Extensional Equality Preservation in Intensional MLTT

Nuria Brede[12] **Tim Richter**[1] Patrik Jansson[3]
Nicola Botta[23]

[1]University of Potsdam, Potsdam, Germany, nuria.brede,tim.richter@uni-potsdam.de

[2]Potsdam Institute for Climate Impact Research, Potsdam, Germany, nicola.botta@pik-potsdam.de

[3]Chalmers University of Technology, Gothenburg, Sweden, patrik.jansson@chalmers.se

TYPES 2023
International Conference on Types for Proofs and Programs
Valencia, 2023-06-15

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.
Sometimes proofs get stuck without this property.

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.
Sometimes proofs get stuck without this property. But how to get it?

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.
Sometimes proofs get stuck without this property. But how to get it?

- Postulate function extensionality! Use cubical Agda!

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.
Sometimes proofs get stuck without this property. But how to get it?

- Postulate function extensionality! Use cubical Agda!
- Alternatively: *Add mapPresEE to the functor/monad ADT.*

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.
Sometimes proofs get stuck without this property. But how to get it?

- Postulate function extensionality! Use cubical Agda!
- Alternatively: *Add mapPresEE to the functor/monad ADT.*
  - Pro: Provable for monads used in MSDPs: *Maybe*, *List*, *FinProb*
  - Contra: Not for others, e.g. *Reader*.

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.
Sometimes proofs get stuck without this property. But how to get it?

- Postulate function extensionality! Use cubical Agda!
- Alternatively: *Add mapPresEE to the functor/monad ADT.*
  - Pro: Provable for monads used in MSDPs: *Maybe*, *List*, *FinProb*
  - Contra: Not for others, e.g. *Reader*. "Setoid Hell" ahead?

Motivation

- Proving a result about (finite horizon) *monadic* sequential decision problems (MSDP) in Idris, [Brede and Botta, 2021] observed:

A useful property: *mapPresEE*

It is nice if the *map* operation of your functors (or monads) sends extensionally equal functions to extensionally equal functions.
Sometimes proofs get stuck without this property. But how to get it?

- Postulate function extensionality! Use cubical Agda!
- Alternatively: *Add mapPresEE to the functor/monad ADT.*
  - Pro: Provable for monads used in MSDPs: *Maybe*, *List*, *FinProb*
  - Contra: Not for others, e.g. *Reader*. "Setoid Hell" ahead?
- [Botta et al., 2021] discusses use of *mapPresEE* and similar laws (*bindPresEE*, *kleisliPresEE*) to prove properties of MSDPs and equivalence of different monad ADTs:
  - Classical: *pure*, *map*, *join* + 5 laws
  - Wadler style: *pure*, *bind* + 3 laws
  - Fat interface with all above and Kleisli composition + laws

Equality in intensional MLTT

Judgemental equality $\Gamma \vdash a = a' : A$

Equality in intensional MLTT

Judgemental equality $\Gamma \vdash a = a' : A$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (f = \lambda x.f\ x) : A \to B}\ \eta$$

Equality in intensional MLTT

Judgemental equality $\Gamma \vdash a = a' : A$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (f = \lambda x.f\ x) : A \to B}\ \eta$$

Intensional (type level) equality

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Equality in intensional MLTT

Judgemental equality $\Gamma \vdash a = a' : A$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (f = \lambda x.f\ x) : A \to B}\ \eta$$

Intensional (type level) equality

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Extensional equality of functions

```
_≐_  : ∀ {A B : Set} → (f g : A → B) → Set
_≐_ {A} f g = (a : A) → f a ≡ g a
_≐≐_ : ∀ {A B C : Set} → (f g : A → B → C) → Set
_≐≐_ {A} {B} f g = (a : A) → (b : B) → f a b ≡ g a b
```

Equality in intensional MLTT

Judgemental equality $\Gamma \vdash a = a' : A$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (f = \lambda x.f\ x) : A \to B}\ \eta$$

Intensional (type level) equality

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Extensional equality of functions

```
_≐_   : ∀ {A B : Set} → (f g : A → B) → Set
_≐_ {A} f g = (a : A) → f a ≡ g a
_≐≐_ : ∀ {A B C : Set} → (f g : A → B → C) → Set
_≐≐_ {A} {B} f g = (a : A) → (b : B) → f a b ≡ g a b
```

$\equiv$, $\doteq$, $\doteq\doteq$ are equivalence relations.

Equality in intensional MLTT

Judgemental equality $\Gamma \vdash a = a' : A$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (f = \lambda x.f\ x) : A \to B}\ \eta$$

Intensional (type level) equality

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Extensional equality of functions

```
_≐_  : ∀ {A B : Set} → (f g : A → B) → Set
_≐_ {A} f g = (a : A) → f a ≡ g a
_≐≐_ : ∀ {A B C : Set} → (f g : A → B → C) → Set
_≐≐_ {A} {B} f g = (a : A) → (b : B) → f a b ≡ g a b
```

$\equiv$, $\doteq$, $\doteq\doteq$ are equivalence relations. We have $\equiv \to \doteq$,*but not* $\doteq \to \equiv$ (i.e. *funExt*).

Equality in intensional MLTT

Judgemental equality $\Gamma \vdash a = a' : A$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (f = \lambda x.f\ x) : A \to B}\ \eta$$

Intensional (type level) equality

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Extensional equality of functions

```
_≐_  : ∀ {A B : Set} → (f g : A → B) → Set
_≐_ {A} f g = (a : A) → f a ≡ g a
_≐≐_ : ∀ {A B C : Set} → (f g : A → B → C) → Set
_≐≐_ {A} {B} f g = (a : A) → (b : B) → f a b ≡ g a b
```

$\equiv$, $\doteq$, $\doteq\doteq$ are equivalence relations. We have $\equiv \to \doteq$,*but not* $\doteq \to \equiv$ (i.e. *funExt*).
Any function preserves $\equiv$:

```
cong : {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Extensional functors

Functor ADT:

```
record Functor (F : Set → Set) : Set₁ where
  field
    map           : ∀ {A B} → (A → B) → (F A → F B)
    mapPresId     : ∀ {A} → map (id {A = A}) ≐ id
    mapPresComp   : ∀ {A B C} (f : B → C) (g : A → B) →
                    map (f ∘ g) ≐ map f ∘ map g
```

Extensional functors

add *mapPresEE*

```
record ExtFunctor (F : Set → Set) : Set₁ where
  field
    map           : ∀ {A B} → (A → B) → (F A → F B)
    mapPresId     : ∀ {A} → map (id {A = A}) ≐ id
    mapPresComp   : ∀ {A B C} (f : B → C) (g : A → B) →
                    map (f ∘ g) ≐ map f ∘ map g
    mapPresEE     : ∀ {A B} (f g : A → B) →
                    f ≐ g → map f ≐ map g
```

Extensional functors

add *mapPresEE*

```
record ExtFunctor (F : Set → Set) : Set₁ where
  field
    map           : ∀ {A B} → (A → B) → (F A → F B)
    mapPresId     : ∀ {A} → map (id {A = A}) ≐ id
    mapPresComp   : ∀ {A B C} (f : B → C) (g : A → B) →
                    map (f ∘ g) ≐ map f ∘ map g
    mapPresEE     : ∀ {A B} (f g : A → B) →
                    f ≐ g → map f ≐ map g
```

[Matthes, 2009] used these "extensional functors" in the context of nested datatypes.

Extensional functors

add *mapPresEE*

```
record ExtFunctor (F : Set → Set) : Set₁ where
  field
    map           : ∀ {A B} → (A → B) → (F A → F B)
    mapPresId     : ∀ {A} → map (id {A = A}) ≐ id
    mapPresComp   : ∀ {A B C} (f : B → C) (g : A → B) →
                    map (f ∘ g) ≐ map f ∘ map g
    mapPresEE     : ∀ {A B} (f g : A → B) →
                    f ≐ g → map f ≐ map g
```

[Matthes, 2009] used these "extensional functors" in the context of nested datatypes. *mapPresEE* for *Maybe*, *List*,... e.g. in Coq and Agda standard libs.

Extensional functors

add *mapPresEE*

```
record ExtFunctor (F : Set → Set) : Set₁ where
  field
    map           : ∀ {A B} → (A → B) → (F A → F B)
    mapPresId     : ∀ {A} → map (id {A = A}) ≐ id
    mapPresComp   : ∀ {A B C} (f : B → C) (g : A → B) →
                    map (f ∘ g) ≐ map f ∘ map g
    mapPresEE     : ∀ {A B} (f g : A → B) →
                    f ≐ g → map f ≐ map g
```

[Matthes, 2009] used these "extensional functors" in the context of nested datatypes. *mapPresEE* for *Maybe*, *List*,... e.g. in Coq and Agda standard libs. Unlike *Functors*, *ExtFunctors compose* ! e.g. *mapPresId* for $F \circ G$:

```
mapPresId = λ x →
  mapF (mapG id) x ≡⟨ mapPresEEF _ _ mapPresIdG x ⟩
  mapF id x        ≡⟨ mapPresIdF x ⟩
  id x             QED
```

Not extensional: Reader

$$\mathsf{Reader} : \mathsf{Set} \to \mathsf{Set} \; ; \; \mathsf{Reader}\ A = \mathit{Env} \to A$$
$$\mathsf{mapReader} : \{A\ B : \mathsf{Set}\} \to (A \to B) \to (\mathit{Env} \to A) \to (\mathit{Env} \to B)$$
$$\mathsf{mapReader} = \_\circ\_$$

Not extensional: Reader

$$\mathsf{Reader} : \mathsf{Set} \to \mathsf{Set} \;;\; \mathsf{Reader}\ A = \mathit{Env} \to A$$
$$\mathsf{mapReader} : \{A\ B : \mathsf{Set}\} \to (A \to B) \to (\mathit{Env} \to A) \to (\mathit{Env} \to B)$$
$$\mathsf{mapReader} = \_\circ\_$$

Post-(and pre-)composition with any function preserves $\doteq$

$$\mathsf{postCompPres}\doteq : \forall\ \{A\ B\ C\}\ (h : B \to C)\ (f\ g : A \to B) \to f \doteq g \to h \circ f \doteq h \circ g$$
$$\mathsf{postCompPres}\doteq\ h\ f\ g\ f\doteq g\ a = \mathsf{cong}\ h\ (f\doteq g\ a)$$

Not extensional: Reader

```
Reader : Set → Set ; Reader A = Env → A
mapReader : {A B : Set} → (A → B) → (Env → A) → (Env → B)
mapReader = _∘_
```

Post-(and pre-)composition with any function preserves $\doteq$

```
postCompPres≐ : ∀ {A B C} (h : B → C) (f g : A → B) →
                f ≐ g → h ∘ f ≐ h ∘ g
postCompPres≐ h f g f≐g a = cong h (f≐g a)
```

Composition itself does not:

```
compNoPres≐ : ∀ {A B C : Set} (f g : B → C) →
              f ≐ g → f ∘_ ≐ g ∘_
compNoPres≐ f g f≐g = λ h → {!!} -- Hole : f ∘ h ≡ g ∘ h
```

Not extensional: Reader

```
Reader : Set → Set ; Reader A = Env → A
mapReader : {A B : Set} → (A → B) → (Env → A) → (Env → B)
mapReader = _∘_
```

Post-(and pre-)composition with any function preserves $\doteq$

```
postCompPres≐ : ∀ {A B C} (h : B → C) (f g : A → B) →
                f ≐ g → h ∘ f ≐ h ∘ g
postCompPres≐ h f g f≐g a = cong h (f≐g a)
```

Composition itself does not:

```
compNoPres≐ : ∀ {A B C : Set} (f g : B → C) →
              f ≐ g → f ∘_ ≐ g ∘_
compNoPres≐ f g f≐g = λ h → {!!} -- Hole : f ∘ h ≡ g ∘ h
```

Taking $h = id$ (and using η !) we get:

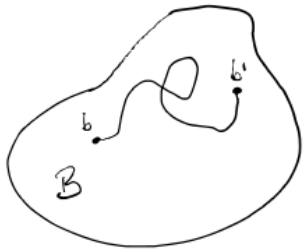
Fact

mapPresEE for Reader implies funExt for all functions on Env.

Can we characterize extensional functors?

Free path space:

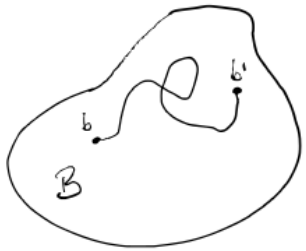
$$\mathsf{Path} : \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{Path}\ B = \Sigma\ (B \times B)\ \lambda\ (b\ ,\ b') \to b \equiv b'$$



Can we characterize extensional functors?

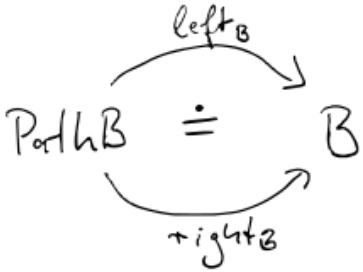
Free path space:

$$\mathsf{Path} : \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{Path}\ B = \Sigma\ (B \times B)\ \lambda\ (b\ ,\ b') \to b \equiv b'$$



Projections are extensionally equal

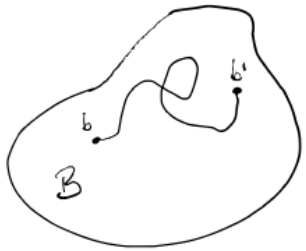
$$\mathsf{l{\doteq}r} : \mathsf{left} \doteq \mathsf{right}$$
$$\mathsf{l{\doteq}r}\ ((\_\ ,\ \_)\ **\ b{\equiv}b') = b{\equiv}b'$$



Can we characterize extensional functors?

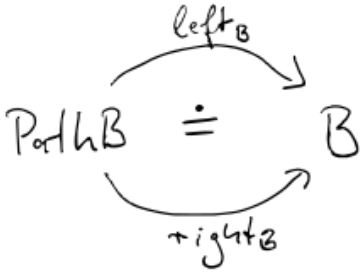
Free path space:

$$\begin{aligned}&\mathsf{Path} : \mathsf{Set} \to \mathsf{Set}\\&\mathsf{Path}\ B = \Sigma\ (B \times B)\ \lambda\ (b\ ,\ b') \to b \equiv b'\end{aligned}$$



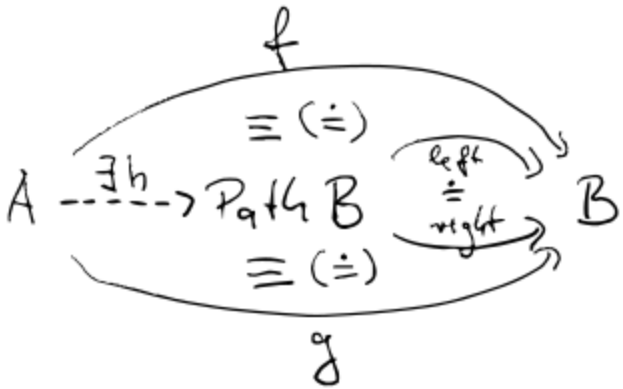
Projections are extensionally equal

$$\begin{aligned}&\mathsf{l{\doteq}r} : \mathsf{left} \doteq \mathsf{right}\\&\mathsf{l{\doteq}r}\ ((\_\ ,\ \_)\ **\ b{\equiv}b') = b{\equiv}b'\end{aligned}$$



Universal property:

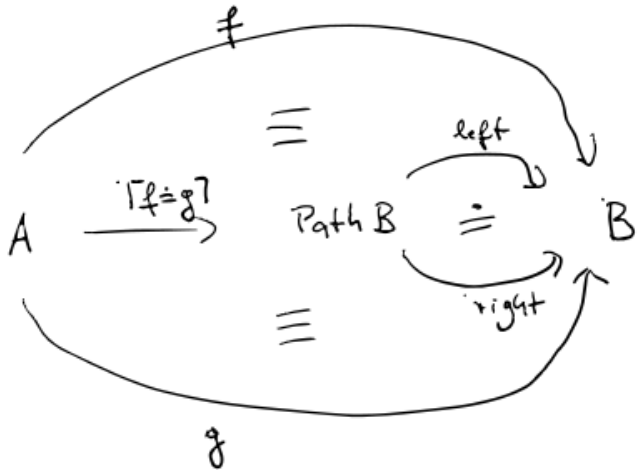
$$\begin{aligned}f \doteq g &\iff \exists h.(\mathit{left} \circ h \equiv f) \wedge (\mathit{right} \circ h \equiv g)\\&\iff \exists h.(\mathit{left} \circ h \doteq f) \wedge (\mathit{right} \circ h \doteq g)\end{aligned}$$



$$\begin{aligned}&\lceil\_\rceil : \forall\ \{A\}\ \{f\ g : A \to B\} \to f \doteq g \to\\&\qquad \Sigma\ (A \to \mathsf{Path}\ B)\\&\qquad\quad \lambda\ h \to (\mathsf{left} \circ h \equiv f) \times (\mathsf{right} \circ h \equiv g)\\&\lceil\ f{\doteq}g\ \rceil = (\lambda\ a \to (\_\ ,\ \_)\ **\ f{\doteq}g\ a)\ **\ \mathsf{refl}\ ,\ \mathsf{refl}\end{aligned}$$

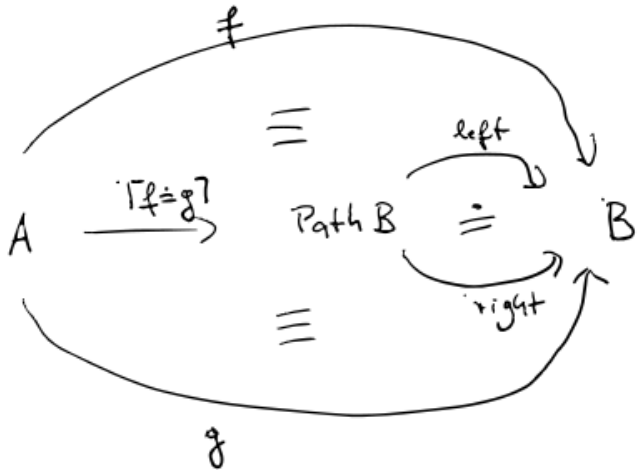
Characterization of extensional functors

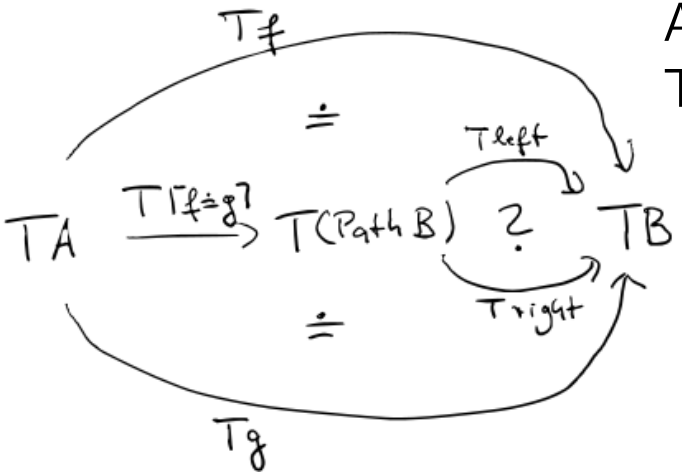
Suppose $T : Set \to Set$, *Functor* T.
Let $f, g : A \to B$ extensionally equal: $f \doteq g : f \doteq g$.



Characterization of extensional functors

Suppose $T : Set \to Set$, *Functor* T.
Let $f, g : A \to B$ extensionally equal: $f \doteq g : f \doteq g$.



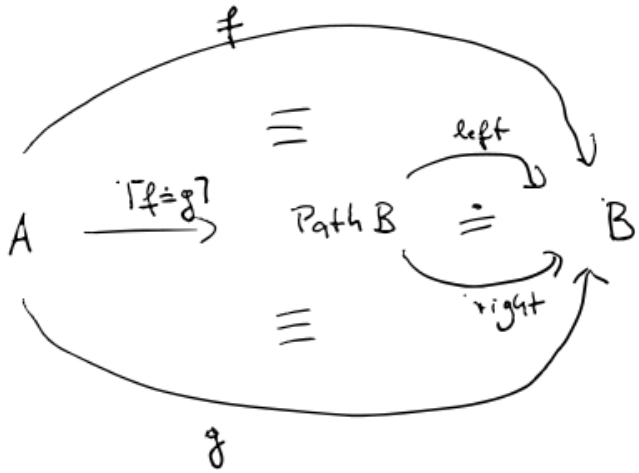


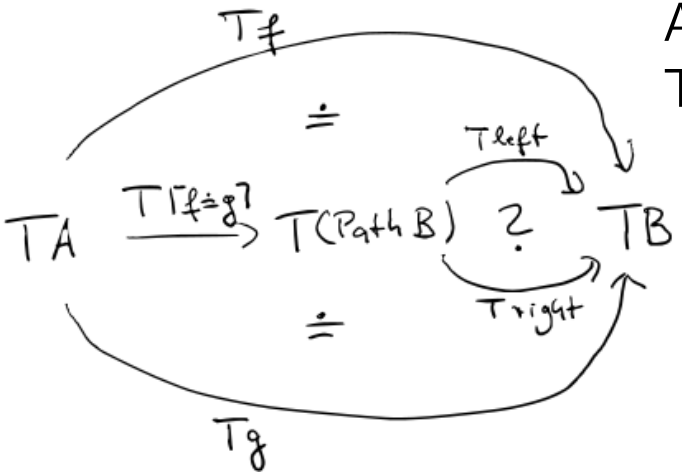
Apply T.
Triangles become $\doteq$:

```
mapT f ta                    ≡⟨ refl {- uses η ! -} ⟩
mapT (left ∘ ⌈ f≐g ⌉) ta ≡⟨ mapPresComp left p ta ⟩
(mapT left ∘ mapT ⌈ f≐g ⌉) ta
```

Characterization of extensional functors

Suppose $T : Set \to Set$, *Functor* T.
Let $f, g : A \to B$ extensionally equal: $f \doteq g : f \doteq g$.





Apply T.
Triangles become $\doteq$:

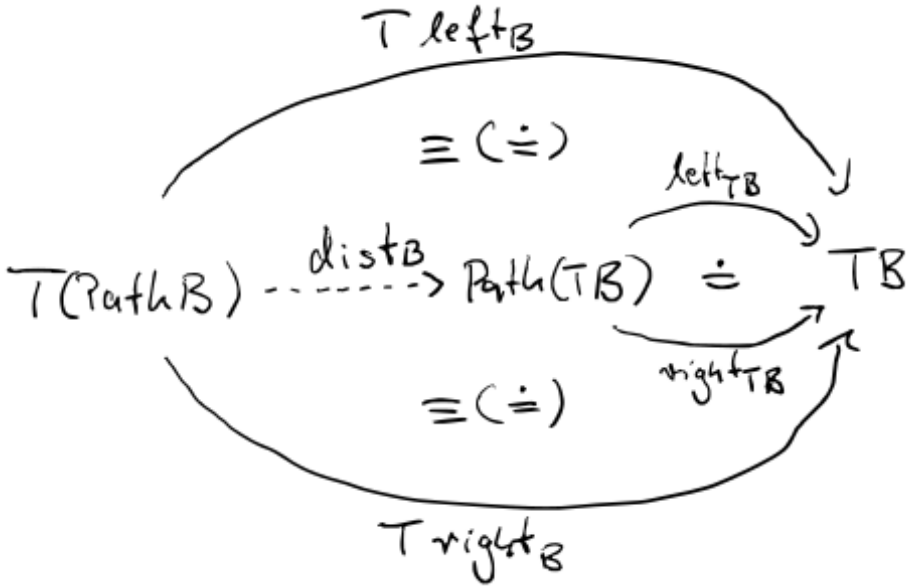
```
mapT f ta                    ≡⟨ refl {- uses η ! -} ⟩
mapT (left ∘ ⌈ f≐g ⌉) ta ≡⟨ mapPresComp left p ta ⟩
(mapT left ∘ mapT ⌈ f≐g ⌉) ta
```

Fact

$$T \text{ extensional} \iff \forall B.\, map_T\, left_B \doteq map_T\, right_B$$

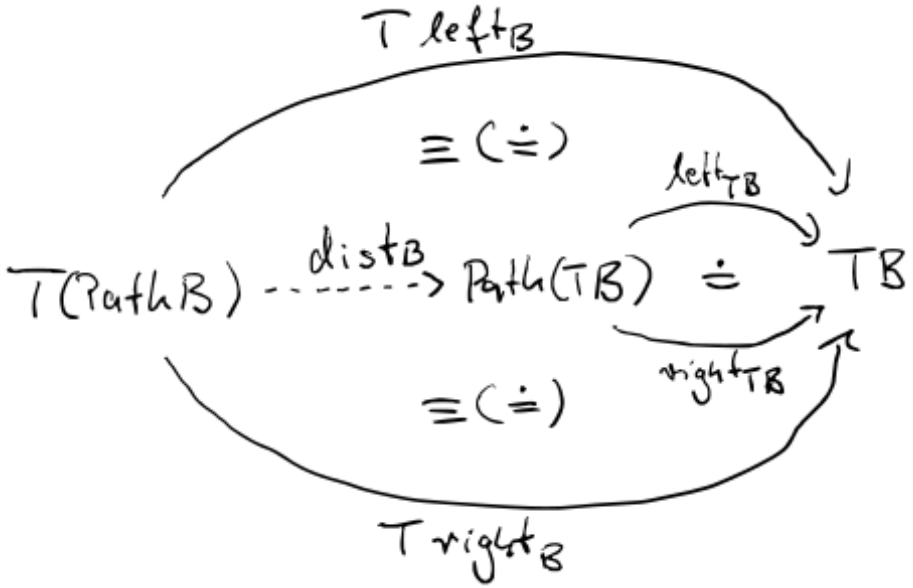
Traversables are extensional

Universal property for the "inner eye":



Traversables are extensional

Universal property for the "inner eye":

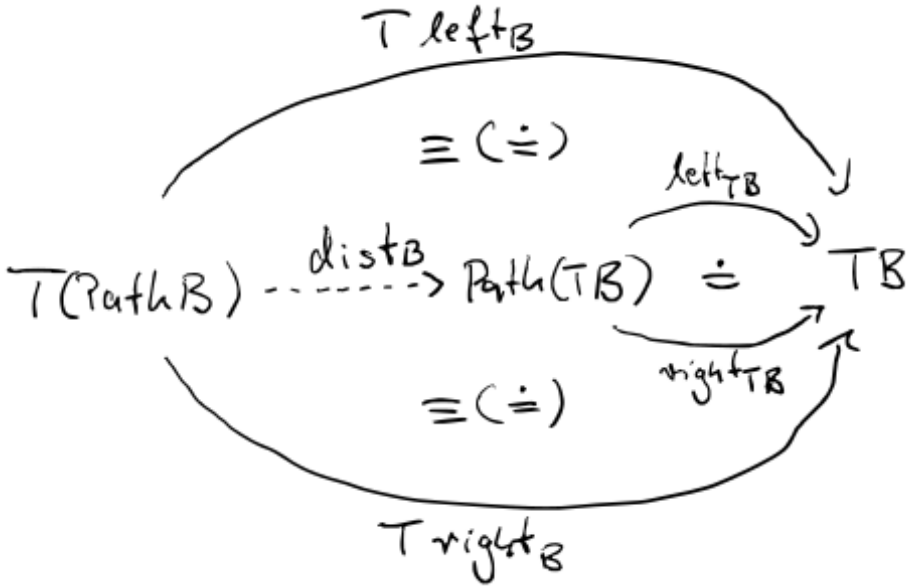


Fact

$$T \text{ extensional} \iff \forall B\, \exists\, dist_B : (T \circ Path)B \to (Path \circ T)B \text{ s.t.}$$
$$left_B \circ dist_B \doteq T\ left_B \text{ and } right_B \circ dist_B \doteq T\ right_B$$

Traversables are extensional

Universal property for the "inner eye":



Fact

$$T \text{ extensional} \iff \forall B\, \exists\, dist_B : (T \circ Path)B \to (Path \circ T)B \text{ s.t.}$$
$$left_B \circ dist_B \doteq T\ left_B \text{ and } right_B \circ dist_B \doteq T\ right_B$$

Corollary

Any traversable functor ([McBride and Paterson, 2008], [Jaskelioff and Rypacek, 2012]) is extensional.

Traversables are extensional

```
record Applicative (F : Set → Set) : Set₁ where
  infixl 30 _⊛_
  field
    pure : ∀ {A} → A → F A
    _⊛_ : ∀ {A B} → F (A → B) → F A → F B
    -- 4 axioms ...
    -- had to add:
    appMapPresEE : ∀ {A B} {f g : A → B} →
                   f ≐ g → pure f ⊛_ ≐ pure g ⊛_
```

```
record TraversableD : Set (lsuc ℓ) where
  field
    dist     :  DistType
    functor  :  Functor T
  mapT : MapType
  mapT = map {{functor}}
  field
    distId      : DistId     dist
    distComp    : DistComp dist
    distNat     : DistNat    dist   mapT
    distNatFam  : DistNatFam dist mapT
```

```
DistType = ∀ F {{FApp : Applicative F}} {A} →
           T (F A) → F (T A)
```

```
DistId     dist =     -- unitarity [J&R, def. 3.3]
  ∀ {A} → dist id {{idApp}} {A} ≐ id
```

```
DistComp dist =     -- linearity [J&R, def. 3.3]
  ∀ F G {{FA : App F}} {{GA : App G}} {A} →
  dist (F ∘ G) {{compApp F G}} {A} ≐
  map {{appIsFun F}} (dist G) ∘ dist F
```

```
DistNat dist mapT = -- naturality [J&R, def. 3.3]
  ∀ F G {{FA : App F}} {{GA : App G}}
  (τ : F -·¿ G) {{τANT : AppNatTrans F G τ}} {A} →
  τ ∘ dist F {A} ≐ dist G {A} ∘ mapT τ
```

```
DistNatFam dist mapT = -- family of ntfs [J&R, def. 3.3]
  ∀ F {{FA : App F}} {A B} (f : A → B) →
  let mapF = map {{appIsFun F}}
  in  mapF (mapT f) ∘ dist F ≐ dist F ∘ mapT (mapF f)
```

Traversables are extensional

- *Path* and *Id* are applicative functors.

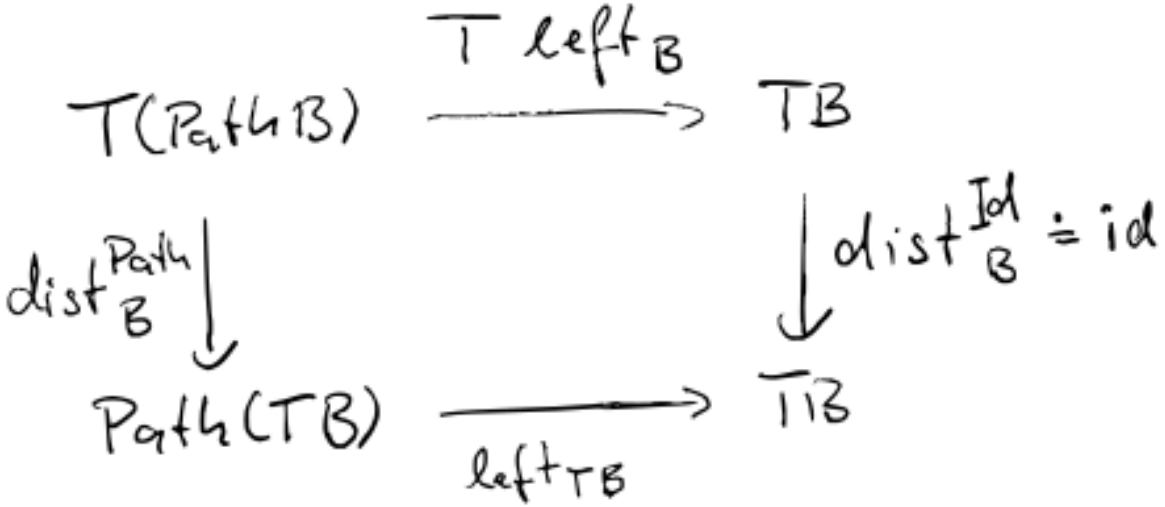
Traversables are extensional

- *Path* and *Id* are applicative functors.
- *left* and *right* are "applicative natural transformations" $\mathit{Path} \dot{\to} \mathit{Id}$

Traversables are extensional

- *Path* and *Id* are applicative functors.
- *left* and *right* are "applicative natural transformations" $Path \dot{\to} Id$

```
(left ∘ dist Path {{pathApp}}) tpa
  ≡⟨ distNat Path id {{pathApp}} {{idApp}} left {{leftANT}} _ ⟩
(dist id {{idApp}} ∘ mapT left) tpa
  ≡⟨ distId _ ⟩
mapT left tpa
  QED
```



Conclusion

- Agda Formalization
  - (Extensional) Applicatives and lax monoidal functors, proved equivalent.
  - Applicatives compose - proofs were long, now we use normalization of "applicative expressions".
  - Applicative lifting of n-ary functions preserves extensional equality $\overset{n}{=}$.
  - 4 interfaces for traversable functors (w.r.t. *extensional* applicatives): (dist, consume, traverse, all).

Conclusion

- Agda Formalization
  - (Extensional) Applicatives and lax monoidal functors, proved equivalent.
  - Applicatives compose - proofs were long, now we use normalization of "applicative expressions".
  - Applicative lifting of n-ary functions preserves extensional equality $\overset{n}{=}$.
  - 4 interfaces for traversable functors (w.r.t. *extensional* applicatives): (dist, consume, traverse, all).
  - Loose end: $\mathit{lift}\,(F \circ G) = \mathit{lift}\,F \circ \mathit{lift}\,G$?

Conclusion

- Agda Formalization
  - (Extensional) Applicatives and lax monoidal functors, proved equivalent.
  - Applicatives compose - proofs were long, now we use normalization of "applicative expressions".
  - Applicative lifting of n-ary functions preserves extensional equality $\overset{n}{=}$.
  - 4 interfaces for traversable functors (w.r.t. *extensional* applicatives): (dist, consume, traverse, all).
  - Loose end: $\mathit{lift}\,(F \circ G) = \mathit{lift}\,F \circ \mathit{lift}\,G$?

- Questions
  - Connections?
  - Non-traversable functor with *mapPresEE*?

Conclusion

- Agda Formalization
  - (Extensional) Applicatives and lax monoidal functors, proved equivalent.
  - Applicatives compose - proofs were long, now we use normalization of "applicative expressions".
  - Applicative lifting of n-ary functions preserves extensional equality $\overset{n}{=}$.
  - 4 interfaces for traversable functors (w.r.t. *extensional* applicatives): (dist, consume, traverse, all).
  - Loose end: $\mathit{lift}\,(F \circ G) = \mathit{lift}\,F \circ \mathit{lift}\,G$?

- Questions
  - Connections?
  - Non-traversable functor with *mapPresEE*?

Thank you!

References

Botta, N., Brede, N., Jansson, P., and Richter, T. (2021).
Extensional equality preservation and verified generic programming.
J. Funct. Program., 31.
arXiv:2008.02123.

Brede, N. and Botta, N. (2021).
On the correctness of monadic backward induction.
J. Funct. Program., 31:e26.

Jaskelioff, M. and Rypacek, O. (2012).
An investigation of the laws of traversals.
In Chapman, J. and Levy, P. B., editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012*, volume 76 of *EPTCS*, pages 40–49.

Matthes, R. (2009).
An induction principle for nested datatypes in intensional type theory.
Journal of Functional Programming, 19(3-4):439–468.

McBride, C. and Paterson, R. (2008).
Applicative programming with effects.
J. Funct. Program., 18(1):1–13.