Vinyl: Records in Haskell and Type Theory

Jon Sterling
jonmsterling.com

July 30, 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 :: [$*$] $\rightarrow$ $*$ **where**
  RNil :: Rec '[]
  (:&) :: !t $\rightarrow$!(Rec rs) $\rightarrow$ Rec ((s ::: t) ': rs)

Roll your own in Haskell

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

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

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

class s $\in$ (rs :: [$*$])
class ss $\subseteq$ (rs :: [$*$]) **where**
  cast :: Rec rs $\rightarrow$ Rec ss
(=:) : s ::: t $\rightarrow$ t $\rightarrow$ Rec '[s ::: t]
($\oplus$) : Rec ss $\rightarrow$ Rec ts $\rightarrow$ Rec (ss ++ ts)
lens : s ::: t $\in$ rs $\Rightarrow$ s ::: t $\rightarrow$ Lens' (Rec rs) t

Roll your own in Haskell

Roll your own in Haskell

$$\begin{array}{l} f :: \text{Rec } (\text{"foo"} ::: A \text{ ': } rs) \\ \quad \rightarrow \text{Rec } (\text{"bar"} ::: B \text{ ': } \text{"foo"} ::: A \text{ ': } rs) \end{array}$$

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 $\rightarrow *$) $\rightarrow$ (rs :: [k]) $\rightarrow *$ **where**
  RNil :: Rec el '[]
  (:&) :: $\forall$(r :: k) (rs':: [k]). !(el r) $\rightarrow$!(Rec el rs) $\rightarrow$ 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 :: $* \to * \to *$
type family (f :: TyFun k l $\to *$) \$ (x :: k) :: l

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

Example

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

data SLabel :: Label $\rightarrow *$
data SAddrKeys :: AddrKeys $\rightarrow *$ **where**
  SName :: SAddrKeys Name
  SPhone :: SLabel l $\rightarrow$ SAddrKeys (Phone l)
  SEmail :: SLabel l $\rightarrow$ SAddrKeys (Email l)

Example

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

data SLabel :: Label $\rightarrow *$
data SAddrKeys :: AddrKeys $\rightarrow *$

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) $\rightarrow *$ **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) $\rightarrow *$ **where**
type instance Id \$ x = x

type HList rs = Rec Id rs

ex :: HList [$\mathbb{Z}$, **Bool**, **String**]

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 $* \rightarrow *$) $\rightarrow$ ($* \rightarrow *$) $\rightarrow$ [k] $\rightarrow *$ **where**
  RNil :: Rec el f '[]
  (:&) :: !(f (el $ r)) $\rightarrow$!(Rec el f rs) $\rightarrow$ 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 $(\rightarrow)$ Identity (**Either** Error)

Compositional validation

newtype Lift o f g x = Lift $\{$ runLift :: f x ‘o‘ g x $\}$
type Validator = Lift $(\rightarrow)$ 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 $(\rightarrow)$ Identity (**Either** Error)

validateName :: AddrRec Validator ’[Name]
validatePhone :: $\forall$l. AddrRec Validator ’[Phone l]
validateEmail :: $\forall$l. AddrRec Validator ’[Email l]

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

validateContact :: AddrRec Validator TotalContact
validateContact = validateName
$\oplus$ validateEmail $\oplus$ validateEmail
$\oplus$ validatePhone $\oplus$ validatePhone

Record effect operators

$$(\circledast) :: \mathrm{Rec\ el\ (Lift\ } (\rightarrow) \mathrm{\ f\ g)\ rs} \rightarrow \mathrm{Rec\ el\ f\ rs} \rightarrow \mathrm{Rec\ el\ g\ rs}$$

Record effect operators

$(\circledast)$ :: 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 (→) f g) rs → Rec el f rs → Rec el g rs

rtraverse :: Applicative h ⇒
  (∀ x. f x → h (g x)) →
  Rec el f rs → h (Rec el g rs)

rdist :: Applicative f ⇒ Rec el f rs → f (Rec el Identity rs)
rdist = rtraverse (fmap Identity)
```

Compositional validation

bobValid :: AddrRec (**Either** Error) [Name, Email Work]

Compositional validation

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

Compositional validation

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

validBob :: **Either** Error (AddrRec Identity [Name, Email Work])

Compositional validation

```
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 }
```

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]

Concurrent records with Async

fetchName :: AddrRec **IO** '[Name]
fetchName = SName $\Lleftarrow$ runDB nameQuery

Concurrent records with Async

fetchName :: AddrRec **IO** '[Name]
fetchName = SName $\Lleftarrow$ runDB nameQuery

fetchWorkEmail :: AddrRec **IO** '[Email Work]

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]
```

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]

Concurrent records with Async

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

$(\circledS) :: (\forall a.\ f\ a \rightarrow g\ a) \rightarrow \text{Rec el f rs} \rightarrow \text{Rec el g rs}$

bobConcurrently :: Rec el Concurrently [Name, Email Work]
bobConcurrently = Concurrently $\circledS$ 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])

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

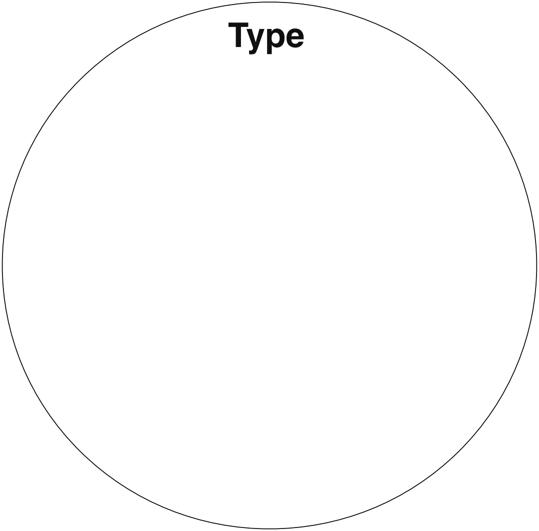
- A type $\mathcal{U}$ of **codes** for types.
- Function $El_{\mathcal{U}} : \mathcal{U} \to$ Type.

Universes à la Tarski

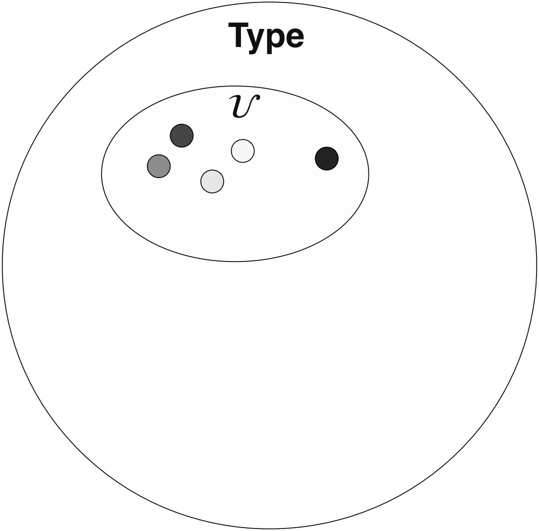
- A type $\mathcal{U}$ of **codes** for types.
- Function $El_{\mathcal{U}} : \mathcal{U} \to \text{Type}$.

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

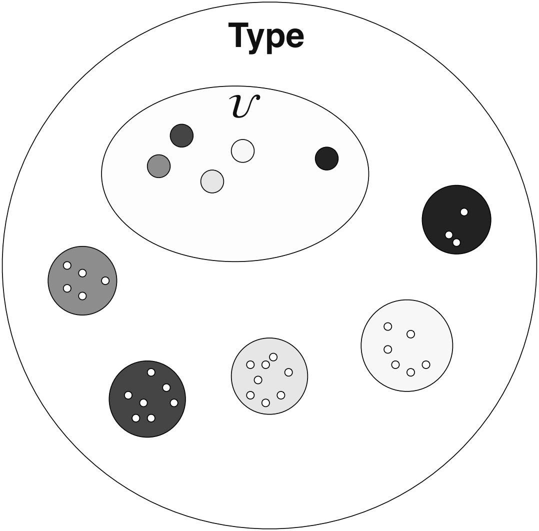
Universes à la Tarski



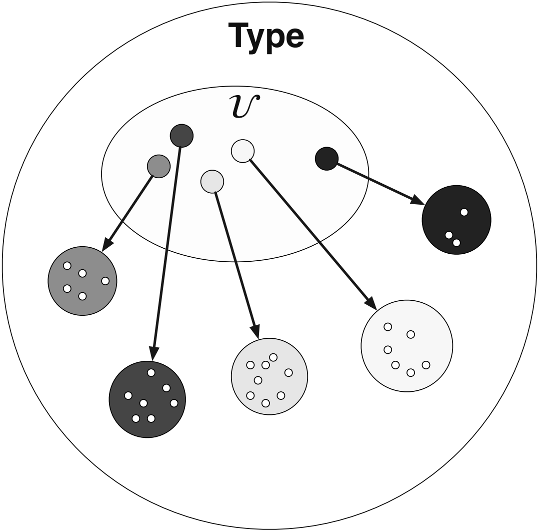
Universes à la Tarski



Universes à la Tarski



Universes à la Tarski

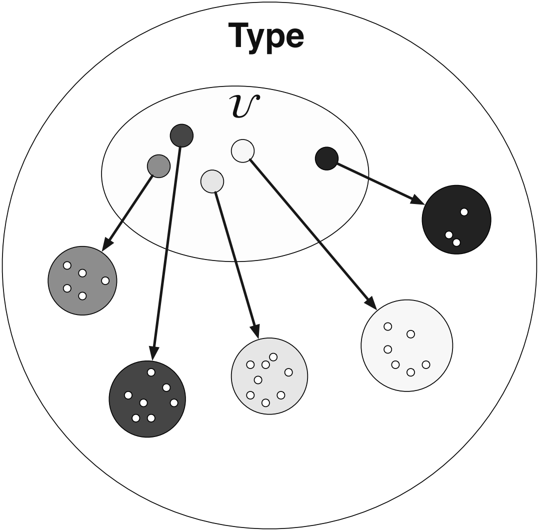


Records as products

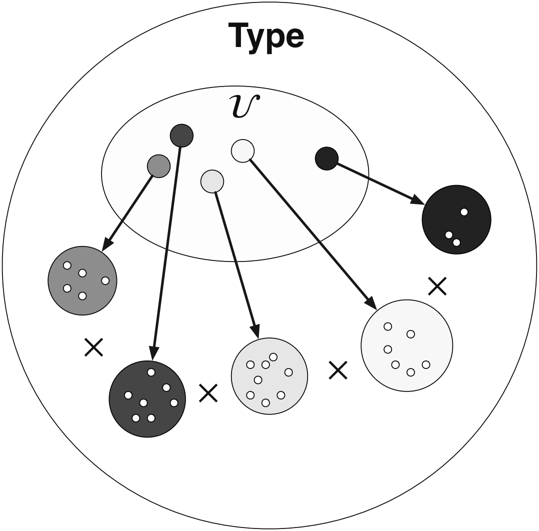
Records as products

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



Containers: the syntax for data types

$$\frac{}{\mathsf{container} : \mathrm{Type}}$$

Containers: the syntax for data types

$$\frac{}{\textsf{container} : \textsf{Type}} \qquad \frac{\mathcal{U} : \textsf{Type} \quad El_{\mathcal{U}} : \mathcal{U} \to \textsf{Type}}{\mathcal{U} \triangleleft El_{\mathcal{U}} : \textsf{container}}$$

Containers: the syntax for data types

$$\frac{}{\textsf{container} : \textsf{Type}} \qquad \frac{\mathcal{U} : \textsf{Type} \quad El_{\mathcal{U}} : \mathcal{U} \to \textsf{Type}}{\mathcal{U} \triangleleft El_{\mathcal{U}} : \textsf{container}}$$

$$\frac{C : \textsf{container}}{C.\textsf{Sh} : \textsf{Type}}$$

Containers: the syntax for data types

$$\frac{}{\mathsf{container} : \mathsf{Type}} \qquad \frac{\mathcal{U} : \mathsf{Type} \quad El_{\mathcal{U}} : \mathcal{U} \to \mathsf{Type}}{\mathcal{U} \lhd El_{\mathcal{U}} : \mathsf{container}}$$

$$\frac{C : \mathsf{container}}{C.\mathsf{Sh} : \mathsf{Type}} \qquad \frac{C \rightsquigarrow \mathcal{U} \lhd El_{\mathcal{U}}}{C.\mathsf{Sh} \rightsquigarrow \mathcal{U}}$$

Containers: the syntax for data types

$$\frac{}{\textsf{container} : \textsf{Type}} \qquad \frac{\mathcal{U} : \textsf{Type} \quad El_{\mathcal{U}} : \mathcal{U} \to \textsf{Type}}{\mathcal{U} \triangleleft El_{\mathcal{U}} : \textsf{container}}$$

$$\frac{C : \textsf{container}}{C.\textsf{Sh} : \textsf{Type}} \qquad \frac{C \rightsquigarrow \mathcal{U} \triangleleft El_{\mathcal{U}}}{C.\textsf{Sh} \rightsquigarrow \mathcal{U}}$$

$$\frac{C : \textsf{container}}{C.\textsf{Po} : C.\textsf{Sh} \to \textsf{Type}}$$

Containers: the syntax for data types

$$\frac{}{\textsf{container} : \textsf{Type}} \qquad \frac{\mathcal{U} : \textsf{Type} \quad El_{\mathcal{U}} : \mathcal{U} \to \textsf{Type}}{\mathcal{U} \lhd El_{\mathcal{U}} : \textsf{container}}$$

$$\frac{C : \textsf{container}}{C.\textsf{Sh} : \textsf{Type}} \qquad \frac{C \rightsquigarrow \mathcal{U} \lhd El_{\mathcal{U}}}{C.\textsf{Sh} \rightsquigarrow \mathcal{U}}$$

$$\frac{C : \textsf{container}}{C.\textsf{Po} : C.\textsf{Sh} \to \textsf{Type}} \qquad \frac{C \rightsquigarrow \mathcal{U} \lhd El_{\mathcal{U}}}{C.\textsf{Po} \rightsquigarrow El_{\mathcal{U}}}$$

Restricting containers

$$\frac{C : \mathsf{container} \quad \mathcal{V} \subseteq C.\mathsf{Sh}}{C|_{\mathcal{V}} : \mathsf{container}}$$

Restricting containers

$$\frac{C : \mathsf{container} \quad \mathcal{V} \subseteq C.\mathsf{Sh}}{C|_{\mathcal{V}} : \mathsf{container}}$$

$$\frac{C \rightsquigarrow \mathcal{U} \lhd El_{\mathcal{U}}}{C|_{\mathcal{V}} \rightsquigarrow \mathcal{V} \lhd El_{\mathcal{U}}|_{\mathcal{V}}}$$

Container lifting

$$\frac{C : \mathsf{container} \quad F : \mathsf{Type} \to \mathsf{Type}}{F \bullet C : \mathsf{container}}$$

Container lifting

$$\frac{C : \mathsf{container} \quad F : \mathsf{Type} \to \mathsf{Type}}{F \bullet C : \mathsf{container}}$$

$$\frac{C \rightsquigarrow \mathcal{U} \lhd El_{\mathcal{U}}}{F \bullet C \rightsquigarrow \mathcal{U} \lhd F \circ El_{\mathcal{U}}}$$

A menagerie of quantifiers

A menagerie of quantifiers

Dependent Products:

$$\frac{\Gamma \vdash A : \textsf{Type} \quad \Gamma, x : A \vdash B : \textsf{Type}}{\Gamma \vdash \prod_A B : \textsf{Type}}$$

$$\frac{\Gamma, x : A \vdash e : B[x]}{\Gamma \vdash \lambda x.e : \prod_A B}$$

A menagerie of quantifiers

Dependent Sums:

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \sum_A B : \mathsf{Type}}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a]}{\Gamma \vdash \langle a, b \rangle : \sum_A B}$$

A menagerie of quantifiers

Inductive Types:

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \mathrm{W}_A\, B : \mathsf{Type}}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma, v : B[a] \vdash b : \mathrm{W}_A\, B}{\Gamma \vdash \mathsf{sup}(a; v.\, b) : \mathrm{W}_A\, B}$$

A menagerie of quantifiers

Coinductive Types:

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \mathrm{M}_A\, B : \mathsf{Type}}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma, v : B[a] \vdash b : \infty\,(\mathrm{M}_A\, B)}{\Gamma \vdash \mathsf{inf}(a; v.\, b) : \mathrm{M}_A\, B}$$

A scheme for quantifiers

A scheme for quantifiers

$$\frac{\Gamma, A : \mathsf{Type}, (x : A \vdash B : \mathsf{Type}) \vdash Q_A B : \mathsf{Type}}{\Gamma \vdash Q \ \mathsf{quantifier}}$$

Quantifiers give containers semantics

$$\frac{\Gamma, A : \mathsf{Type}, (x : A \vdash B : \mathsf{Type}) \vdash Q_A B : \mathsf{Type}}{\Gamma \vdash Q \text{ quantifier}}$$

Quantifiers give containers semantics

$$\frac{\Gamma, A : \mathsf{Type}, (x : A \vdash B : \mathsf{Type}) \vdash Q_A B : \mathsf{Type}}{\Gamma \vdash Q \text{ quantifier}}$$

$$\frac{C : \text{container} \quad Q \text{ quantifier}}{[\![C]\!]_Q : \mathsf{Type}}$$

Quantifiers give containers semantics

$$\frac{\Gamma, A : \textsf{Type}, (x : A \vdash B : \textsf{Type}) \vdash Q_A B : \textsf{Type}}{\Gamma \vdash Q \text{ quantifier}}$$

$$\frac{C : \text{container} \quad Q \text{ quantifier}}{[\![C]\!]_Q : \textsf{Type}}$$

$$\frac{C \rightsquigarrow \mathcal{U} \lhd El_{\mathcal{U}}}{[\![C]\!]_Q \rightsquigarrow Q_{\mathcal{U}} El_{\mathcal{U}}}$$

Vinyl records as containers

Vinyl records as containers

Records and corecords are finite products and sums respectively.

Vinyl records as containers

Records and corecords are finite products and sums respectively.

$$\mathsf{Rec}\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \lhd El_{\mathcal{U}})|_{rs\ni -}]\!]_{\Pi}$$

Vinyl records as containers

Records and corecords are finite products and sums respectively.

$$\mathsf{Rec}\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \lhd El_{\mathcal{U}})|_{rs \ni -}]\!]_{\Pi}$$
$$\mathsf{CoRec}\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \lhd El_{\mathcal{U}})|_{rs \ni -}]\!]_{\Sigma}$$

Vinyl records as containers

Records and corecords are finite products and sums respectively.

$$\textsf{Rec}\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \lhd El_{\mathcal{U}})|_{rs \ni -}]\!]_{\Pi}$$
$$\textsf{CoRec}\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \lhd El_{\mathcal{U}})|_{rs \ni -}]\!]_{\Sigma}$$

$$???\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \lhd El_{\mathcal{U}})|_{rs \ni -}]\!]_{\mathrm{W}}$$

Vinyl records as containers

Records and corecords are finite products and sums respectively.

$$\mathsf{Rec}\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \triangleleft El_{\mathcal{U}})|_{rs \ni -}]\!]_{\Pi}$$
$$\mathsf{CoRec}\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \triangleleft El_{\mathcal{U}})|_{rs \ni -}]\!]_{\Sigma}$$

$$???\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \triangleleft El_{\mathcal{U}})|_{rs \ni -}]\!]_{\mathrm{W}}$$
$$???\ El_{\mathcal{U}}\ F\ rs \cong [\![F \bullet (\mathcal{U} \triangleleft El_{\mathcal{U}})|_{rs \ni -}]\!]_{\mathrm{M}}$$

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

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

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

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

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

Records are sheaves

Records are sheaves

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

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

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:

$$
\begin{tikzcd}
\mathcal{R}(U) \arrow[rr, "e"] & & \prod_i \mathcal{R}(U_i) \arrow[r, shift left, "p"] \arrow[r, shift right, "q"'] & \prod_{i,j} \mathcal{R}(U_i \cap U_j) \\
& & & \\
& & \Gamma \arrow[uu, "m"'] \arrow[uull, dashed, "!u"] &
\end{tikzcd}
$$

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 with non-trivial topology

Records with non-trivial topology

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

Records with non-trivial topology

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

Records with non-trivial topology

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

Records with non-trivial topology

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

Records with non-trivial topology

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

Records with non-trivial topology

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

Records with non-trivial topology

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

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