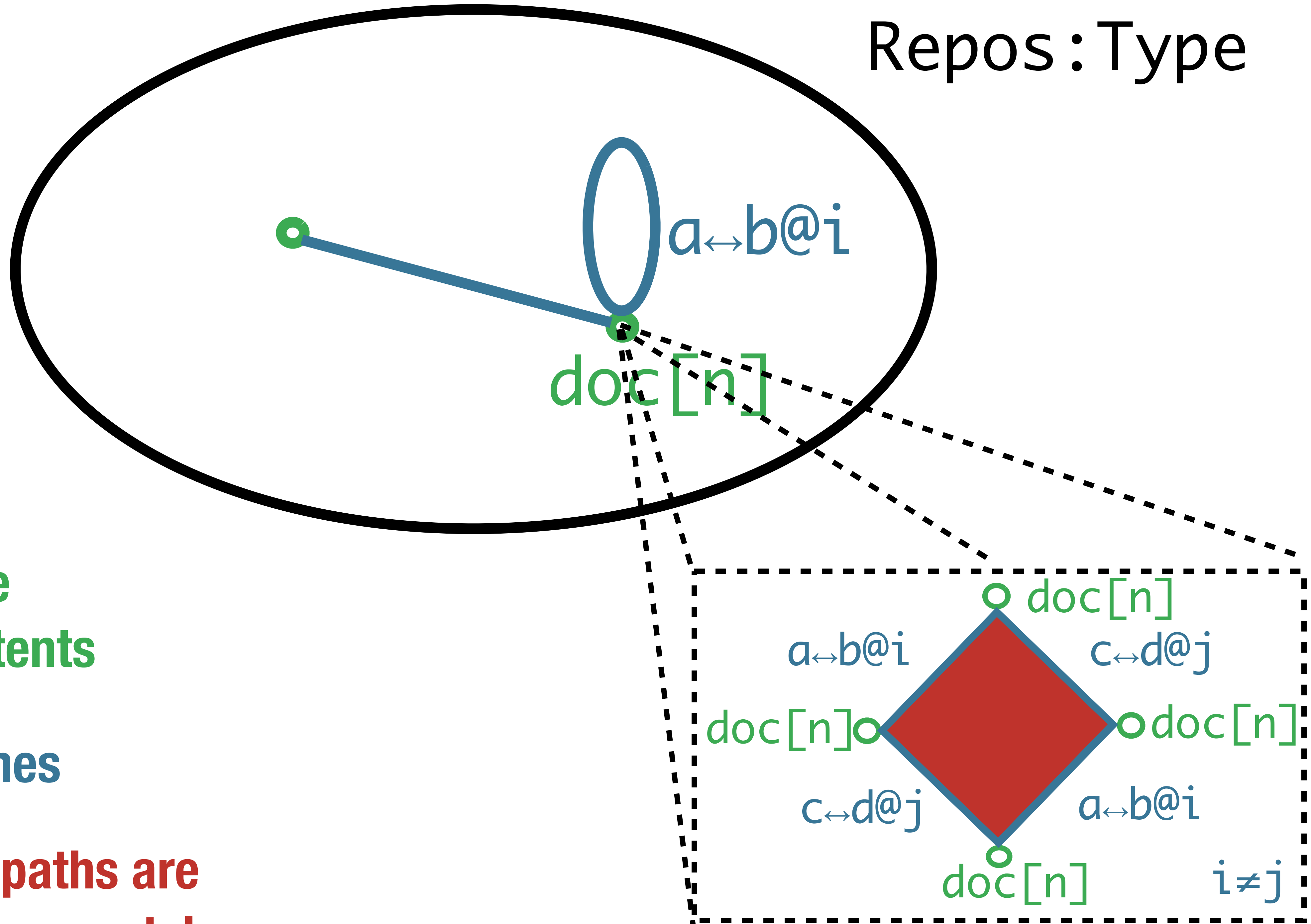


points describe repository contents

paths are patches

Patches as Paths

[Angiuli,Morehouse, L.,Harper]



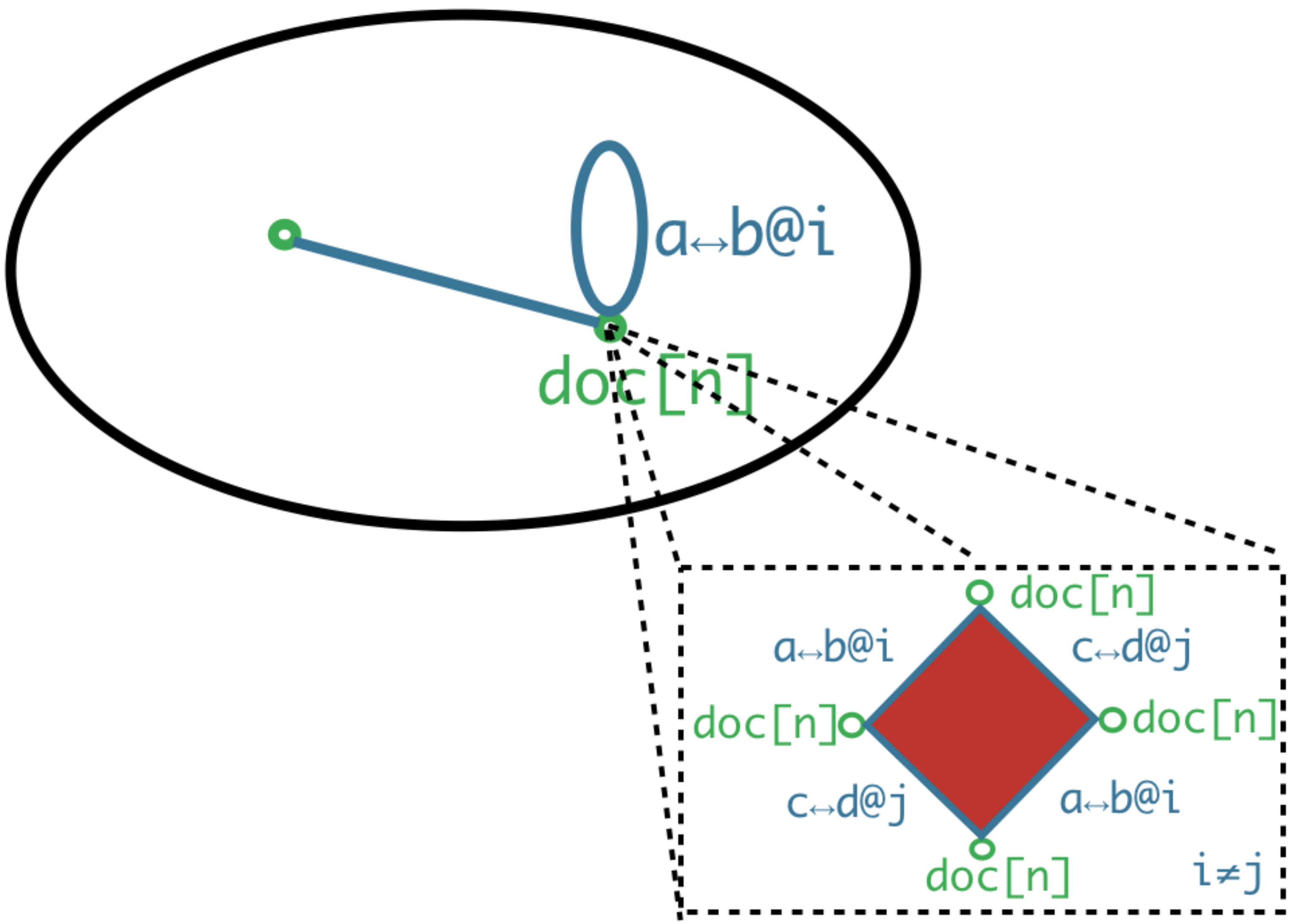
points describe repository contents

paths are patches

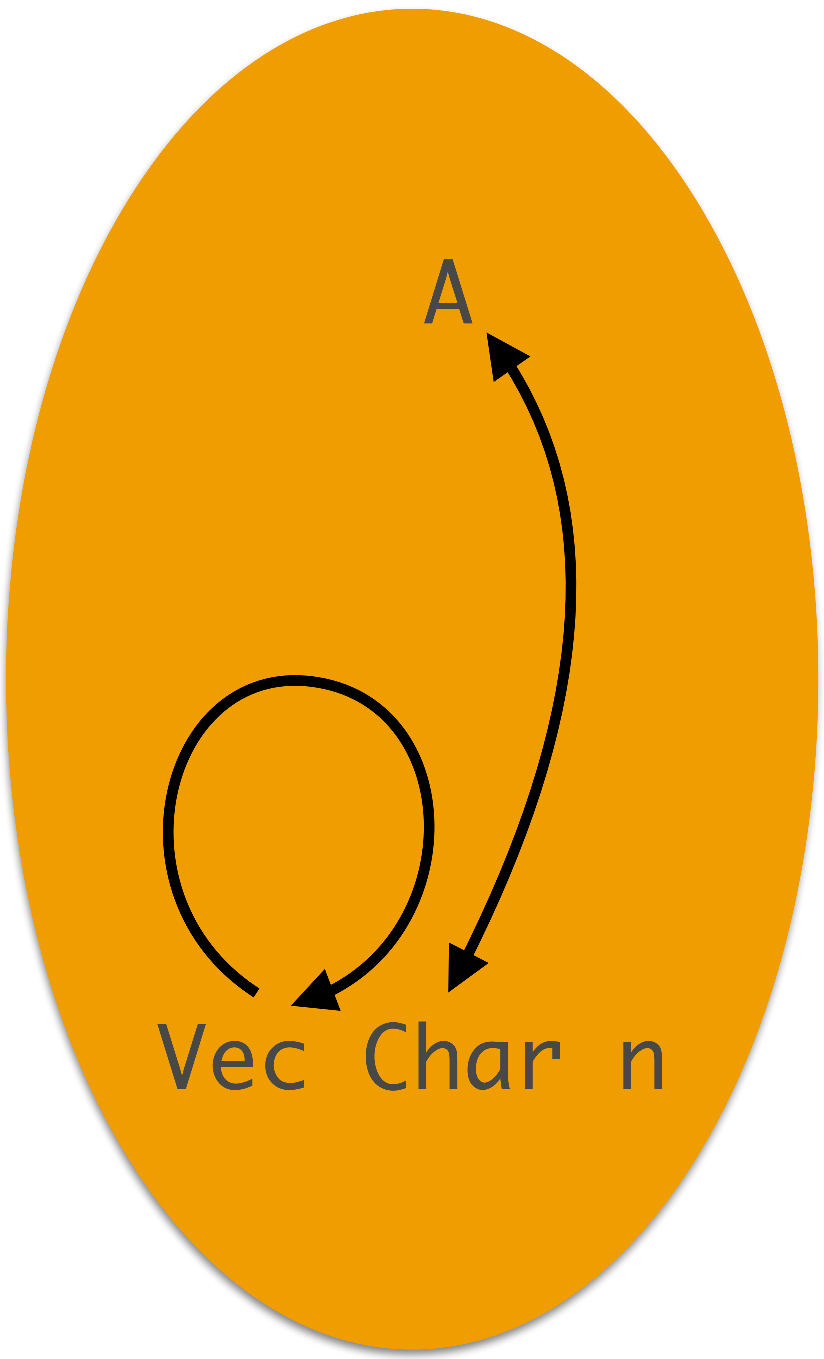
paths between paths are equations between patches

Higher inductive type

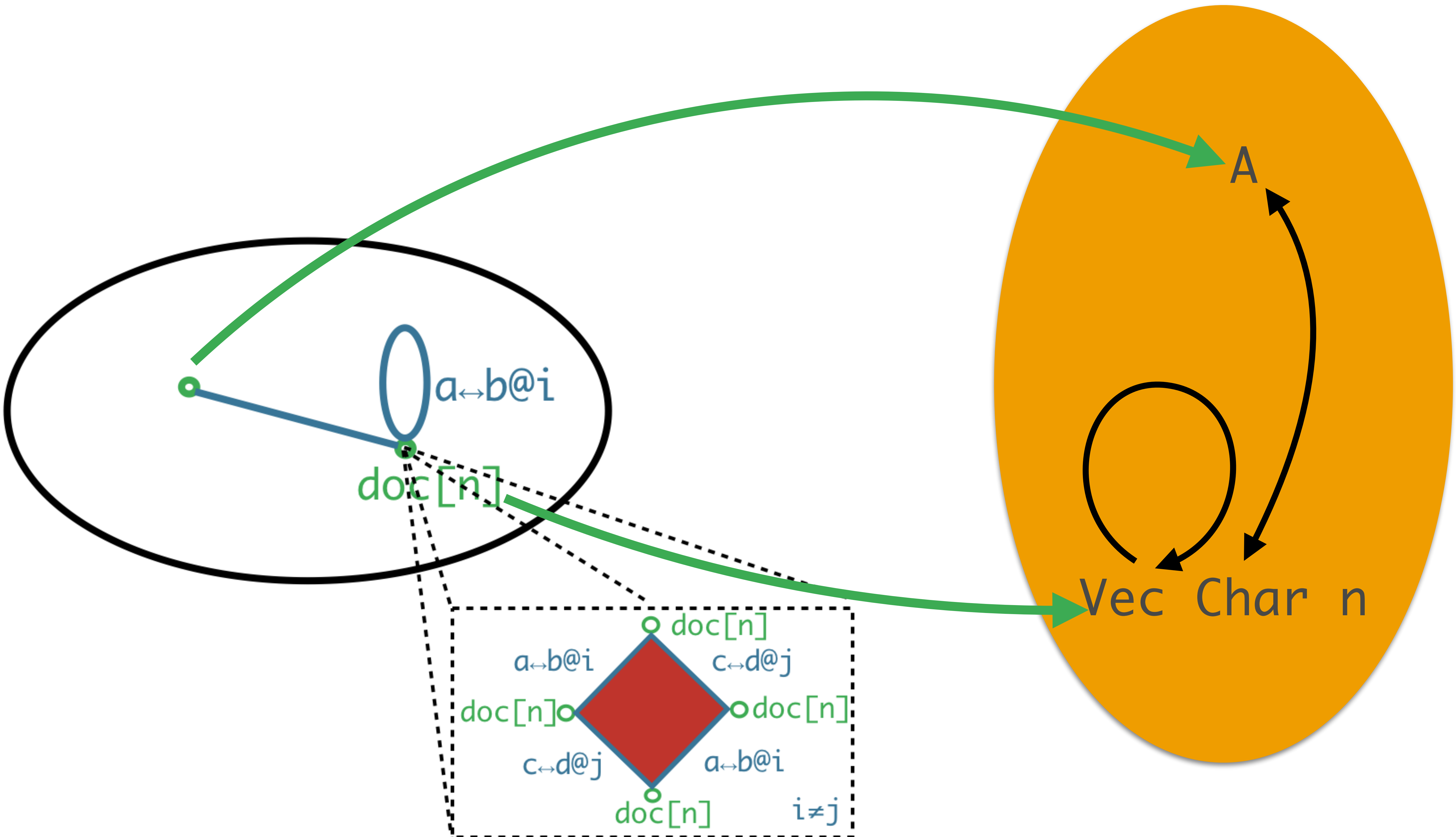
```
space Repos where
    doc[n:Nat] : Repos
    a↔b@i      : Path doc[n] doc[n]
    commute    : (i<n, j<n, i≠j) →
      Square (a↔b@i) (c↔d@j) (c↔d@j) (a↔b@i)
```

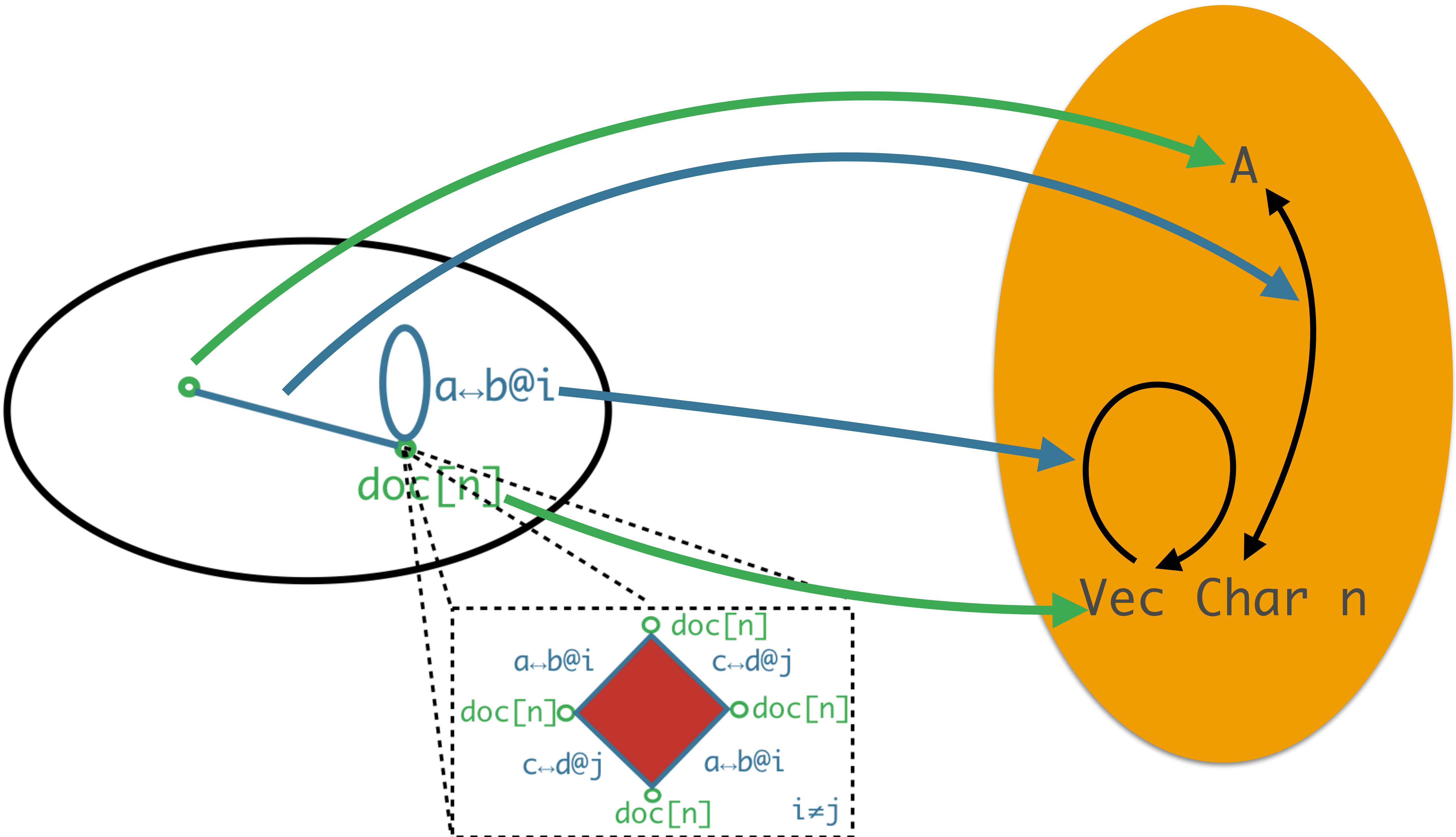


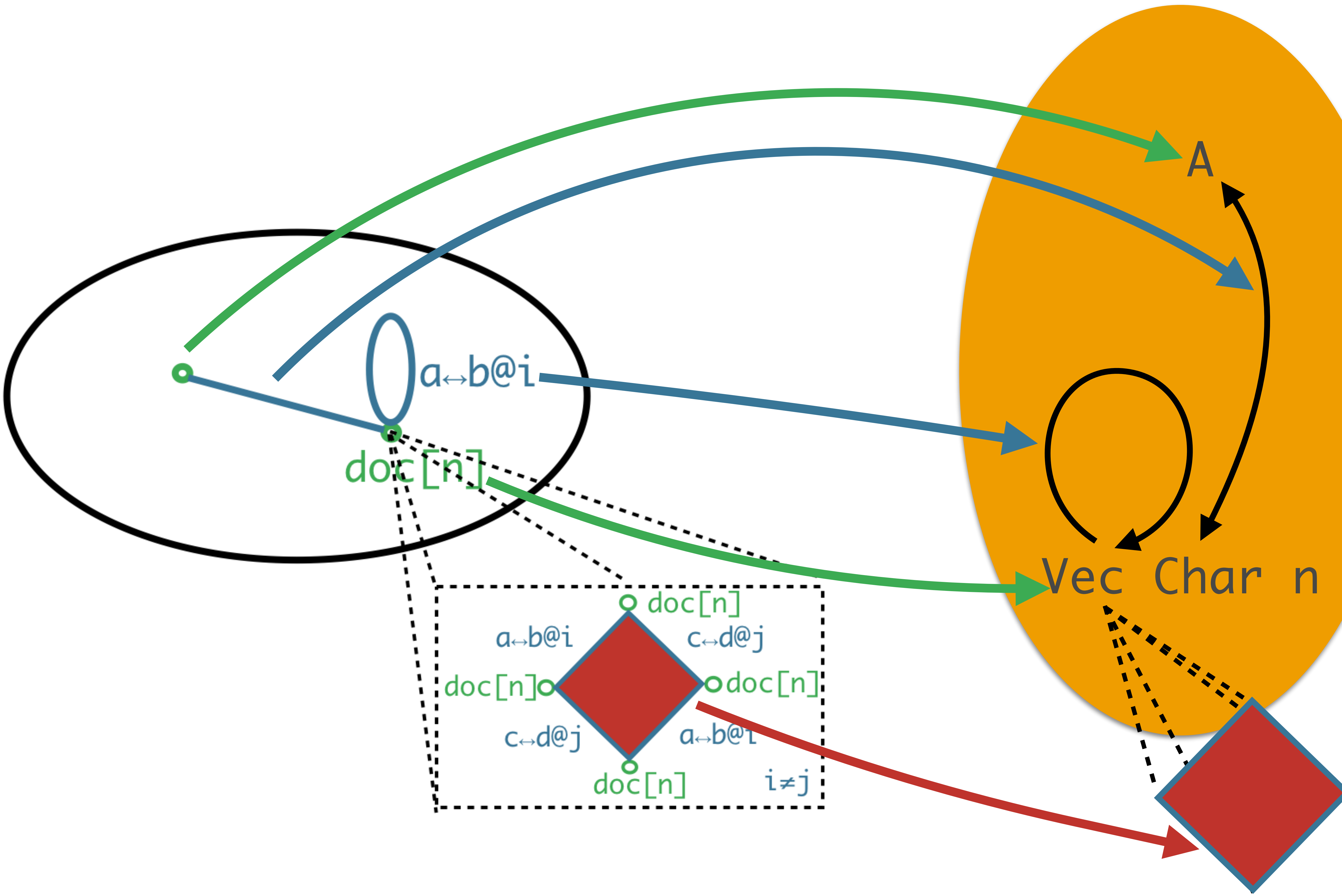
a↔b@i
doc[n]
doc[n]
a↔b@i
c↔d@j
doc[n]
doc[n]
c↔d@j
a↔b@i
doc[n]
i≠j



A
Vec Char n


a↔b@i
doc[n]
doc[n]
a↔b@i
c↔d@j
doc[n]
doc[n]
c↔d@j
a↔b@i
doc[n]
i≠j
A
Vec Char n

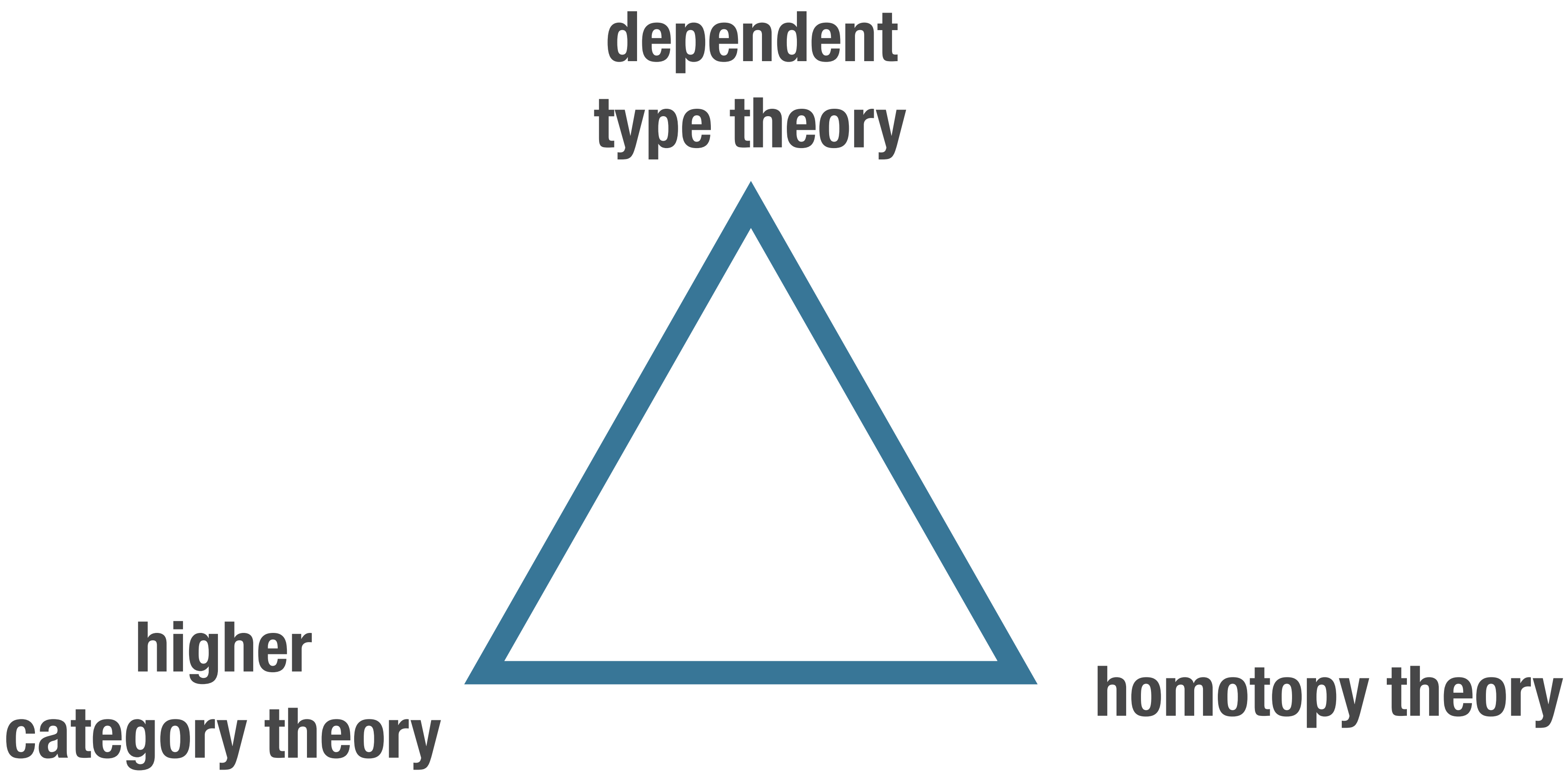

A
a↔b@i
doc[n]
Vec Char n
doc[n]
a↔b@i
c↔d@j
doc[n]
doc[n]
c↔d@j
a↔b@i
doc[n]
i≠j



A
a↔b@i
doc[n]
Vec Char n
doc[n]
a↔b@i
c↔d@j
doc[n]
doc[n]
c↔d@j
a↔b@i
doc[n]
i≠j

Interpreter

```
interp : Repos → Type
interp(doc[n]) = Vec Char n
interp(a↔b@i) = ua(… actual swap code …)
interp(commute) = … proof about above …
```


dependent
type theory
higher
category theory
homotopy theory

In a world where all functions secretly **are** *something…*

In a world where all functions secretly **do** *something…*