Introduction to Vinyl

Jon Sterling
jonmsterling.com

July 9, 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)
```

Universes à la Tarski

Universes à la Tarski

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

Universes à la Tarski

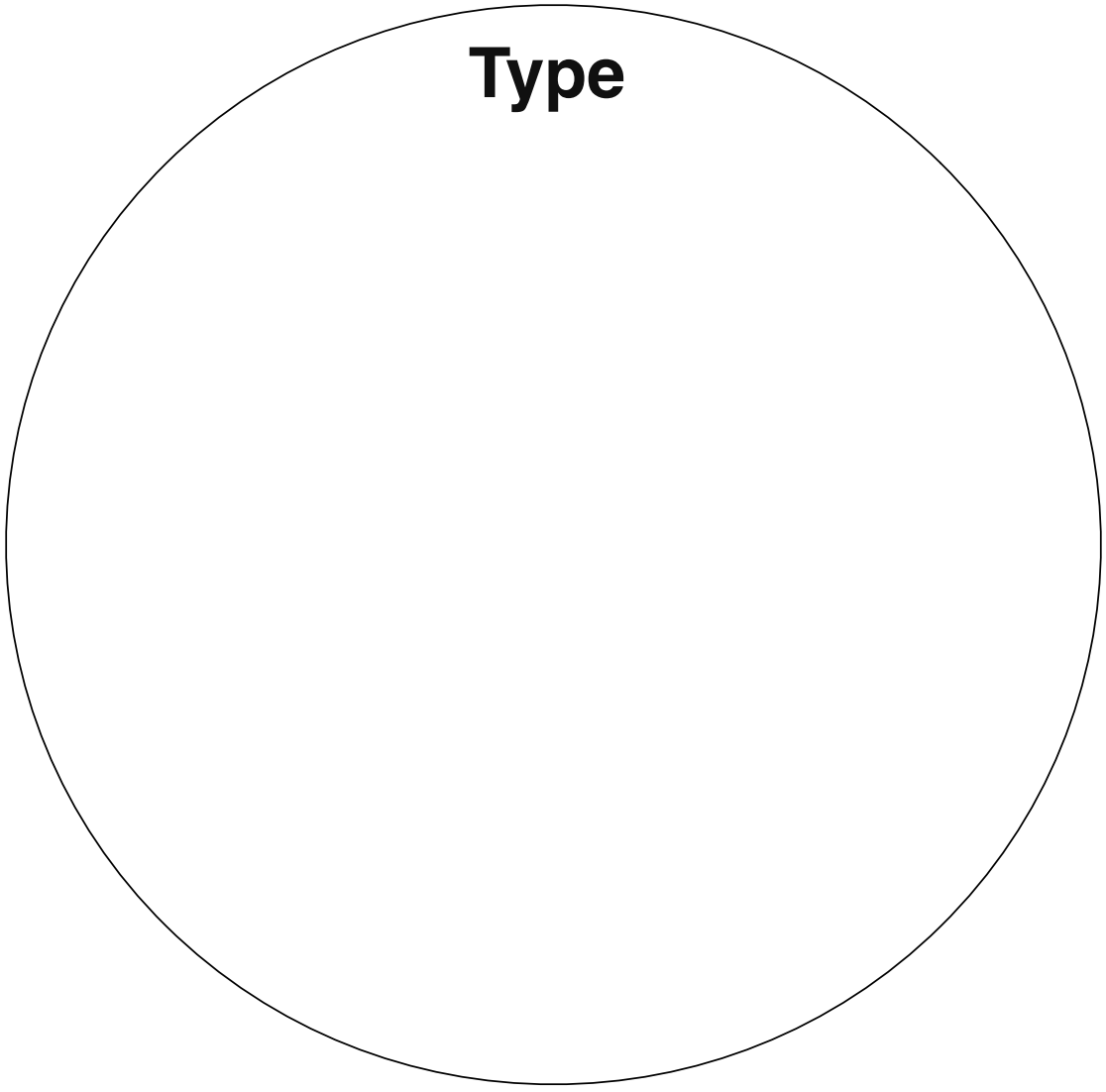
- A type $\mathcal{U}$ of **codes** for types.
- Function $El_{\mathcal{U}} : \mathcal{U} \to$ Type.

Universes à la Tarski

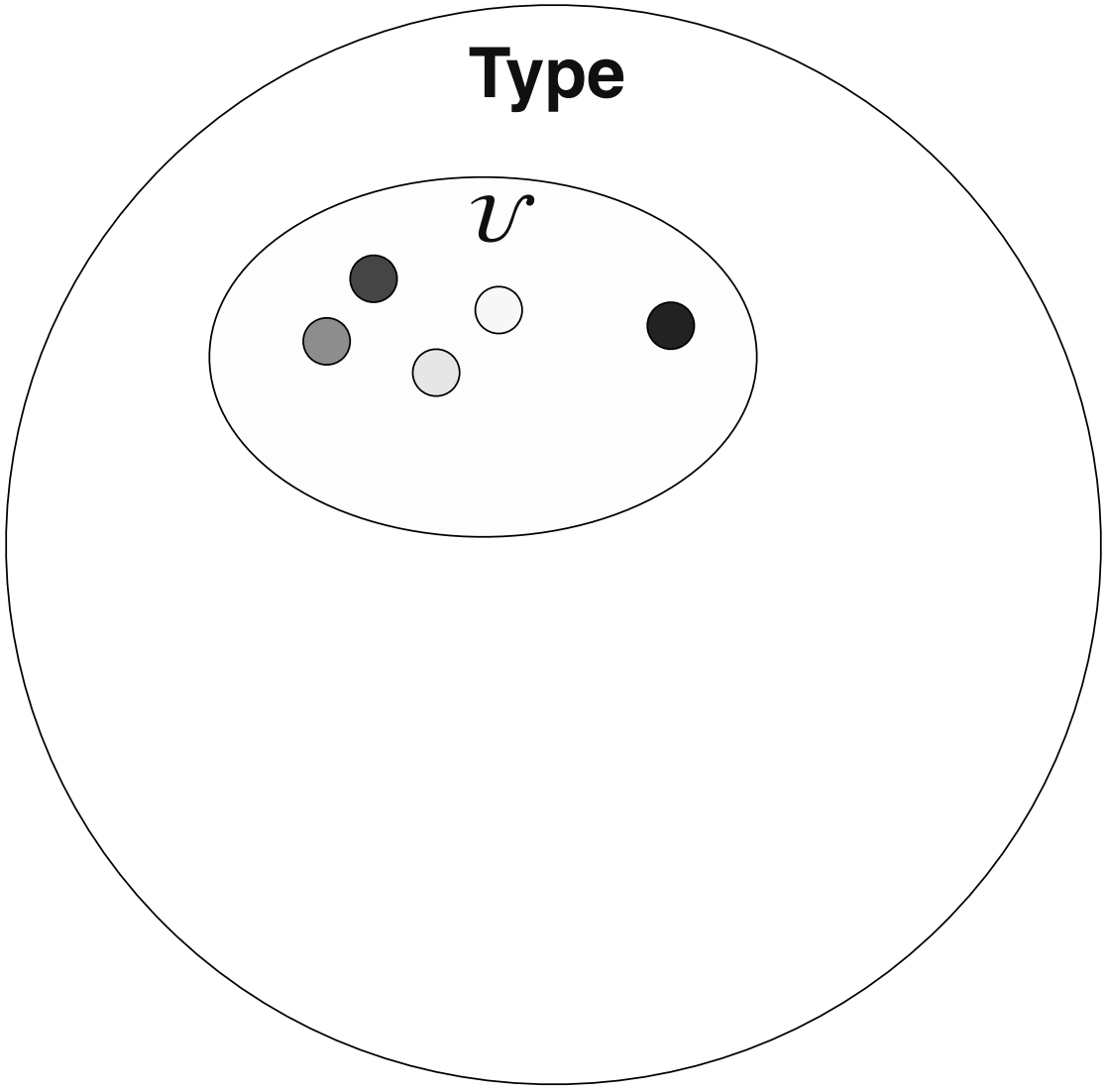
- A type $\mathcal{U}$ of **codes** for types.
- Function $El_{\mathcal{U}} : \mathcal{U} \to$ Type.

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

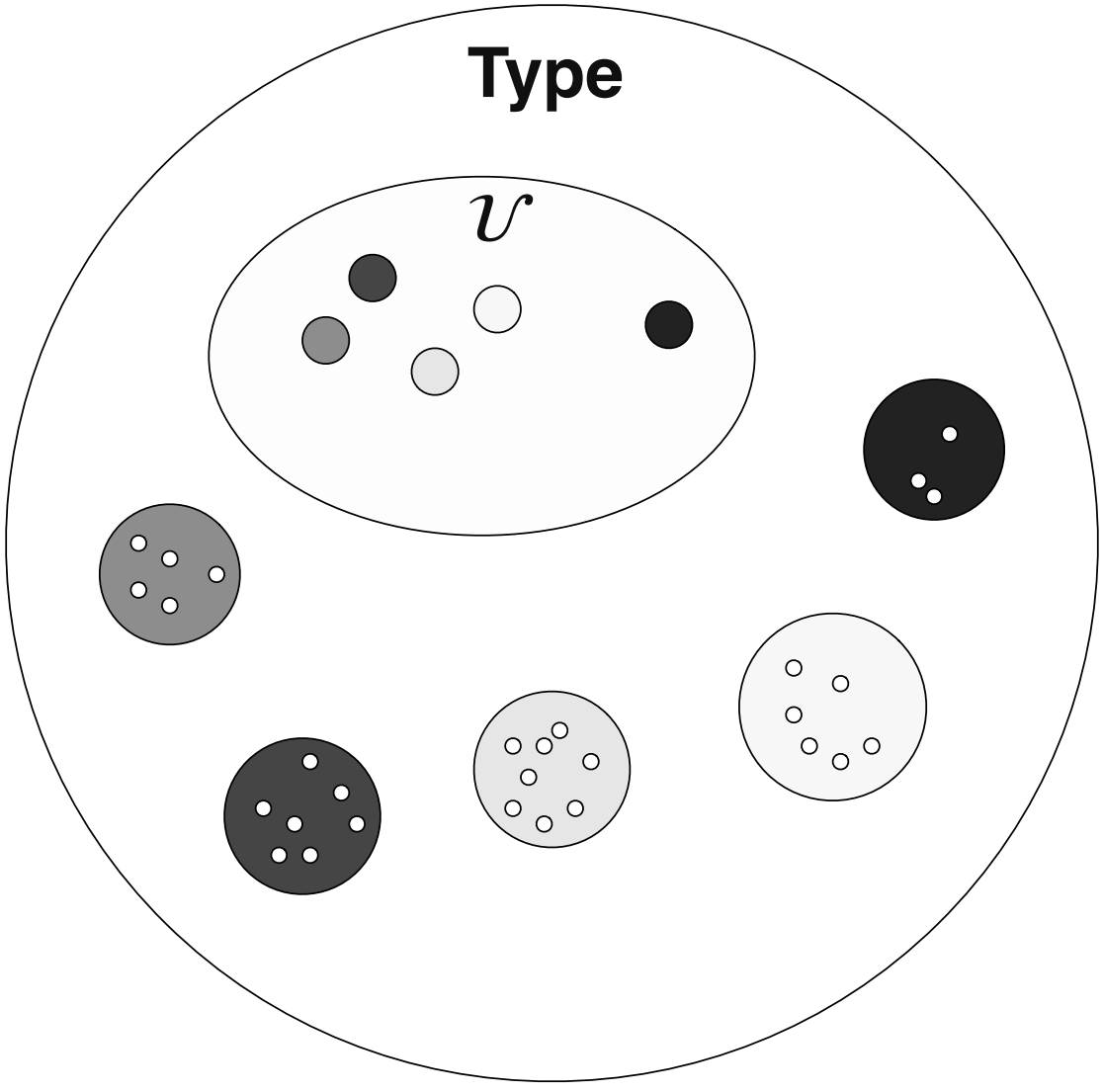
Universes à la Tarski



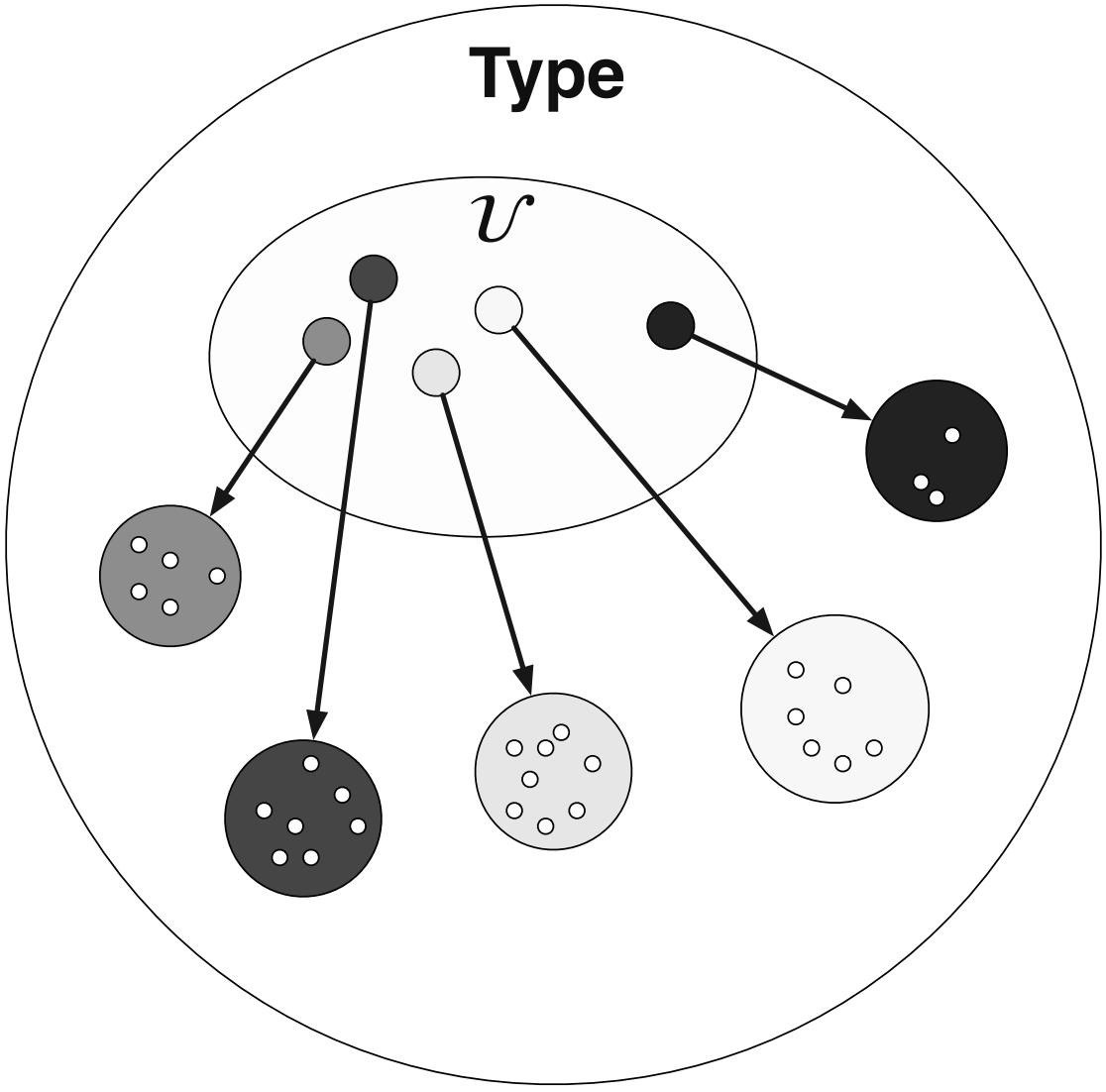
Universes à la Tarski



Universes à la Tarski



Universes à la Tarski



Records in Haskell

Records in Haskell

data Rec :: $(\mathcal{U} \to *) \to [\,\mathcal{U}\,] \to *$ **where**

Records in Haskell

data Rec :: $(\mathcal{U} \to *) \to [\,\mathcal{U}\,] \to *$ **where**
  RNil :: Rec $\mathsf{el}_{\mathcal{U}}$ '[]

Records in Haskell

data Rec :: $(\mathcal{U} \to *) \to [\,\mathcal{U}\,] \to *$ **where**
  RNil :: Rec el$_{\mathcal{U}}$ '[]
  (:&) :: !(el$_{\mathcal{U}}$ r) $\to$!(Rec el$_{\mathcal{U}}$ rs) $\to$ Rec el$_{\mathcal{U}}$ (r ': rs)

Records in Haskell (Actually)

Records in Haskell (Actually)

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

Records in Haskell (Actually)

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

Records in Haskell (Actually)

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

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

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]
```

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
```

Effects inside records

data Rec :: (TyFun $\mathcal{U}$ $*$ $\to$ $*$) $\to$ ($*$ $\to$ $*$) $\to$ [$\mathcal{U}$] $\to$ $*$ **where**
  RNil :: Rec el$_{\mathcal{U}}$ f '[]
  (:&) :: !(f (el$_{\mathcal{U}}$ $ r)) $\to$!(Rec el$_{\mathcal{U}}$ f rs) $\to$ Rec el$_{\mathcal{U}}$ f (r ': rs)

Effects inside records

data Rec :: (TyFun $\mathcal{U}$ $*$ $\to$ $*$) $\to$ ($*$ $\to$ $*$) $\to$ [$\mathcal{U}$] $\to$ $*$ **where**
  RNil :: Rec el$_{\mathcal{U}}$ f '[]
  (:&) :: !(f (el$_{\mathcal{U}}$ $ r)) $\to$!(Rec el$_{\mathcal{U}}$ f rs) $\to$ Rec el$_{\mathcal{U}}$ f (r ': rs)

(=:) : Applicative f $\Rightarrow$ sing r $\to$ el$_{\mathcal{U}}$ $ r $\to$ Rec el$_{\mathcal{U}}$ f '[r]

Effects inside records

data Rec :: (TyFun $\mathcal{U}$ $*$ $\to$ $*$) $\to$ ($*$ $\to$ $*$) $\to$ [$\mathcal{U}$] $\to$ $*$ **where**
  RNil :: Rec el$_{\mathcal{U}}$ f '[]
  (:&) :: !(f (el$_{\mathcal{U}}$ $ r)) $\to$!(Rec el$_{\mathcal{U}}$ f rs) $\to$ Rec el$_{\mathcal{U}}$ f (r ': rs)

(=:) : Applicative f $\Rightarrow$ sing r $\to$ el$_{\mathcal{U}}$ $ r $\to$ Rec el$_{\mathcal{U}}$ f '[r]
k =: x = pure x :& RNil

Effects inside records

data Rec :: (TyFun $\mathcal{U}$ $*$ $\to$ $*$) $\to$ ($*$ $\to$ $*$) $\to$ [$\mathcal{U}$] $\to$ $*$ **where**
  RNil :: Rec el$_{\mathcal{U}}$ f '[]
  (:&) :: !(f (el$_{\mathcal{U}}$ $ r)) $\to$!(Rec el$_{\mathcal{U}}$ f rs) $\to$ Rec el$_{\mathcal{U}}$ f (r ': rs)

(=:) : Applicative f $\Rightarrow$ sing r $\to$ el$_{\mathcal{U}}$ $ r $\to$ Rec el$_{\mathcal{U}}$ f '[r]
k =: x = pure x :& RNil

($\Leftarrow\!\!\shortmid$): sing r $\to$ f (el$_{\mathcal{U}}$ $ r) $\to$ Rec el$_{\mathcal{U}}$ f '[r]

Effects inside records

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

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

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

Compositional Kit

Compositional Kit

```
newtype Lift o f g x = Lift { runLift :: f x `o` g x }
```

Compositional Kit

newtype Lift o f g x = Lift { runLift :: f x `o` g x }

type Validator = Lift $(\rightarrow)$ Identity (**Either** Error)

Compositional Kit

```
newtype Lift o f g x = Lift { runLift :: f x `o` g x }

type Validator = Lift (→) Identity (Either Error)
type JSONRenderer = Lift (→) Identity (Const Aeson.Pair)
```

Compositional Kit

newtype Lift o f g x = Lift { runLift :: f x `o` g x }

type Validator = Lift $(\rightarrow)$ Identity (**Either** Error)
type JSONRenderer = Lift $(\rightarrow)$ Identity (Const Aeson.Pair)

(⊛) :: $\text{Rec}_{\mathcal{U}}$ (Lift $(\rightarrow)$ f g) rs $\rightarrow$ $\text{Rec}_{\mathcal{U}}$ f rs $\rightarrow$ $\text{Rec}_{\mathcal{U}}$ g rs

Compositional Kit

newtype Lift o f g x = Lift { runLift :: f x `o` g x }

type Validator = Lift $(\rightarrow)$ Identity (**Either** Error)
type JSONRenderer = Lift $(\rightarrow)$ Identity (Const Aeson.Pair)

$(\circledcirc)$ :: $\text{Rec}_{\mathcal{U}}$ (Lift $(\rightarrow)$ f g) rs $\rightarrow$ $\text{Rec}_{\mathcal{U}}$ f rs $\rightarrow$ $\text{Rec}_{\mathcal{U}}$ g rs

rdist :: Applicative f $\Rightarrow$ $\text{Rec}_{\mathcal{U}}$ f rs $\rightarrow$ f ($\text{Rec}_{\mathcal{U}}$ Identity rs)

Compositional Kit

newtype Lift o f g x = Lift { runLift :: f x `o` g x }

type Validator = Lift $(\rightarrow)$ Identity (**Either** Error)
type JSONRenderer = Lift $(\rightarrow)$ Identity (Const Aeson.Pair)

$(\circledcirc)$ :: $\text{Rec}_{\mathcal{U}}$ (Lift $(\rightarrow)$ f g) rs $\rightarrow$ $\text{Rec}_{\mathcal{U}}$ f rs $\rightarrow$ $\text{Rec}_{\mathcal{U}}$ g rs

rdist :: Applicative f $\Rightarrow$ $\text{Rec}_{\mathcal{U}}$ f rs $\rightarrow$ f ($\text{Rec}_{\mathcal{U}}$ Identity rs)

rtraverse ::
  Applicative h
    $\Rightarrow$ (f $\rightsquigarrow$ h $\circ$ g)
    $\rightarrow$ $\text{Rec}_{\mathcal{U}}$ f rs
    $\rightarrow$ h ($\text{Rec}_{\mathcal{U}}$ g rs)

Demonstration