Vinyl: Records in Haskell and Type Theory

Jon Sterling
jonmsterling.com

August 12, 2014

Records in GHC 7.8

Records in GHC 7.8

- Haskell records are nominally typed

Records in GHC 7.8

- Haskell records are nominally typed
- They may not share field names

Records in GHC 7.8

- Haskell records are nominally typed
- They may not share field names

data R = R { x :: X }

Records in GHC 7.8

- Haskell records are nominally typed
- They may not share field names

```
data R = R { x :: X }
data R' = R' { x :: X } -- ^ Error
```

Structural typing

Structural typing

- Sharing field names and accessors

Structural typing

- Sharing field names and accessors
- Record types may be characterized *structurally*

Row polymorphism

Row polymorphism

How do we express the type of a function which adds a field to a record?

Row polymorphism

How do we express the type of a function which adds a field to a record?

$$\frac{x : \{foo : A\}}{f(x) : \{foo : A, bar : B\}}$$

Row polymorphism

How do we express the type of a function which adds a field to a record?

$$\frac{x : \{foo : A; \vec{rs}\}}{f(x) : \{foo : A, bar : B; \vec{rs}\}}$$

Roll your own in Haskell

Roll your own in Haskell

data (s :: Symbol) ::: (t :: $*$) = Field

Roll your own in Haskell

data (s :: Symbol) ::: (t :: $*$) = Field

data Rec :: [$*$] $\rightarrow$ $*$ **where**

Roll your own in Haskell

```
data (s :: Symbol) ::: (t :: *) = Field

data Rec :: [*] -> * where
  RNil :: Rec '[]
```

Roll your own in Haskell

```
data (s :: Symbol) ::: (t :: *) = Field

data Rec :: [*] → * where
  RNil :: Rec '[]
  (:&) :: !t → !(Rec rs) → Rec ((s ::: t) ': rs)
```

Roll your own in Haskell

```
data (s :: Symbol) ::: (t :: *) = Field

data Rec :: [*] -> * where
  RNil :: Rec '[]
  (:&) :: !t -> !(Rec rs) -> Rec ((s ::: t) ': rs)


class s ∈ (rs :: [*])
```

Roll your own in Haskell

```
data (s :: Symbol) ::: (t :: ∗) = Field

data Rec :: [∗] → ∗ where
  RNil :: Rec '[]
  (:&) :: !t → !(Rec rs) → Rec ((s ::: t) ': rs)


class s ∈ (rs :: [∗])
class ss ⊆ (rs :: [∗]) where
  cast :: Rec rs → Rec ss
```

Roll your own in Haskell

```
data (s :: Symbol) ::: (t :: *) = Field

data Rec :: [*] → * where
  RNil :: Rec '[]
  (:&) :: !t → !(Rec rs) → Rec ((s ::: t) ': rs)


class s ∈ (rs :: [*])
class ss ⊆ (rs :: [*]) where
  cast :: Rec rs → Rec ss
(=:) : s ::: t → t → Rec '[s ::: t]
```

Roll your own in Haskell

```
data (s :: Symbol) ::: (t :: ∗) = Field

data Rec :: [∗] → ∗ where
  RNil :: Rec '[]
  (:&) :: !t → !(Rec rs) → Rec ((s ::: t) ': rs)


class s ∈ (rs :: [∗])
class ss ⊆ (rs :: [∗]) where
  cast :: Rec rs → Rec ss
(=:) : s ::: t → t → Rec '[s ::: t]
(⊕) : Rec ss → Rec ts → Rec (ss ++ ts)
```

Roll your own in Haskell

```
data (s :: Symbol) ::: (t :: *) = Field

data Rec :: [*] → * where
  RNil :: Rec '[]
  (:&) :: !t → !(Rec rs) → Rec ((s ::: t) ': rs)


class s ∈ (rs :: [*])
class ss ⊆ (rs :: [*]) where
  cast :: Rec rs → Rec ss
(=:) : s ::: t → t → Rec '[s ::: t]
(⊕) : Rec ss → Rec ts → Rec (ss ++ ts)
lens : s ::: t ∈ rs ⇒ s ::: t → Lens' (Rec rs) t
```

Roll your own in Haskell

Roll your own in Haskell

```
f :: Rec (”foo” ::: A ’: rs)
   → Rec (”bar” ::: B ’: ”foo” ::: A ’: rs)
```

Why be stringly typed?

Why be stringly typed?

- Let's generalize our key space

Old

We relied on a single representation for keys as pairs of strings and types.

data Rec :: $[*] \rightarrow *$ **where**
  RNil :: Rec '[]
  (:&) :: !t $\rightarrow$!(Rec rs) $\rightarrow$ Rec ((s ::: t) ': rs)

Proposed

We put the keys in an arbitrary type (kind) and describe their semantics with a function el (for *elements*).

data Rec :: $[*] \to *$ **where**
  RNil :: Rec '[]
  (:&) :: !t $\to$!(Rec rs) $\to$ Rec ((s ::: t) ': rs)

Proposed

We put the keys in an arbitrary type (kind) and describe their semantics with a function el (for *elements*).

```
data Rec :: (el :: k → ∗) → (rs :: [k]) → ∗ where
  RNil :: Rec el '[]
  (:&) :: ∀(r :: k) (rs' :: [k]). !(el r) → !(Rec el rs) → Rec (r ': rs')
```

Proposed

We put the keys in an arbitrary type (kind) and describe their semantics with a function el (for *elements*).

data Rec :: (k $\rightarrow *$) $\rightarrow$ [k] $\rightarrow *$ **where**
  RNil :: Rec el '[]
  (:&) :: !(el r) $\rightarrow$!(Rec el rs) $\rightarrow$ Rec (r ': rs)

Actual

Type families are *not* functions, but in many cases can simulate them using Richard Eisenberg's technique outlined in *Defunctionalization for the win*.

data TyFun :: $* \to * \to *$

Actual

Type families are *not* functions, but in many cases can simulate them using Richard Eisenberg's technique outlined in *Defunctionalization for the win*.

data TyFun :: $* \to * \to *$
type family (f :: TyFun k l $\to *$) $ (x :: k) :: l

Actual

Type families are *not* functions, but in many cases can simulate them using Richard Eisenberg's technique outlined in *Defunctionalization for the win*.

```
data TyFun :: * → * → *
type family (f :: TyFun k l → *) $ (x :: k) :: l

data Rec :: (TyFun k * → *) → [k] → * where
  RNil :: Rec el '[]
  (:&) :: !(el $ r) → !(Rec el rs) → Rec el (r ': rs)
```

Example

```
data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label
```

Example

```
data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label

data SLabel :: Label → ∗ where
  SHome :: SLabel Home
  SWork :: SLabel Work
```

Example

```
data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label

data SLabel :: Label → ∗
```

Example

```
data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label

data SLabel :: Label → ∗
data SAddrKeys :: AddrKeys → ∗ where
  SName :: SAddrKeys Name
  SPhone :: SLabel l → SAddrKeys (Phone l)
  SEmail :: SLabel l → SAddrKeys (Email l)
```

Example

```
data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label

data SLabel :: Label → ∗
data SAddrKeys :: AddrKeys → ∗
```

Example

```
data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label

data SLabel :: Label → ∗
data SAddrKeys :: AddrKeys → ∗

data ElAddr :: (TyFun AddrKeys ∗) → ∗ where
  ElAddr :: ElAddr el
```

Example

```
data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label

data SLabel :: Label → ∗
data SAddrKeys :: AddrKeys → ∗

data ElAddr :: (TyFun AddrKeys ∗) → ∗ where
  ElAddr :: ElAddr el

type instance ElAddr $ Name = String
type instance ElAddr $ (Phone l) = [ℕ]
type instance ElAddr $ (Email l) = String

type AddrRec rs = Rec ElAddr rs
```

Sugared example

```
import Data.Singletons as S
```

Sugared example

```
import Data.Singletons as S

data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label
S.genSingletons [ ''Label, ''AddrKeys ]
```

Sugared example

```
import Data.Singletons as S

data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label
S.genSingletons [ ''Label, ''AddrKeys ]

makeUniverse ''AddrKeys "ElAddr"
```

Sugared example

```
import Data.Singletons as S

data Label = Home | Work
data AddrKeys = Name | Phone Label | Email Label
S.genSingletons [ ''Label, ''AddrKeys ]

makeUniverse ''AddrKeys "ElAddr"

type instance ElAddr $ Name = String
type instance ElAddr $ (Phone l) = [ℕ]
type instance ElAddr $ (Email l) = String

type AddrRec rs = Rec ElAddr rs
```

Example records

bob :: AddrRec [Name, Email Work]
bob = SName =: ”Robert␣W.␣Harper”
⊕ SEmail SWork =: ”rwh@cs.cmu.edu”

Example records

```
bob :: AddrRec [Name, Email Work]
bob = SName =: "Robert␣W.␣Harper"
      ⊕ SEmail SWork =: "rwh@cs.cmu.edu"

jon :: AddrRec [Name, Email Work, Email Home]
jon = SName =: "Jon␣M.␣Sterling"
    ⊕ SEmail SWork =: "jon@fobo.net"
    ⊕ SEmail SHome =: "jon@jonmsterling.com"
```

Recovering HList

Recovering HList

```
data Id :: (TyFun k k) → ∗ where
type instance Id $ x = x
```

Recovering HList

```
data Id :: (TyFun k k) → ∗ where
type instance Id $ x = x

type HList rs = Rec Id rs
```

Recovering HList

```
data Id :: (TyFun k k) → ∗ where
type instance Id $ x = x

type HList rs = Rec Id rs

ex :: HList [ℤ, Bool, String]
ex = 34 :& True :& ”vinyl” :& RNil
```

Validating records

validateName :: **String** $\rightarrow$ **Either** Error **String**
validateEmail :: **String** $\rightarrow$ **Either** Error **String**
validatePhone :: $[\mathbb{N}]$ $\rightarrow$ **Either** Error $[\mathbb{N}]$

Validating records

validateName :: **String** $\rightarrow$ **Either** Error **String**
validateEmail :: **String** $\rightarrow$ **Either** Error **String**
validatePhone :: $[\mathbb{N}]$ $\rightarrow$ **Either** Error $[\mathbb{N}]$

unnnnnhhh...

Validating records

validateName :: **String** $\rightarrow$ **Either** Error **String**
validateEmail :: **String** $\rightarrow$ **Either** Error **String**
validatePhone :: $[\mathbb{N}]$ $\rightarrow$ **Either** Error $[\mathbb{N}]$

unnnnnhhh...

validateContact
  :: AddrRec [Name, Email Work]
  $\rightarrow$ **Either** Error (AddrRec [Name, Email Work])

Welp.

Effects inside records

```
data Rec :: (TyFun k ∗ → ∗) → [k] → ∗ where
  RNil :: Rec el '[]
  (:&) :: !(el $ r) → !(Rec el rs) → Rec el (r ': rs)
```

Effects inside records

```
data Rec :: (TyFun k ∗ → ∗) → (∗ → ∗) → [k] → ∗ where
  RNil :: Rec el f '[]
  (:&) :: !(f (el $ r)) → !(Rec el f rs) → Rec el f (r ': rs)
```

Effects inside records

```
data Rec :: (TyFun k ∗ → ∗) → (∗ → ∗) → [k] → ∗ where
  RNil :: Rec el f '[]
  (:&) :: !(f (el $ r)) → !(Rec el f rs) → Rec el f (r ': rs)
```

Effects inside records

```
data Rec :: (TyFun k ∗ → ∗) → (∗ → ∗) → [k] → ∗ where
  RNil :: Rec el f '[]
  (:&) :: !(f (el $ r)) → !(Rec el f rs) → Rec el f (r ': rs)

(=:) : Applicative f ⇒ sing r → el $ r → Rec el f '[r]
k =: x = pure x :& RNil
```

Effects inside records

```
data Rec :: (TyFun k ∗ → ∗) → (∗ → ∗) → [k] → ∗ where
  RNil :: Rec el f '[]
  (:&) :: !(f (el $ r)) → !(Rec el f rs) → Rec el f (r ': rs)

(=:) : Applicative f ⇒ sing r → el $ r → Rec el f '[r]
k =: x = pure x :& RNil

(⇚): sing r → f (el $ r) → Rec el f '[r]
k ⇚ x = x :& RNil
```

Compositional validation

type Validator a = a $\rightarrow$ **Either** Error a

Compositional validation

```
newtype Lift o f g x = Lift { runLift :: f x `o` g x }
type Validator = Lift (→) Identity (Either Error)
```

Compositional validation

```
newtype Lift o f g x = Lift { runLift :: f x `o` g x }
type Validator = Lift (→) Identity (Either Error)
```

Compositional validation

```
newtype Lift o f g x = Lift { runLift :: f x `o` g x }
type Validator = Lift (→) Identity (Either Error)

validateName :: AddrRec Validator '[Name]
validatePhone :: ∀l. AddrRec Validator '[Phone l]
validateEmail :: ∀l. AddrRec Validator '[Email l]
```

Compositional validation

```
newtype Lift o f g x = Lift { runLift :: f x `o` g x }
type Validator = Lift (→) Identity (Either Error)

validateName :: AddrRec Validator '[Name]
validatePhone :: ∀l. AddrRec Validator '[Phone l]
validateEmail :: ∀l. AddrRec Validator '[Email l]

type TotalContact =
  [ Name, Email Home, Email Work
  , Phone Home, Phone Work ]
```

Compositional validation

```
newtype Lift o f g x = Lift { runLift :: f x `o` g x }
type Validator = Lift (→) Identity (Either Error)

validateName :: AddrRec Validator '[Name]
validatePhone :: ∀l. AddrRec Validator '[Phone l]
validateEmail :: ∀l. AddrRec Validator '[Email l]

type TotalContact =
  [ Name, Email Home, Email Work
  , Phone Home, Phone Work ]

validateContact :: AddrRec Validator TotalContact
validateContact = validateName
                ⊕ validateEmail ⊕ validateEmail
                ⊕ validatePhone ⊕ validatePhone
```

Record effect operators

Record effect operators

$$(\circledcirc) :: \text{Rec el (Lift } (\rightarrow) \text{ f g) rs} \rightarrow \text{Rec el f rs} \rightarrow \text{Rec el g rs}$$

Record effect operators

$(\circledcirc)$ :: Rec el (Lift $(\rightarrow)$ f g) rs $\rightarrow$ Rec el f rs $\rightarrow$ Rec el g rs

rtraverse :: Applicative h $\Rightarrow$
  ($\forall$ x. f x $\rightarrow$ h (g x)) $\rightarrow$
  Rec el f rs $\rightarrow$ h (Rec el g rs)

Record effect operators

(⊛) :: Rec el (Lift $(\rightarrow)$ f g) rs $\rightarrow$ Rec el f rs $\rightarrow$ Rec el g rs

rtraverse :: Applicative h $\Rightarrow$
  ($\forall$ x. f x $\rightarrow$ h (g x)) $\rightarrow$
  Rec el f rs $\rightarrow$ h (Rec el g rs)

rdist :: Applicative f $\Rightarrow$ Rec el f rs $\rightarrow$ f (Rec el Identity rs)
rdist = rtraverse (fmap Identity)

Compositional validation

```
bob :: AddrRec [Name, Email Work]
validateContact :: AddrRec Validator TotalContact
```

Compositional validation

```
bob :: AddrRec [Name, Email Work]
validateContact :: AddrRec Validator TotalContact

bobValid :: AddrRec (Either Error) [Name, Email Work]
bobValid = cast validateContact ⊛ bob
```

Compositional validation

```
bob :: AddrRec [Name, Email Work]
validateContact :: AddrRec Validator TotalContact

bobValid :: AddrRec (Either Error) [Name, Email Work]
bobValid = cast validateContact ⊛ bob

validBob :: Either Error (AddrRec Identity [Name, Email Work])
validBob = rdist bobValid
```

Laziness as an effect

Laziness as an effect

```
newtype Identity a = Identity { runIdentity :: a }
data Thunk a = Thunk { unThunk :: a }
```

Laziness as an effect

```
newtype Identity a = Identity { runIdentity :: a }
data Thunk a = Thunk { unThunk :: a }

type PlainRec el rs = Rec el Identity rs
```

Laziness as an effect

```
newtype Identity a = Identity { runIdentity :: a }
data Thunk a = Thunk { unThunk :: a }

type PlainRec el rs = Rec el Identity rs
type LazyRec el rs = Rec el Thunk rs
```

Concurrent records with Async

Concurrent records with Async

```
fetchName :: AddrRec IO '[Name]
fetchName = SName ⇦ runDB nameQuery
```

Concurrent records with Async

```
fetchName :: AddrRec IO '[Name]
fetchName = SName ⇚ runDB nameQuery

fetchWorkEmail :: AddrRec IO '[Email Work]
fetchWorkEmail = SEmail SWork ⇚ runDB emailQuery
```

Concurrent records with Async

```
fetchName :: AddrRec IO '[Name]
fetchName = SName ⇚ runDB nameQuery

fetchWorkEmail :: AddrRec IO '[Email Work]
fetchWorkEmail = SEmail SWork ⇚ runDB emailQuery

fetchBob :: AddrRec IO [Name, Email Work]
fetchBob = fetchName ⊕ fetchWorkEmail
```

Concurrent records with Async

Concurrent records with Async

```
newtype Concurrently a
  = Concurrently { runConcurrently :: IO a }
```

Concurrent records with Async

```
newtype Concurrently a
  = Concurrently { runConcurrently :: IO a }

(<$>) :: (∀ a. f a → g a) → Rec el f rs → Rec el g rs
```

Concurrent records with Async

```
newtype Concurrently a
  = Concurrently { runConcurrently :: IO a }

(<$>) :: (∀ a. f a → g a) → Rec el f rs → Rec el g rs

bobConcurrently :: Rec el Concurrently [Name, Email Work]
bobConcurrently = Concurrently <$> fetchBob
```

Concurrent records with Async

```
newtype Concurrently a
  = Concurrently { runConcurrently :: IO a }

(<$>) :: (∀ a. f a → g a) → Rec el f rs → Rec el g rs

bobConcurrently :: Rec el Concurrently [Name, Email Work]
bobConcurrently = Concurrently <$> fetchBob

concurrentBob :: IO (PlainRec el [Name, Email Work])
concurrentBob = runConcurrently (rdist bobConcurrently)
```

Type Theoretic Semantics for Records

Universes à la Tarski

Universes à la Tarski

- A type $\mathcal{U}$ of **codes** for types.

Universes à la Tarski

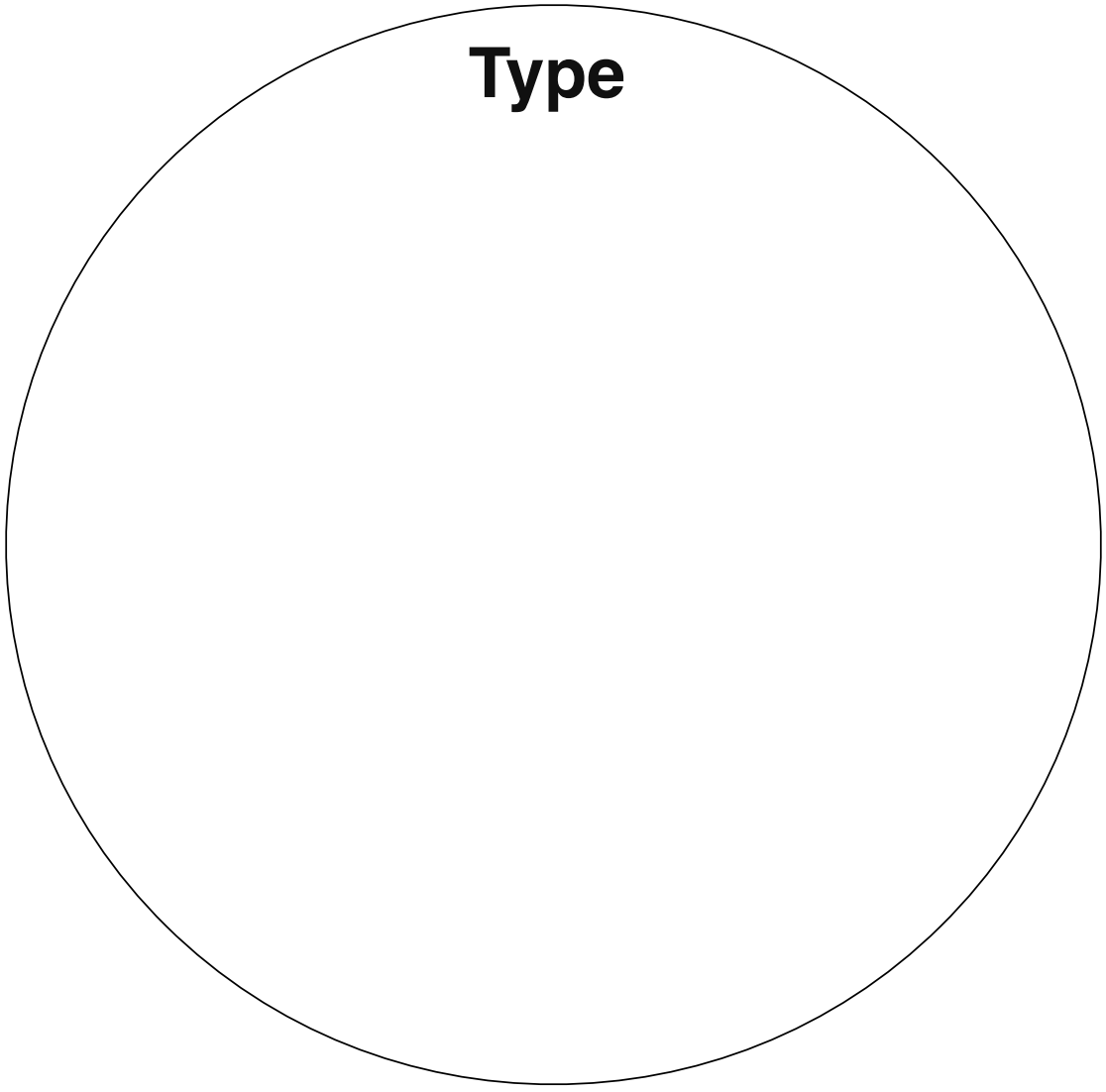
- A type $\mathcal{U}$ of **codes** for types.
- Function $El_{\mathcal{U}} : \mathcal{U} \to$ Type.

Universes à la Tarski

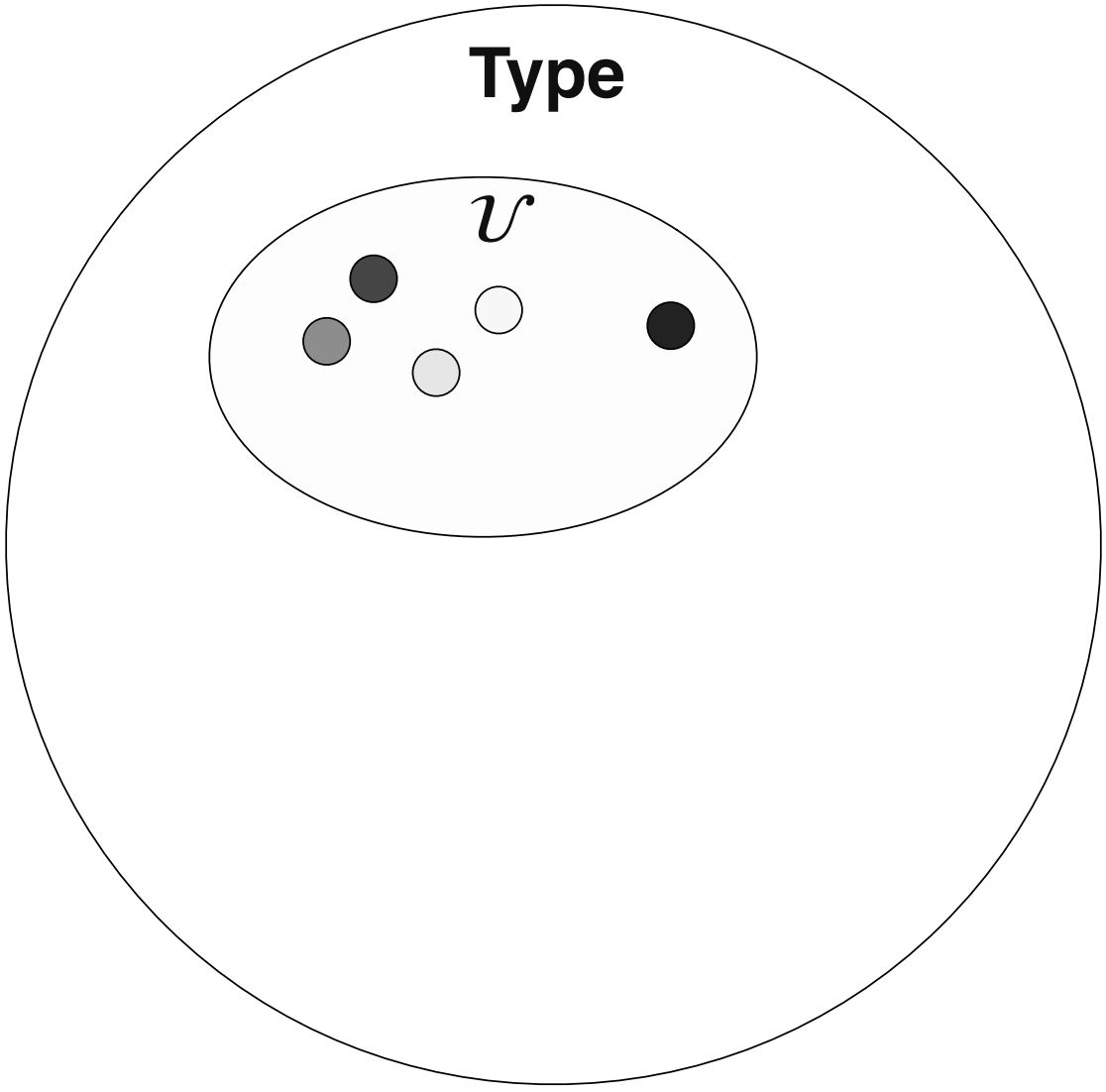
- A type $\mathcal{U}$ of **codes** for types.
- Function $El_{\mathcal{U}} : \mathcal{U} \to$ Type.

$$\frac{\Gamma \vdash s : \mathcal{U}}{\Gamma \vdash El_{\mathcal{U}}(s) : \text{Type}}$$

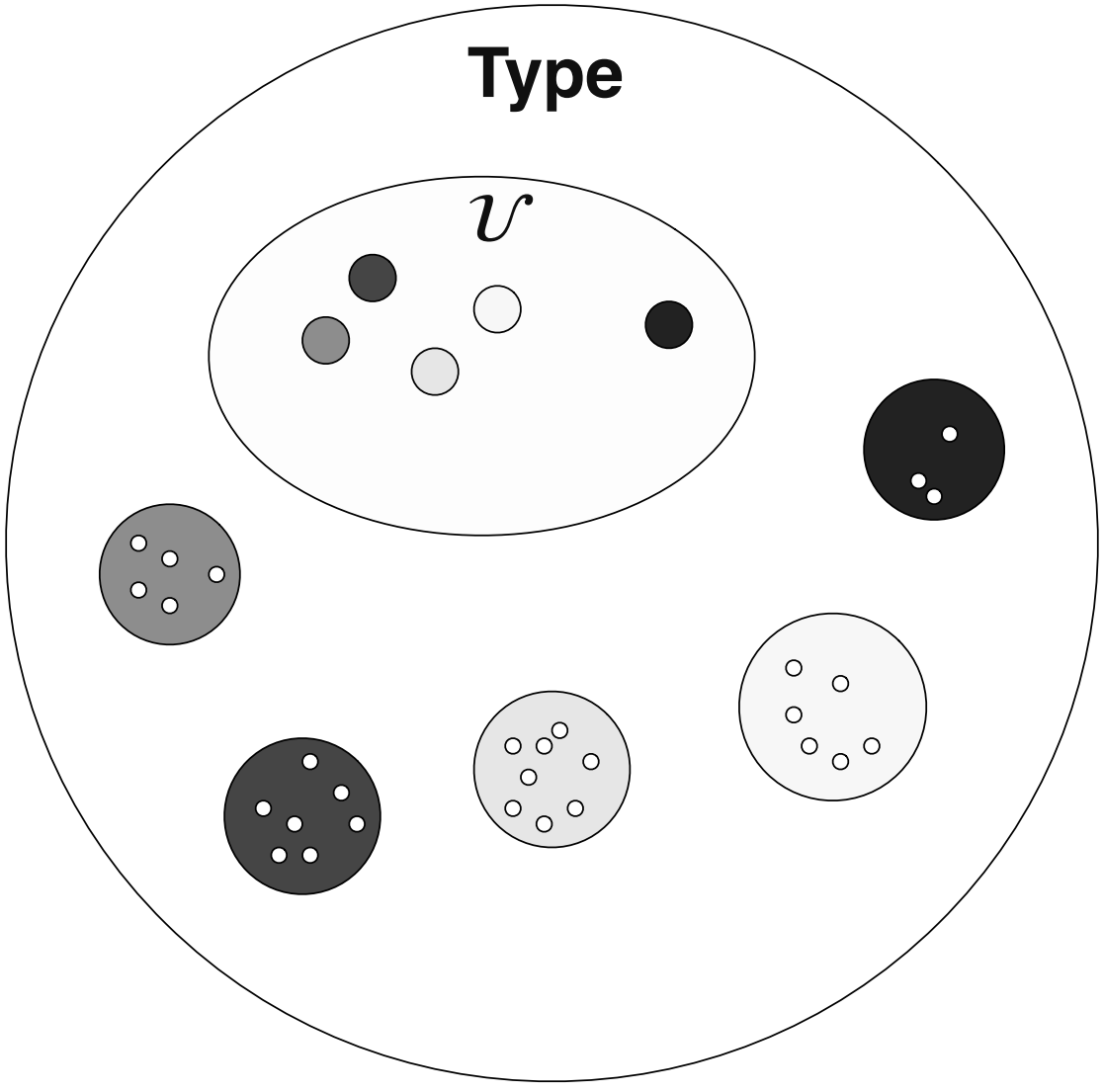
Universes à la Tarski



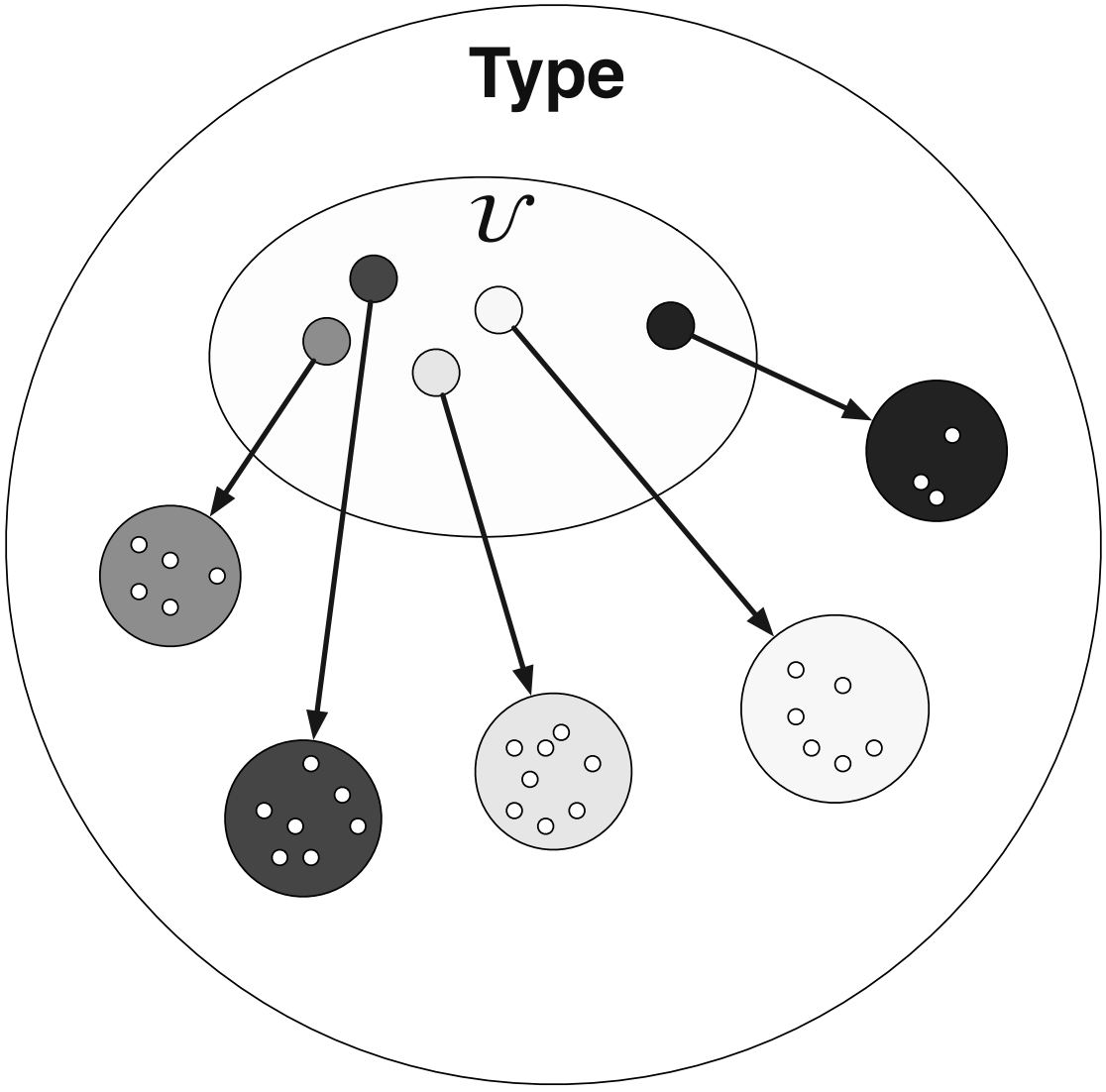
Universes à la Tarski



Universes à la Tarski



Universes à la Tarski

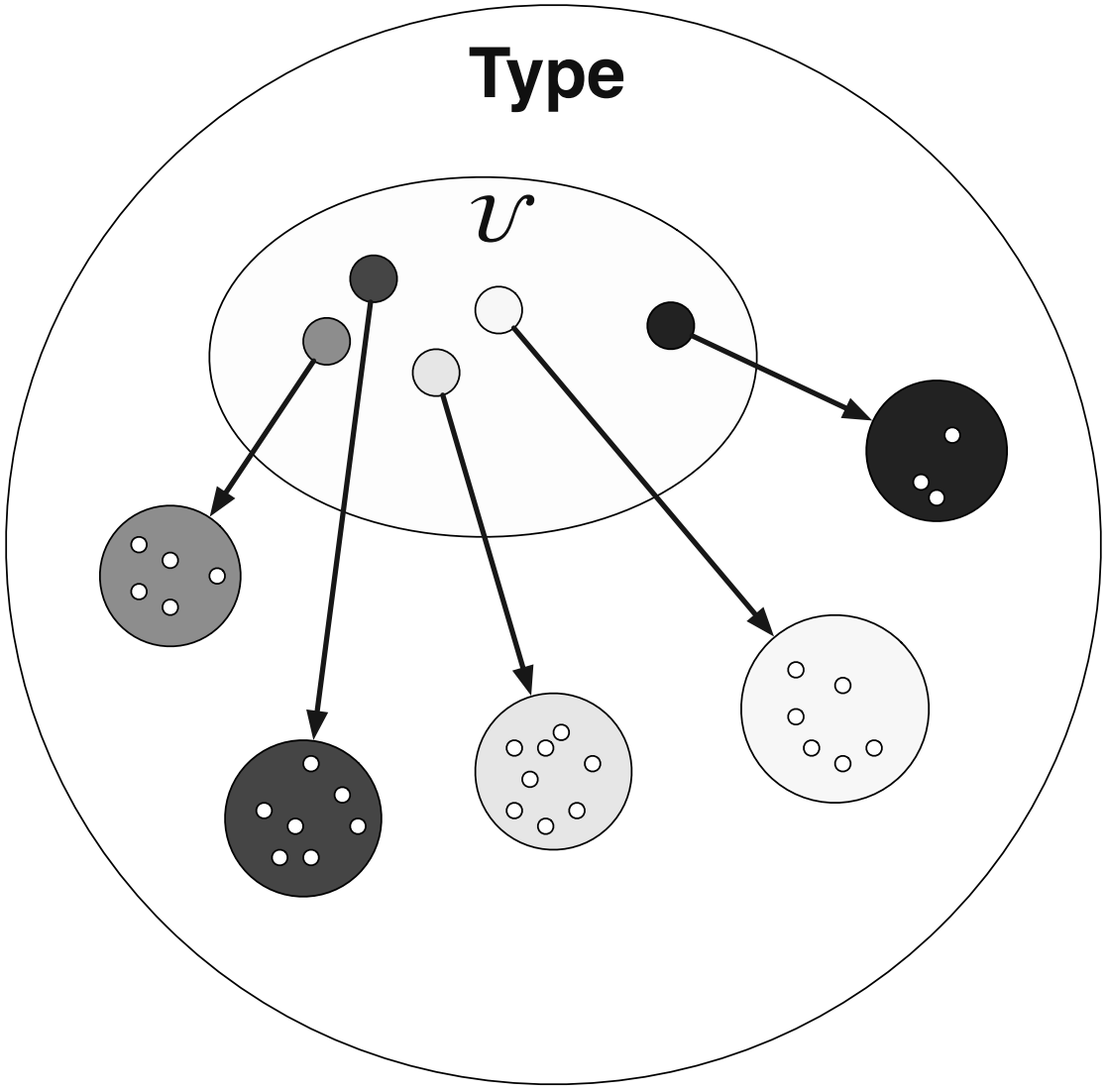


Records as products

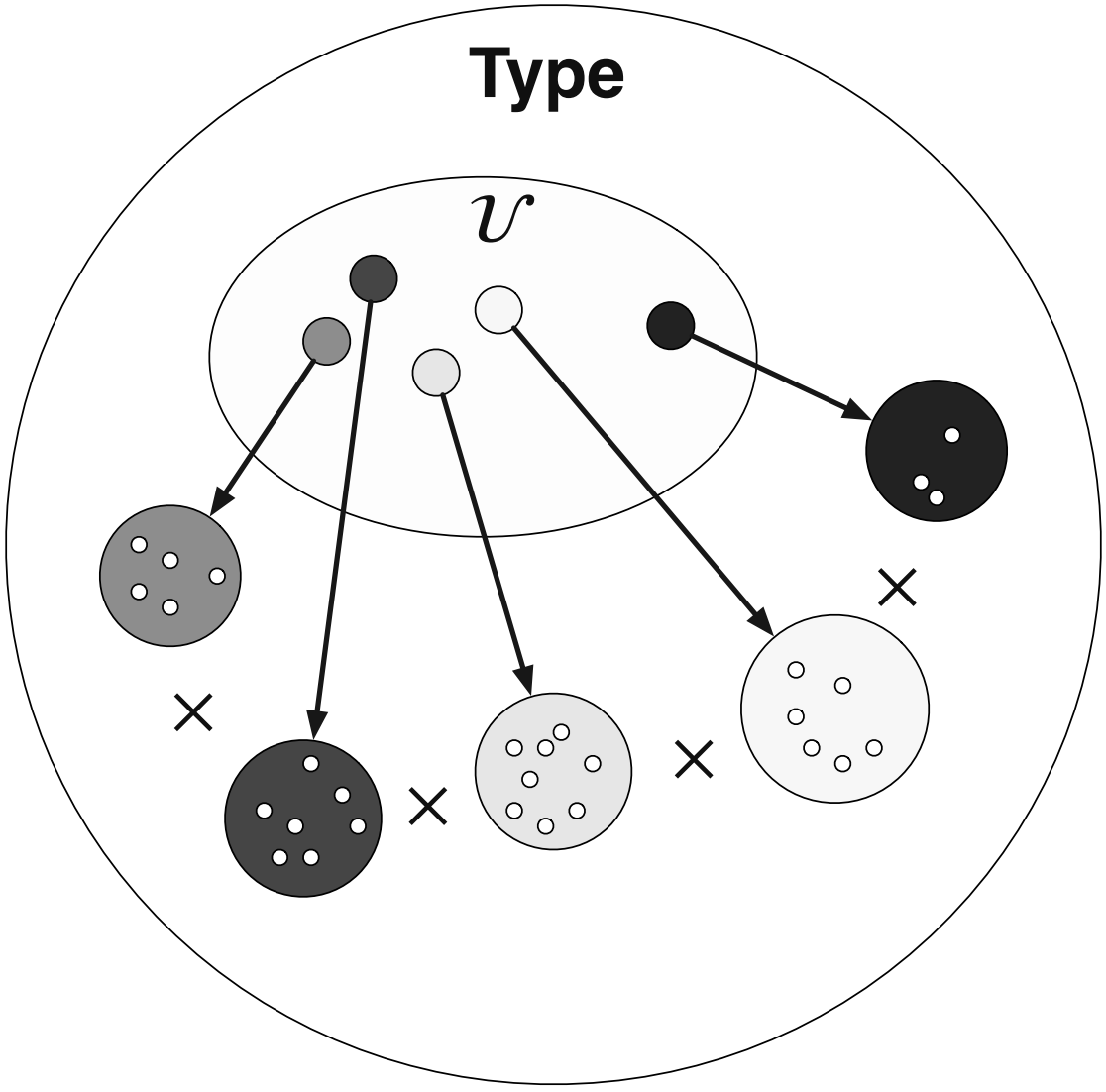
Records as products

Records: the product of the image of $El_{\mathcal{U}}$ in Type restricted to a subset of $\mathcal{U}$.

Records as products



Records as products



Vinyl: beyond records?

Vinyl: beyond records?

Records and corecords are finite products and sums respectively.

Vinyl: beyond records?

Records and corecords are finite products and sums respectively.

$$\mathsf{Rec}(El_{\mathcal{U}}; F; rs) :\equiv \prod_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

Vinyl: beyond records?

Records and corecords are finite products and sums respectively.

$$\mathsf{Rec}(El_{\mathcal{U}}; F; rs) :\equiv \prod_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

$$\mathsf{CoRec}(El_{\mathcal{U}}; F; rs) :\equiv \sum_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

Vinyl: beyond records?

Records and corecords are finite products and sums respectively.

$$\mathsf{Rec}(El_{\mathcal{U}}; F; rs) :\equiv \prod_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

$$\mathsf{CoRec}(El_{\mathcal{U}}; F; rs) :\equiv \sum_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

$$\mathsf{Data}(El_{\mathcal{U}}; F; rs) :\equiv \mathbf{W}_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

Vinyl: beyond records?

Records and corecords are finite products and sums respectively.

$$\mathsf{Rec}(El_{\mathcal{U}}; F; rs) :\equiv \prod_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

$$\mathsf{CoRec}(El_{\mathcal{U}}; F; rs) :\equiv \sum_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

$$\mathsf{Data}(El_{\mathcal{U}}; F; rs) :\equiv \mathbf{W}_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

$$\mathsf{Codata}(El_{\mathcal{U}}; F; rs) :\equiv \mathbf{M}_{\mathcal{U}|_{rs}} F \circ El_{\mathcal{U}}$$

Vinyl: beyond records?

$$(\mathcal{U}|_{rs}) \triangleleft (F \circ El_{\mathcal{U}})$$

Vinyl: beyond records?

$$[\![(\mathcal{U}|_{rs}) \triangleleft (F \circ El_{\mathcal{U}})]\!]_{\Pi}$$

Vinyl: beyond records?

$$[\![(\mathcal{U}|_{rs}) \triangleleft (F \circ El_{\mathcal{U}})]\!]_{\mathsf{W}}$$

Vinyl: beyond records?

$$[\![(\mathcal{U}|_{rs}) \triangleleft (F \circ El_{\mathcal{U}})]\!]_{\mathsf{M}}$$

Presheaves

Presheaves

A presheaf on some space X is a functor $\mathcal{O}(X)^{\mathrm{op}} \to \mathbf{Type}$, where $\mathcal{O}$ is the category of open sets of X for whatever topology you have chosen.

Topologies on some space X

Topologies on some space X

- What are the open sets on X?

Topologies on some space X

- What are the open sets on X?
- The empty set and X are open sets

Topologies on some space X

- What are the open sets on X?
- The empty set and X are open sets
- The union of open sets is open

Topologies on some space X

- What are the open sets on X?
- The empty set and X are open sets
- The union of open sets is open
- Finite intersections of open sets are open

Records are presheaves

Records are presheaves

- Let $\mathcal{O} = \mathcal{P}$, the discrete topology

Records are presheaves

- Let $\mathcal{O} = \mathcal{P}$, the discrete topology
- Then records on a universe X give rise to a presheaf $\mathcal{R}$: subset inclusions are taken to casts from larger to smaller records

Records are presheaves

- Let $\mathcal{O} = \mathcal{P}$, the discrete topology
- Then records on a universe X give rise to a presheaf $\mathcal{R}$: subset inclusions are taken to casts from larger to smaller records

$$\text{for } U \in \mathcal{P}(X) \quad \mathcal{R}(U) :\equiv \prod_{U} El_X|_U : \mathbf{Type}$$

Records are presheaves

- Let $\mathcal{O} = \mathcal{P}$, the discrete topology
- Then records on a universe X give rise to a presheaf $\mathcal{R}$: subset inclusions are taken to casts from larger to smaller records

$$\text{for } U \in \mathcal{P}(X) \quad \mathcal{R}(U) :\equiv \prod_U El_X|_U : \mathbf{Type}$$

$$\text{for } i : V \hookrightarrow U \quad \mathcal{R}(i) :\equiv \mathsf{cast} : \mathcal{R}(U) \to \mathcal{R}(V)$$

Records are sheaves

Records are sheaves

For a cover $U = \bigcup_i U_i$ on X, then:

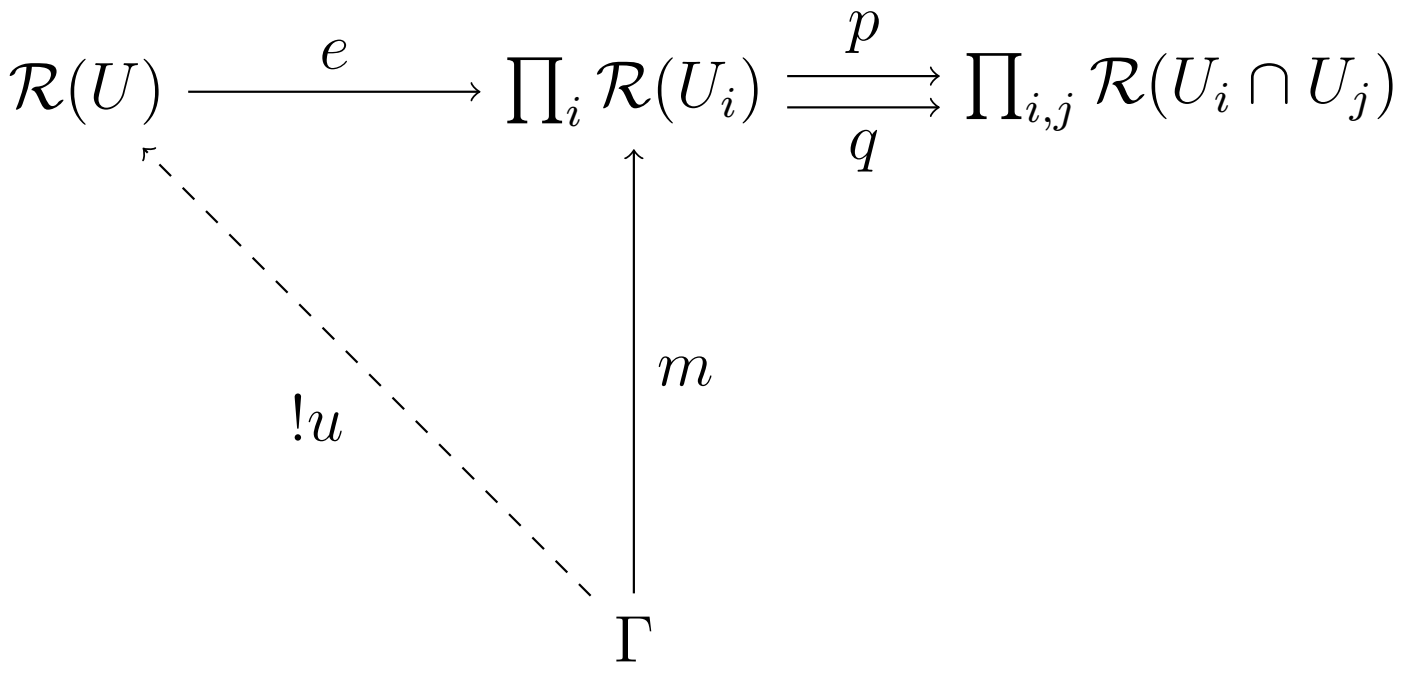
$$\mathcal{R}(U) \xrightarrow{\quad e \quad} \prod_i \mathcal{R}(U_i) \underset{q}{\overset{p}{\rightrightarrows}} \prod_{i,j} \mathcal{R}(U_i \cap U_j)$$

is an equalizer, where

$$\begin{aligned} e &= \lambda r.\lambda i.\ \mathsf{cast}_{U_i}(r) \\ p &= \lambda f.\lambda i.\lambda j.\ \mathsf{cast}_{U_i \cap U_j}(f(i)) \\ q &= \lambda f.\lambda i.\lambda j.\ \mathsf{cast}_{U_i \cap U_j}(f(j)) \end{aligned}$$

Records are sheaves

For a cover $U = \bigcup_i U_i$ on X, then:



where

$$e = \lambda r.\lambda i.\ \mathsf{cast}_{U_i}(r)$$
$$p = \lambda f.\lambda i.\lambda j.\ \mathsf{cast}_{U_i \cap U_j}(f(i))$$
$$q = \lambda f.\lambda i.\lambda j.\ \mathsf{cast}_{U_i \cap U_j}(f(j))$$

Records with non-trivial topology

Records with non-trivial topology

NameType $:\equiv \{F, M, L\}$

Records with non-trivial topology

$$\mathsf{NameType} :\equiv \{F, M, L\}$$

$$\mathcal{A} :\equiv \{\mathsf{Name}[t] \mid t \in \mathsf{NameType}\} \cup \{\mathsf{Phone}[\ell] \mid \ell \in \mathsf{Label}\}$$

Records with non-trivial topology

$$\begin{aligned}
\mathsf{NameType} &:\equiv \{F, M, L\} \\
\mathcal{A} &:\equiv \{\mathsf{Name}[t] \mid t \in \mathsf{NameType}\} \cup \{\mathsf{Phone}[\ell] \mid \ell \in \mathsf{Label}\} \\
El_{\mathcal{A}} &:\equiv \lambda_.\mathbf{String}
\end{aligned}$$

Records with non-trivial topology

$$
\begin{aligned}
\mathsf{NameType} &:\equiv \{F, M, L\} \\
\mathcal{A} &:\equiv \{\mathsf{Name}[t] \mid t \in \mathsf{NameType}\} \cup \{\mathsf{Phone}[\ell] \mid \ell \in \mathsf{Label}\} \\
El_{\mathcal{A}} &:\equiv \lambda\_.\mathbf{String} \\
T &:\equiv \{U \in \mathcal{P}(\mathcal{A}) \mid \mathsf{Name}[M] \in U \\
&\qquad\qquad \Rightarrow \{\mathsf{Name}[F], \mathsf{Name}[L]\} \subseteq U\}
\end{aligned}
$$

Records with non-trivial topology

$$
\begin{aligned}
\mathsf{NameType} &:\equiv \{F, M, L\} \\
\mathcal{A} &:\equiv \{\mathsf{Name}[t] \mid t \in \mathsf{NameType}\} \cup \{\mathsf{Phone}[\ell] \mid \ell \in \mathsf{Label}\} \\
El_{\mathcal{A}} &:\equiv \lambda\_.\mathbf{String} \\
T &:\equiv \{U \in \mathcal{P}(\mathcal{A}) \mid \mathsf{Name}[M] \in U \\
&\qquad\qquad \Rightarrow \{\mathsf{Name}[F], \mathsf{Name}[L]\} \subseteq U\} \\
ex &: \mathcal{R}_T(\{\mathsf{Name}[t] \mid t \in \mathsf{NameType}\}) \\
ex &:\equiv \{\mathsf{Name}[F] \mapsto \text{"Robert"}; \\
&\qquad \mathsf{Name}[M] \mapsto \text{"W"}; \\
&\qquad \mathsf{Name}[L] \mapsto \text{"Harper"}\}
\end{aligned}
$$

Records with non-trivial topology

$$
\begin{aligned}
\mathsf{NameType} &:\equiv \{F, M, L\} \\
\mathcal{A} &:\equiv \{\mathsf{Name}[t] \mid t \in \mathsf{NameType}\} \cup \{\mathsf{Phone}[\ell] \mid \ell \in \mathsf{Label}\} \\
El_{\mathcal{A}} &:\equiv \lambda\_.\mathbf{String} \\
T &:\equiv \{U \in \mathcal{P}(\mathcal{A}) \mid \mathsf{Name}[M] \in U \\
&\qquad\qquad \Rightarrow \{\mathsf{Name}[F], \mathsf{Name}[L]\} \subseteq U\} \\
ex &: \mathcal{R}_T(\{\mathsf{Name}[t] \mid t \in \mathsf{NameType}\}) \\
ex &:\equiv \{\mathsf{Name}[F] \mapsto \text{"Robert"}; \\
&\qquad \mathsf{Name}[M] \mapsto \text{"W"}; \\
&\qquad \mathsf{Name}[L] \mapsto \text{"Harper"}\} \\
nope &: \mathcal{R}_T(\{\mathsf{Name}[M], \mathsf{Phone}[Work]\}) \\
nope &:\equiv \{\mathsf{Name}[M] \mapsto \text{"W"}; \\
&\qquad \mathsf{Phone}[Work] \mapsto \text{"5555555555"}\}
\end{aligned}
$$

Future work

Future work

- Complete: formalization of records with topologies as presheaves in Coq

Future work

- Complete: formalization of records with topologies as presheaves in Coq
- In Progress: formalization of records as sheaves

Future work

- Complete: formalization of records with topologies as presheaves in Coq
- In Progress: formalization of records as sheaves
- Future: extension to *dependent* records using extensional type theory and realizability (Nuprl, MetaPRL)

Questions