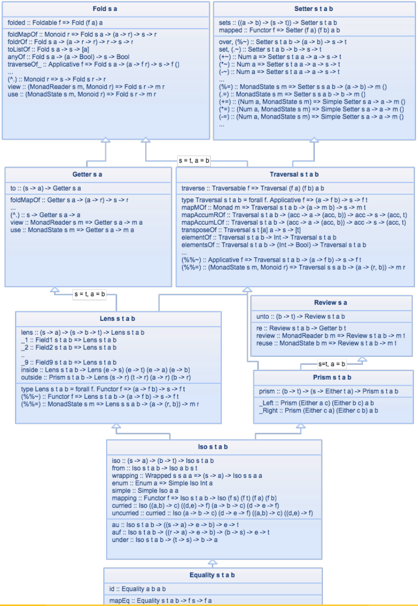
Optics and Type Equivalences

Jacques Carette, Amr Sabry

Optics


Fold s a
folded :: Foldable f => Fold (f a) a
foldMapOf :: Monoid r => Fold s a -> (a -> r) -> s -> r
foldrOf :: Fold s a -> (a -> r -> r) -> r -> s -> r
toListOf :: Fold s a -> s -> [a]
anyOf :: Fold s a -> (a -> Bool) -> s -> Bool
traverseOf_ :: Applicative f => Fold s a -> (a -> f r) -> s -> f ()
...
(^.) :: Monoid r => s -> Fold s r -> r
view :: (MonadReader s m, Monoid r) => Fold s r -> m r
use :: (MonadState s m, Monoid r) => Fold s r -> m r
Setter s t a b
sets :: ((a -> b) -> (s -> t)) -> Setter s t a b
mapped :: Functor f => Setter (f a) (f b) a b
over, (%~) :: Setter s t a b -> (a -> b) -> s -> t
set, (.~) :: Setter s t a b -> b -> s -> t
(+~) :: Num a => Setter s t a a -> a -> s -> t
(*~) :: Num a => Setter s t a a -> a -> s -> t
(-~) :: Num a => Setter s t a a -> a -> s -> t
...
(%=) :: MonadState s m => Setter s s a b -> (a -> b) -> m ()
(.=) :: MonadState s m => Setter s s a b -> b -> m ()
(+=) :: (Num a, MonadState s m) => Simple Setter s a -> a -> m ()
(*=) :: (Num a, MonadState s m) => Simple Setter s a -> a -> m ()
(-=) :: (Num a, MonadState s m) => Simple Setter s a -> a -> m ()
...
s = t, a = b
Getter s a
to :: (s -> a) -> Getter s a
foldMapOf :: Getter s a -> (a -> r) -> s -> r
...
(^.) :: s -> Getter s a -> a
view :: MonadReader s m => Getter s a -> m a
use :: MonadState s m => Getter s a -> m a
Traversal s t a b
traverse :: Traversable f => Traversal (f a) (f b) a b
type Traversal s t a b = forall f. Applicative f => (a -> f b) -> s -> f t
mapMOf :: Monad m => Traversal s t a b -> (a -> m b) -> s -> m t
mapAccumROf :: Traversal s t a b -> (acc -> a -> (acc, b)) -> acc -> s -> (acc, t)
mapAccumLOf :: Traversal s t a b -> (acc -> a -> (acc, b)) -> acc -> s -> (acc, t)
transposeOf :: Traversal s t [a] a -> s -> [t]
elementOf :: Traversal s t a b -> Int -> Traversal s t a b
elementsOf :: Traversal s t a b -> (Int -> Bool) -> Traversal s t a b
...
(%%~) :: Applicative f => Traversal s t a b -> (a -> f b) -> s -> f t
(%%=) :: (MonadState s m, Monoid r) => Traversal s s a b -> (a -> (r, b)) -> m r
s = t, a = b
Review s a
unto :: (b -> t) -> Review s t a b
re :: Review s t a b -> Getter b t
review :: MonadReader b m => Review s t a b -> m t
reuse :: MonadState b m => Review s t a b -> m t
Lens s t a b
lens :: (s -> a) -> (s -> b -> t) -> Lens s t a b
_1 :: Field1 s t a b => Lens s t a b
_2 :: Field2 s t a b => Lens s t a b
..
_9 :: Field9 s t a b => Lens s t a b
inside :: Lens s t a b -> Lens (e -> s) (e -> t) (e -> a) (e -> b)
outside :: Prism s t a b -> Lens (s -> r) (t -> r) (a -> r) (b -> r)
type Lens s t a b = forall f. Functor f => (a -> f b) -> s -> f t
(%%~) :: Functor f => Lens s t a b -> (a -> f b) -> s -> f t
(%%=) :: MonadState s m => Lens s s a b -> (a -> (r, b)) -> m r
s=t, a = b
Prism s t a b
prism :: (b -> t) -> (s -> Either t a) -> Prism s t a b
_Left :: Prism (Either a c) (Either b c) a b
_Right :: Prism (Either c a) (Either c b) a b
Iso s t a b
iso :: (s -> a) -> (b -> t) -> Iso s t a b
from :: Iso s t a b -> Iso a b s t
wrapping :: Wrapped s s a a => (s -> a) -> Iso s s a a
enum :: Enum a => Simple Iso Int a
simple :: Simple Iso a a
mapping :: Functor f => Iso s t a b -> Iso (f s) (f t) (f a) (f b)
curried :: Iso ((a,b) -> c) ((d,e) -> f) (a -> b -> c) (d -> e -> f)
uncurried :: curried :: Iso (a -> b -> c) (d -> e -> f) ((a,b) -> c) ((d,e) -> f)
au :: Iso s t a b -> ((s -> a) -> e -> b) -> e -> t
auf :: Iso s t a b -> ((r -> a) -> e -> b) -> (b -> s) -> e -> t
under :: Iso s t a b -> (t -> s) -> b -> a
Equality s t a b
id :: Equality a b a b
mapEq :: Equality s t a b -> f s -> f a

Based on a 'reversible' core:

```
Iso s t a b

iso :: (s -> a) -> (b -> t) -> Iso s t a b
from :: Iso s t a b -> Iso a b s t
wrapping :: Wrapped s s a a => (s -> a) -> Iso s s a a
enum :: Enum a => Simple Iso Int a
simple :: Simple Iso a a
mapping :: Functor f => Iso s t a b -> Iso (f s) (f t) (f a) (f b)
curried :: Iso ((a,b) -> c) ((d,e) -> f) (a -> b -> c) (d -> e -> f)
uncurried :: curried :: Iso (a -> b -> c) (d -> e -> f) ((a,b) -> c) ((d,e) -> f)

au :: Iso s t a b -> ((s -> a) -> e -> b) -> e -> t
auf :: Iso s t a b -> ((r -> a) -> e -> b) -> (b -> s) -> e -> t
under :: Iso s t a b -> (t -> s) -> b -> a
```

Lens in Haskell

```haskell
data Lens s a = Lens { view :: s -> a, set :: s -> a -> s }
```

Lens in Haskell

```haskell
data Lens s a = Lens { view :: s -> a, set :: s -> a -> s }
```

Example?

```haskell
fst :: Lens (a , b) a
fst = Lens { view = \(a,b) -> a , set = \(a,b) a' -> (a',b) }
```

Lens in Haskell

```
data Lens s a = Lens { view :: s -> a, set :: s -> a -> s }
```

Example?

```
fst :: Lens (a , b) a
fst = Lens { view = \(a,b) -> a , set = \(a,b) a' -> (a',b) }
```

Laws? Optimizations?

```
view (set s a) == a
set s (view s) == s
set (set s a) a' == set s a'
```

Lens in Agda

```
record GS-Lens {ℓs ℓa : Level} (S : Set ℓs) (A : Set ℓa) : Set (ℓs ⊔ ℓa) where
  field
    get    : S → A
    set    : S → A → S
    getput : {s : S} {a : A} → get (set s a) ≡ a
    putget : (s : S)         → set s (get s) ≡ s
    putput : (s : S) (a a' : A) → set (set s a) a' ≡ set s a'
open GS-Lens
```

Works... but the proofs can be tedious.

```
fst : {A B : Set} → GS-Lens (A × B) A
fst = record { get = λ {(a , b) → a}
             ; set = λ {(a , b) a' → (a' , b)}
             ; getput = λ {s} {a} → refl
             ; putget = λ { (a , b) → refl }
             ; putput = λ { (a₀ , b) a₁ a₂ → refl } }
```

Lens in Agda 2

Or, the return of constant-complement lenses:

```
record Lens₁ {ℓ : Level} (S : Set ℓ) (A : Set ℓ) : Set (suc ℓ) where
  constructor ∃-lens
  field
    {C} : Set ℓ
    iso : S ≃ (C × A)
fst : {A B : Set} → Lens₁ (A × B) A
fst = ∃-lens swap⋆equiv
```

Lens in Agda 2

where

```
record isqinv {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} (f : A → B) :
  Set (ℓ ⊔ ℓ') where
  constructor qinv
  field
    g : B → A
    α : (f ∘ g) ∼ id
    β : (g ∘ f) ∼ id

_≃_ : ∀ {ℓ ℓ'} → Set ℓ → Set ℓ' → Set (ℓ ⊔ ℓ')
A ≃ B = Σ (A → B) isqinv
```

Lens in Agda 2

Or, the return of constant-complement lenses:

```
record Lens₁ {ℓ : Level} (S : Set ℓ) (A : Set ℓ) : Set (suc ℓ) where
  constructor ∃-lens
  field
    {C} : Set ℓ
    iso : S ≃ (C × A)
fst : {A B : Set} → Lens₁ (A × B) A
fst = ∃-lens swap⋆equiv
```

Lens in Agda 2

Or, the return of constant-complement lenses:

```
record Lens₁ {ℓ : Level} (S : Set ℓ) (A : Set ℓ) : Set (suc ℓ) where
  constructor ∃-lens
  field
    {C} : Set ℓ
    iso : S ≃ (C × A)
fst : {A B : Set} → Lens₁ (A × B) A
fst = ∃-lens swap⋆equiv
sound : {ℓ : Level} {S A : Set ℓ} → Lens₁ S A → GS-Lens S A
```

Lens in Agda 2

Or, the return of constant-complement lenses:

```
record Lens₁ {ℓ : Level} (S : Set ℓ) (A : Set ℓ) : Set (suc ℓ) where
  constructor ∃-lens
  field
    {C} : Set ℓ
    iso : S ≃ (C × A)
fst : {A B : Set} → Lens₁ (A × B) A
fst = ∃-lens swap⋆equiv
sound : {ℓ : Level} {S A : Set ℓ} → Lens₁ S A → GS-Lens S A
```

complete requires moving to Setoid – see online code.

```
_≈_under_ : ∀ {ℓ} {S A : Set ℓ} → (s t : S) (l : GS-Lens S A) → Set ℓ
_≈_under_ s t l = ∀ a → set l s a ≡ set l t a
```

Exploiting type equivalences

```
module _ {A B D : Set} where
  l₁ : Lens₁ A A
  l₁ = ∃-lens uniti⋆equiv
  l₂ : Lens₁ (B × A) A
  l₂ = ∃-lens id≃
  l₃ : Lens₁ (B × A) B
  l₃ = ∃-lens swap⋆equiv
  l₄ : Lens₁ (D × (B × A)) A
  l₄ = ∃-lens assocl⋆equiv
  l₅ : Lens₁ ⊥ A
  l₅ = ∃-lens factorzequiv
  l₆ : Lens₁ ((D × A) ⊎ (B × A)) A
  l₆ = ∃-lens factorequiv
```

uniti⋆equiv : $A \simeq (\top \times A)$
id≃ : $A \simeq A$
swap⋆equiv : $A \times B \simeq B \times A$
assocl⋆equiv : $(A \times B) \times C \simeq A \times (B \times C)$
factorzequiv : $\bot \simeq (\bot \times A)$
factorequiv : $((A \times D) \uplus (B \times D)) \simeq ((A \uplus B) \times D)$