Scale by the Bay
San Francisco, November 17th, 2017

Adjunctions in everyday life

What we talk about when we talk about monads

Rúnar Bjarnason (@runarorama)

runar@takt.com

Scala

Functional Programming

Paul Chiusano
Rúnar Bjarnason

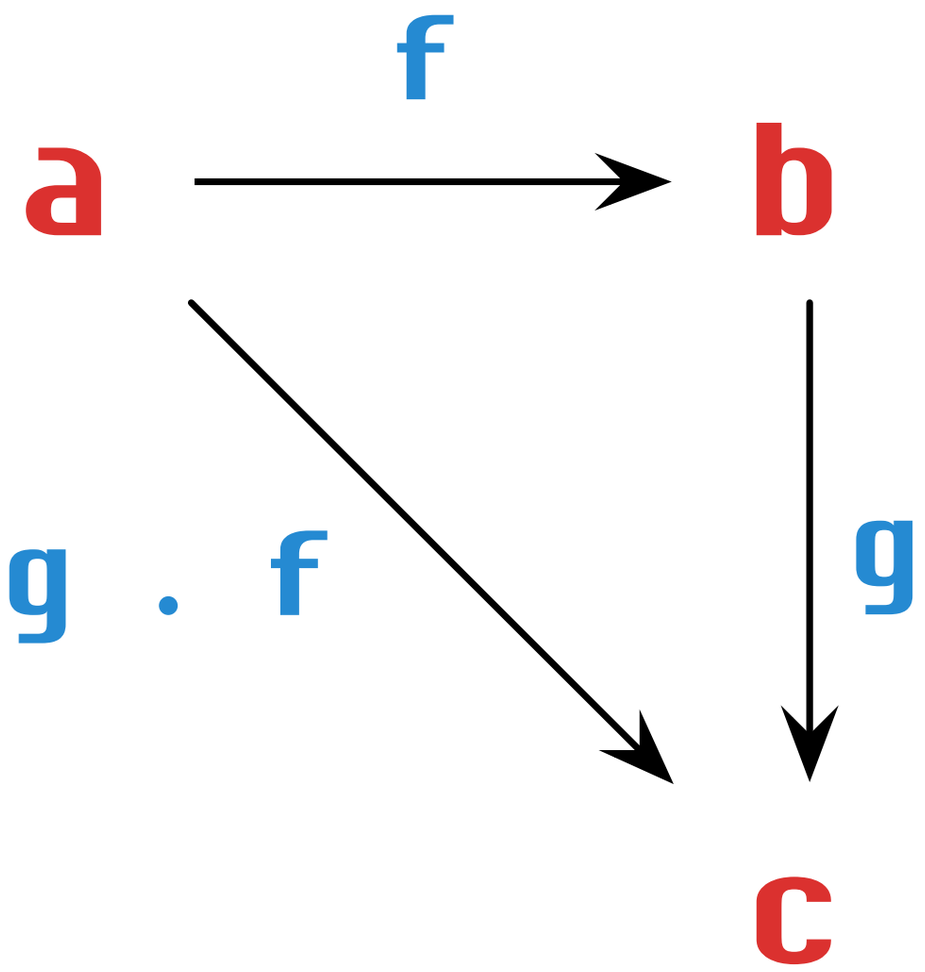
Foreword by Martin Odersky

MANNING

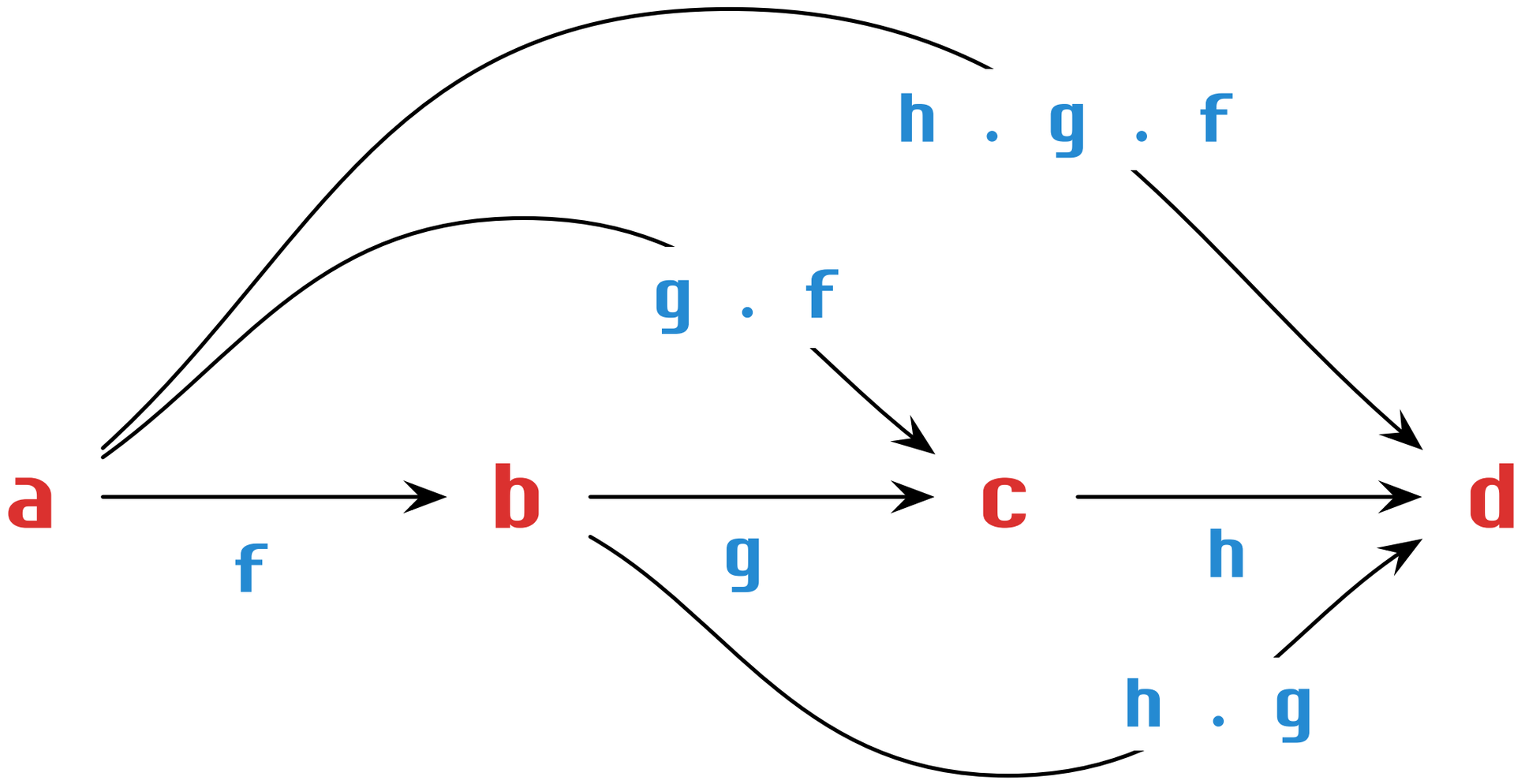
The Plan

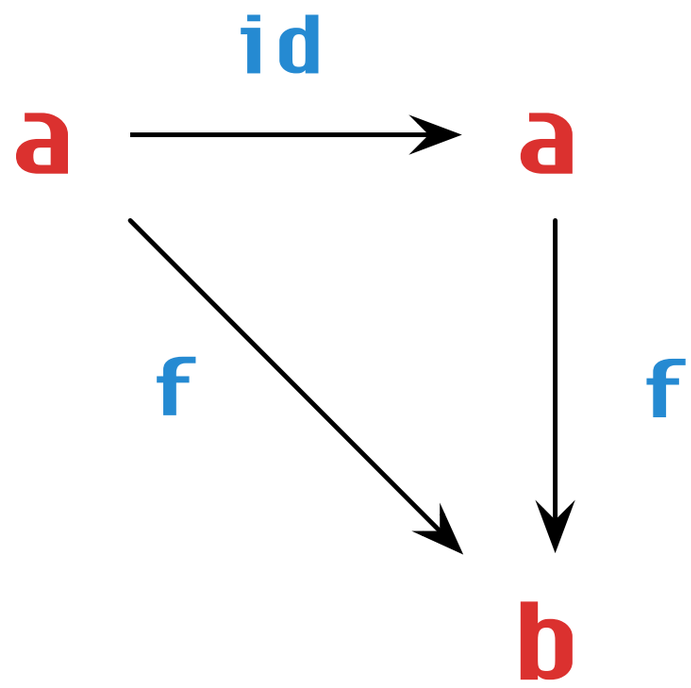
1. I'm going to teach you category theory.
2. I’m going to show you the same pattern several times and say “adjunction” a lot.
3. You’re going to start seeing adjunctions everywhere.
4. You're going to tell me about the adjunctions that you discover.

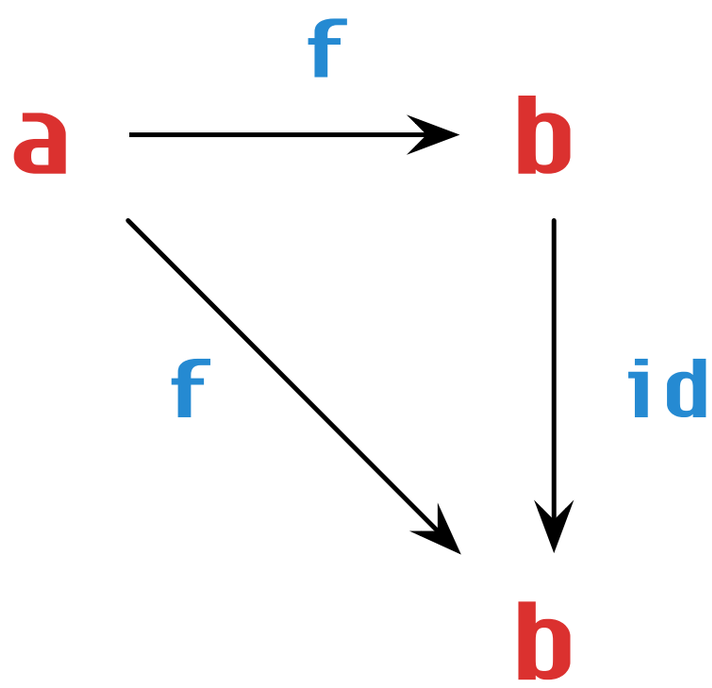
Categories




```
\x -> g (f x)
```


h . g . f
g . f
a
b
c
d
f
g
h
h . g


id
a
a
f
f
b


f
a
b
f
id
b

Category

- Objects
- Arrows between objects
- Composition of arrows
  - Which is associative
  - And has an identity

The category *Hask*

- Objects: Haskell types
- Arrows: Haskell functions
- Composition: function composition
  - `λx → f (g (h x))`
  - `λx → x`

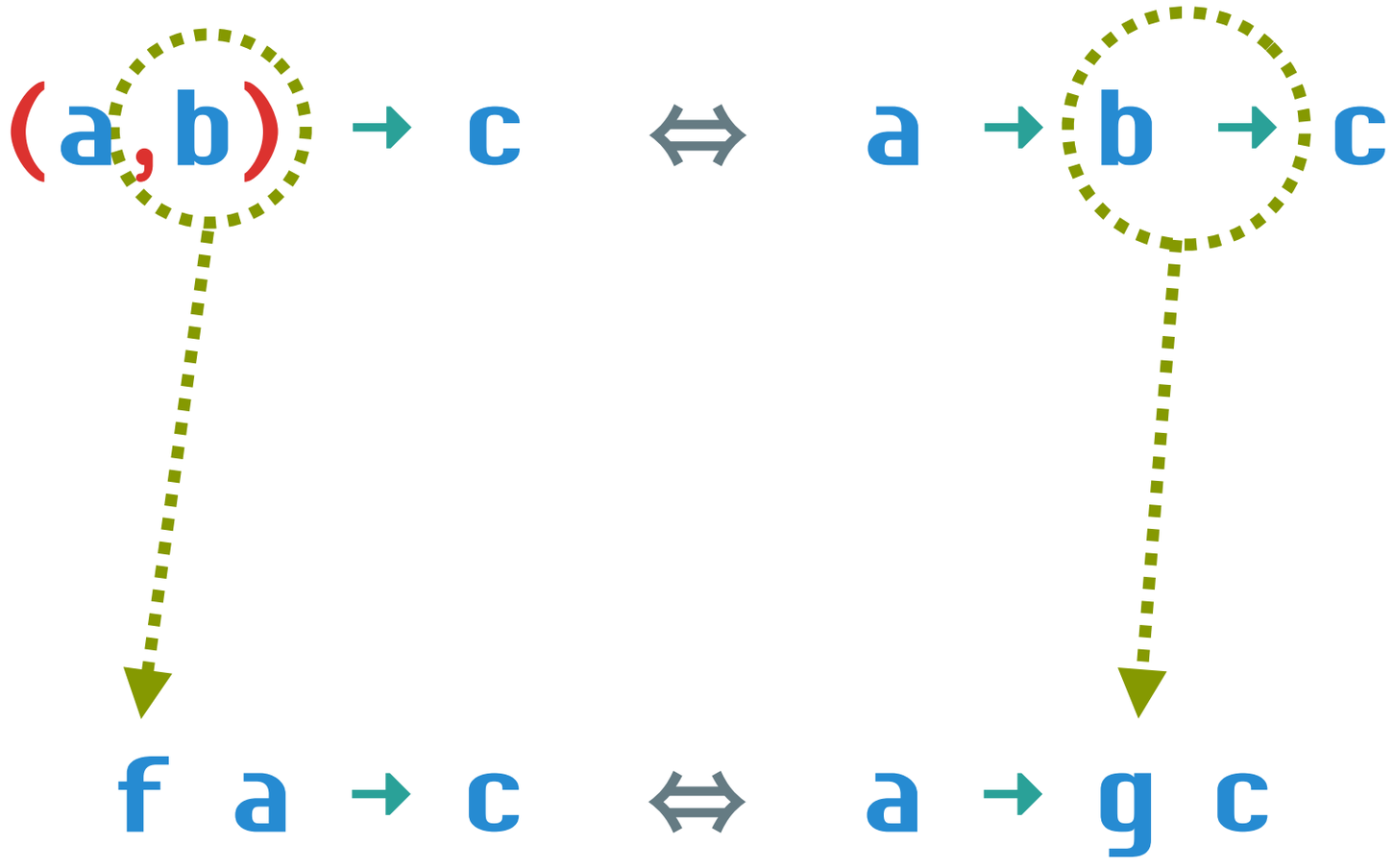
Adjunctions

- “Adjoint functors arise everywhere”
- “An adjoint functor is a way of giving the most efficient solution to some problem via a method which is formulaic.”
- Dually, finding the most difficult problem that such a formulaic method solves.

```haskell
curry :: ((a,b) → c) → a → b → c
curry f a b = f (a,b)
```

```haskell
uncurry :: (a → b → c) → (a,b) → c
uncurry f (a,b) = f a b
```

$$(a,b) \to c \quad \Leftrightarrow \quad a \to b \to c$$


(a,b) → c ⇔ a → b → c
f a → c ⇔ a → g c

```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a → b) → a → g b
    rightAdjunct :: (a → g b) → f a → b
```

```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a → b) → a → g b
    rightAdjunct :: (a → g b) → f a → b
```

The law of adjunctions:

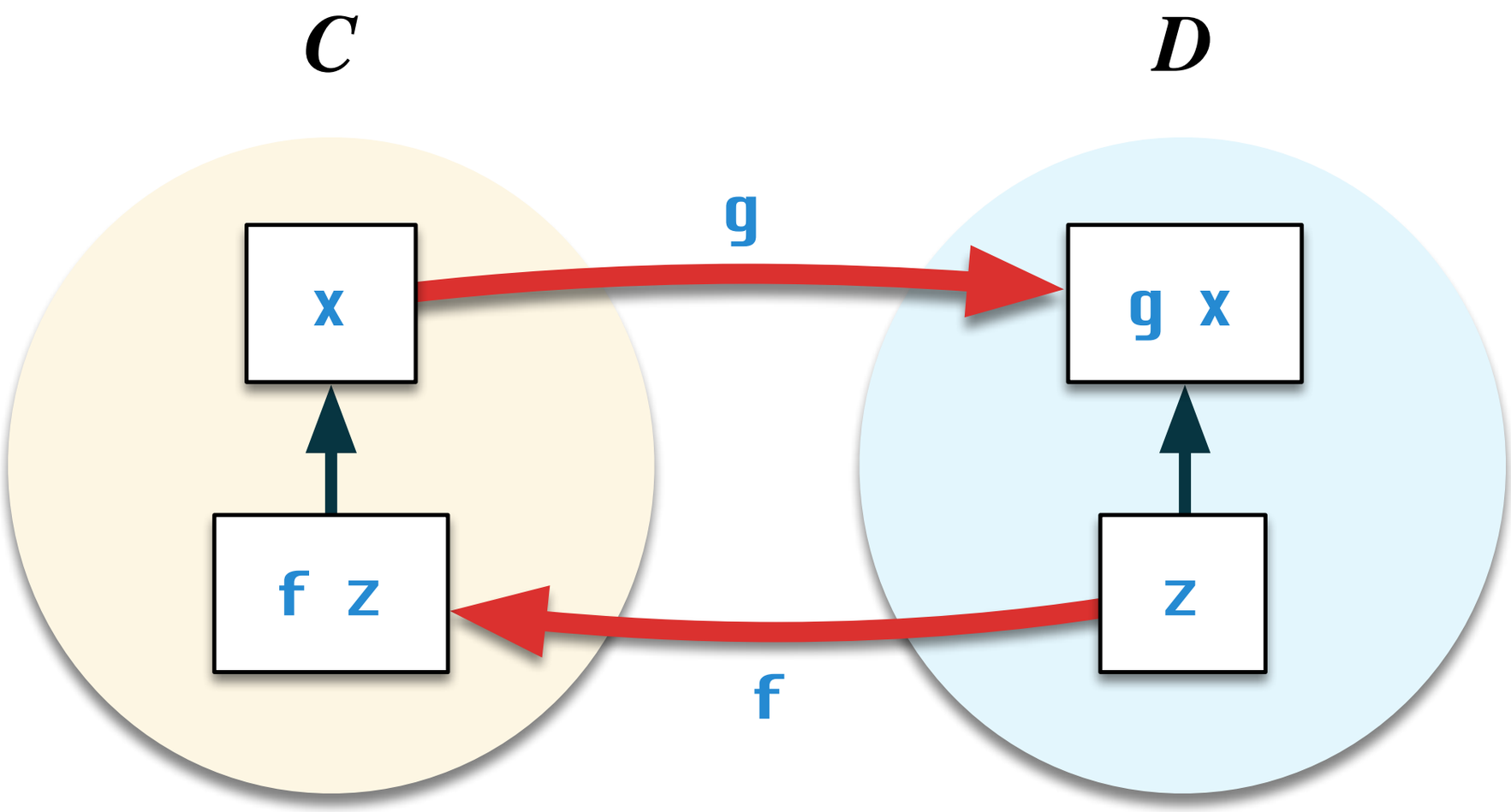
```haskell
leftAdjunct . rightAdjunct = id
rightAdjunct . leftAdjunct = id
```

```haskell
instance Adjunction (,s) (s→) where
  leftAdjunct  = curry
  rightAdjunct = uncurry
```

```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a → b) → a → g b
    rightAdjunct :: (a → g b) → f a → b

    unit   :: a → g (f a)
    unit = leftAdjunct id

    counit :: f (g a) → a
    counit = rightAdjunct id
```

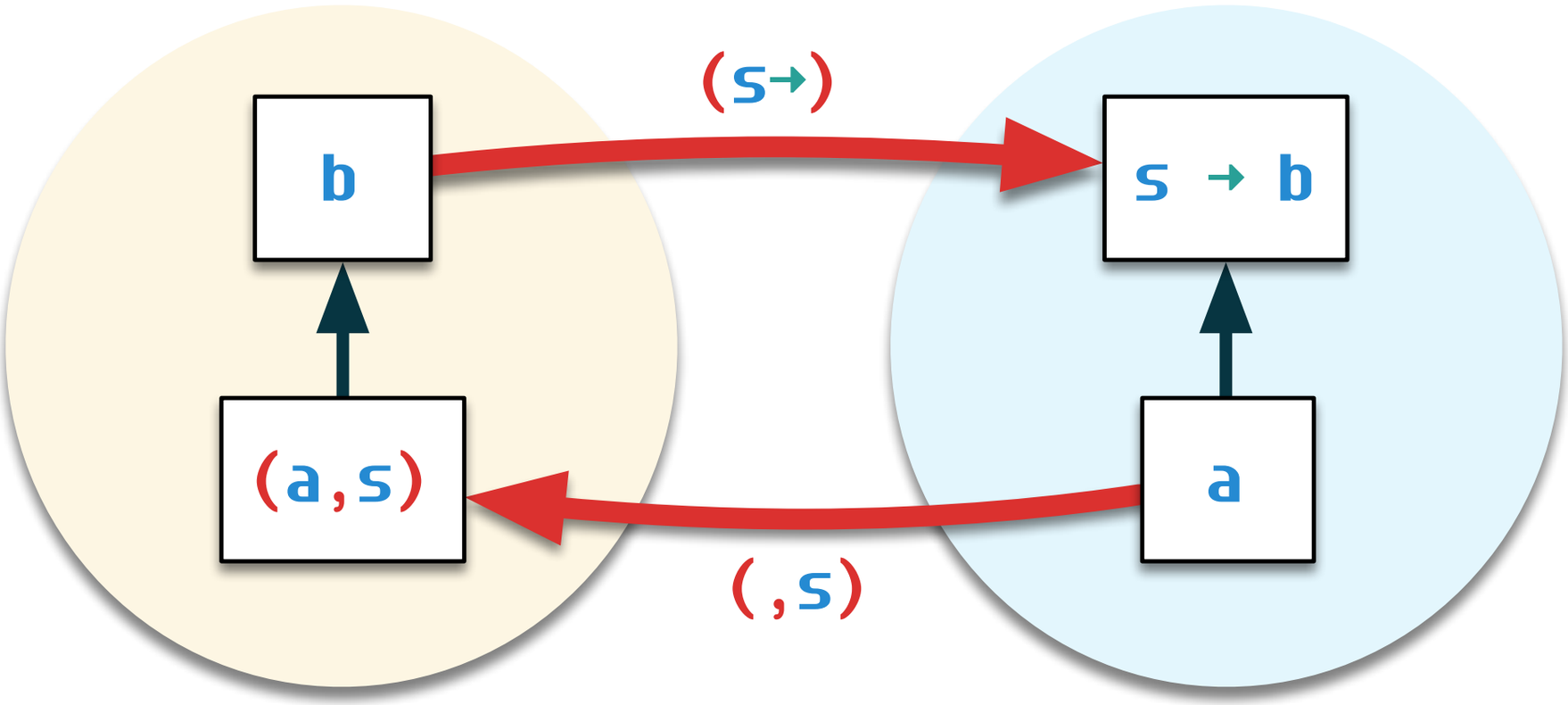
```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a → b) → a → g b
    leftAdjunct h = fmap h . unit

    rightAdjunct :: (a → g b) → f a → b
    rightAdjunct h = counit . fmap h

    unit   :: a → g (f a)
    unit = leftAdjunct id

    counit :: f (g a) → a
    counit = rightAdjunct id
```

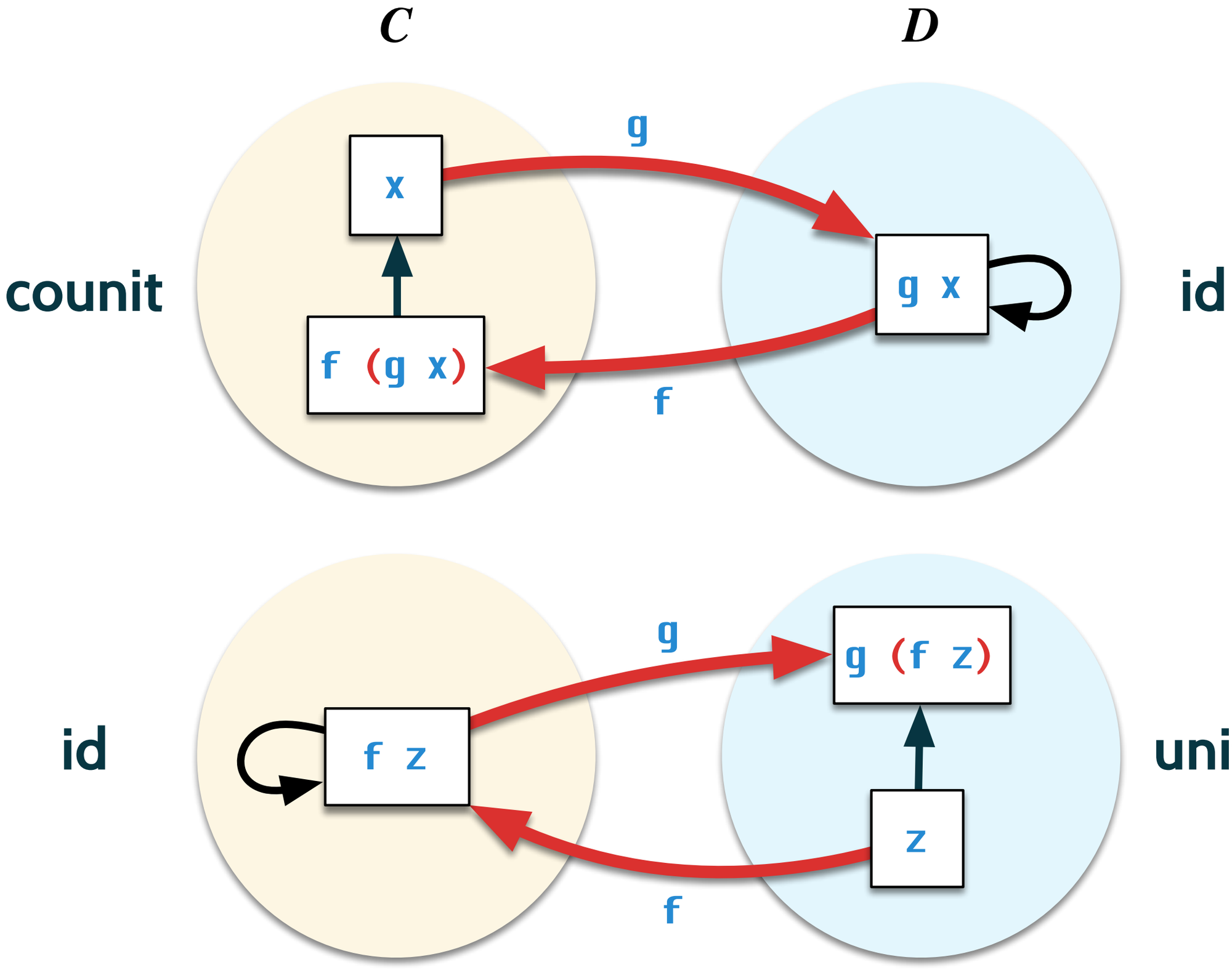
$$f \dashv g$$



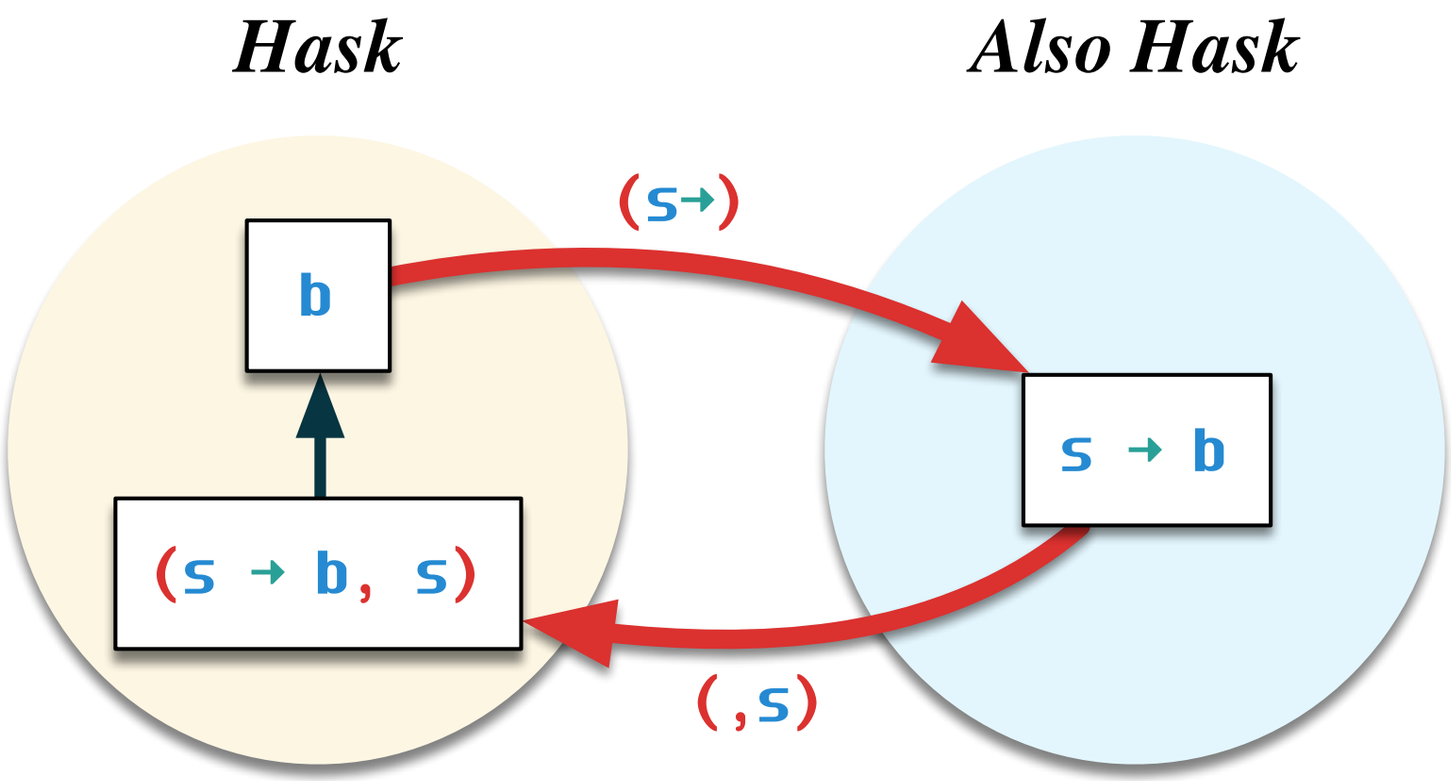
$$(,s) \dashv (s\to)$$

Hask *Also Hask*

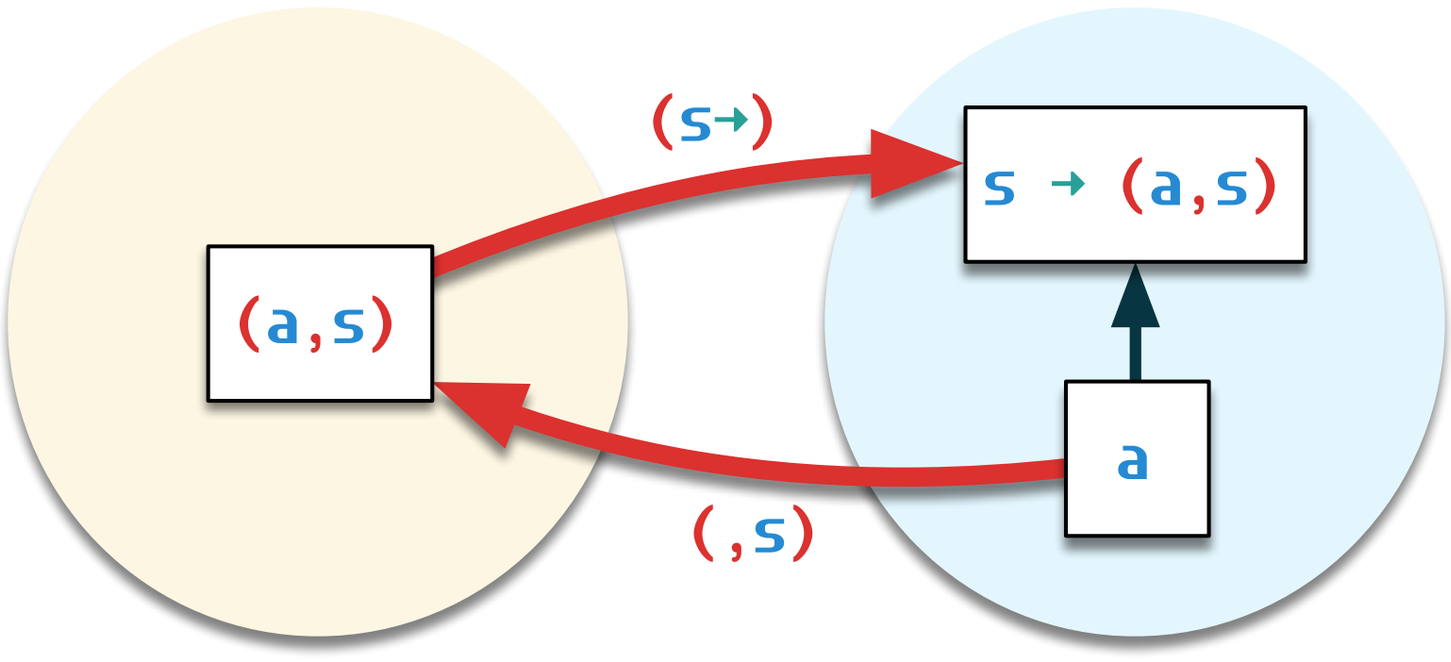



C
D
counit
x
g
g x
id
f (g x)
f
id
g
g (f z)
f z
unit
z
f

counit =
uncurry id



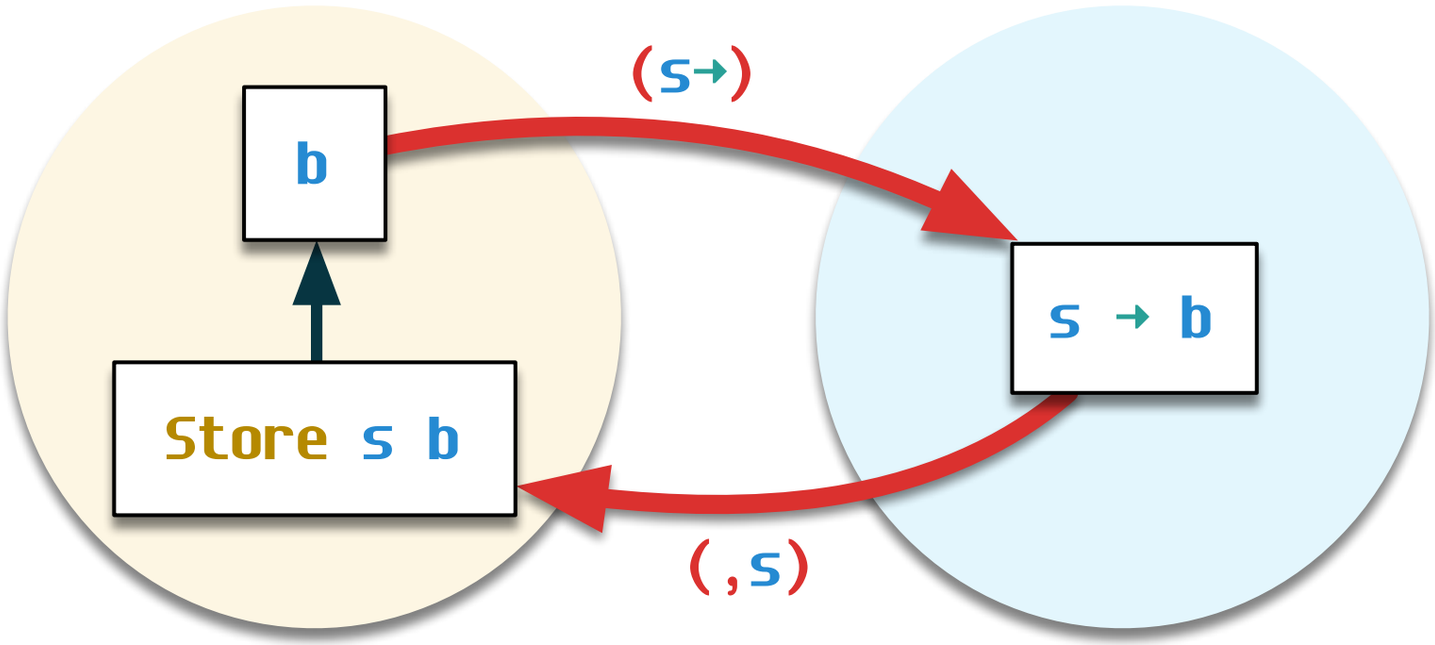
id

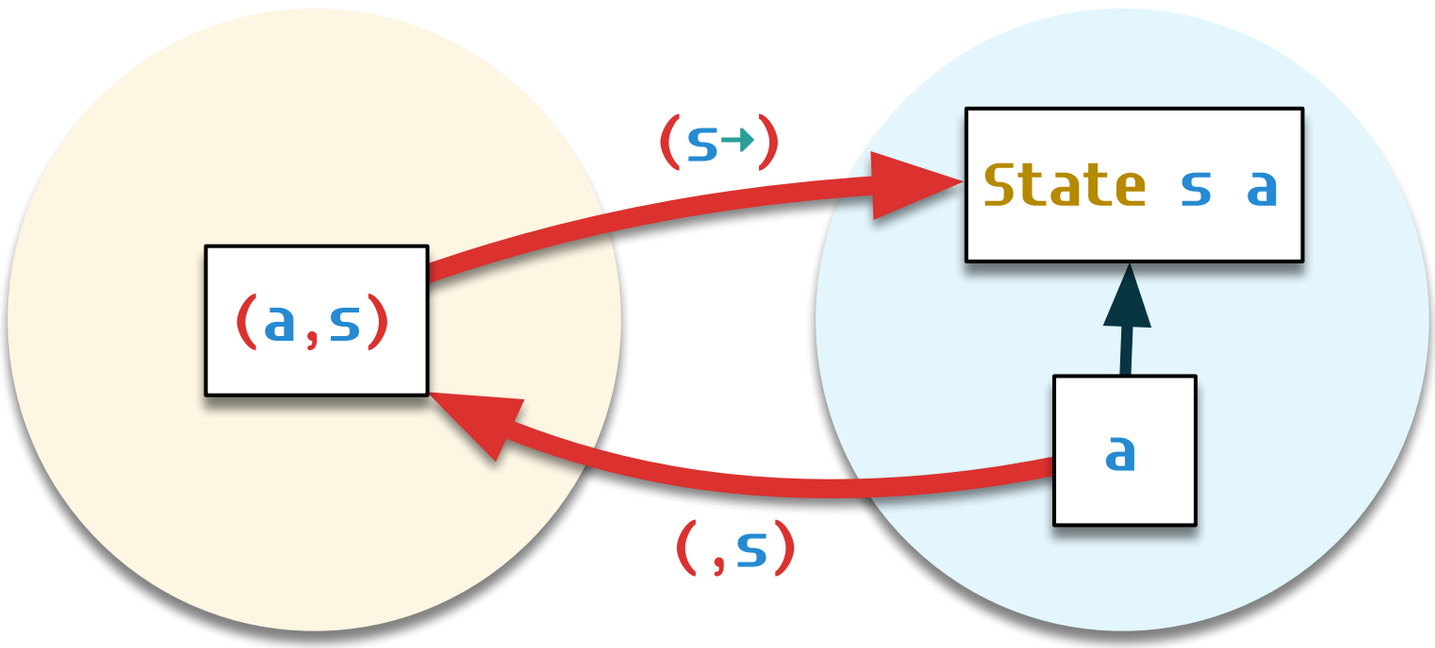


unit =
curry id

Hask *Also Hask*

counit





unit

state before

state after

```
type State s a = s → (a,s)
```

output

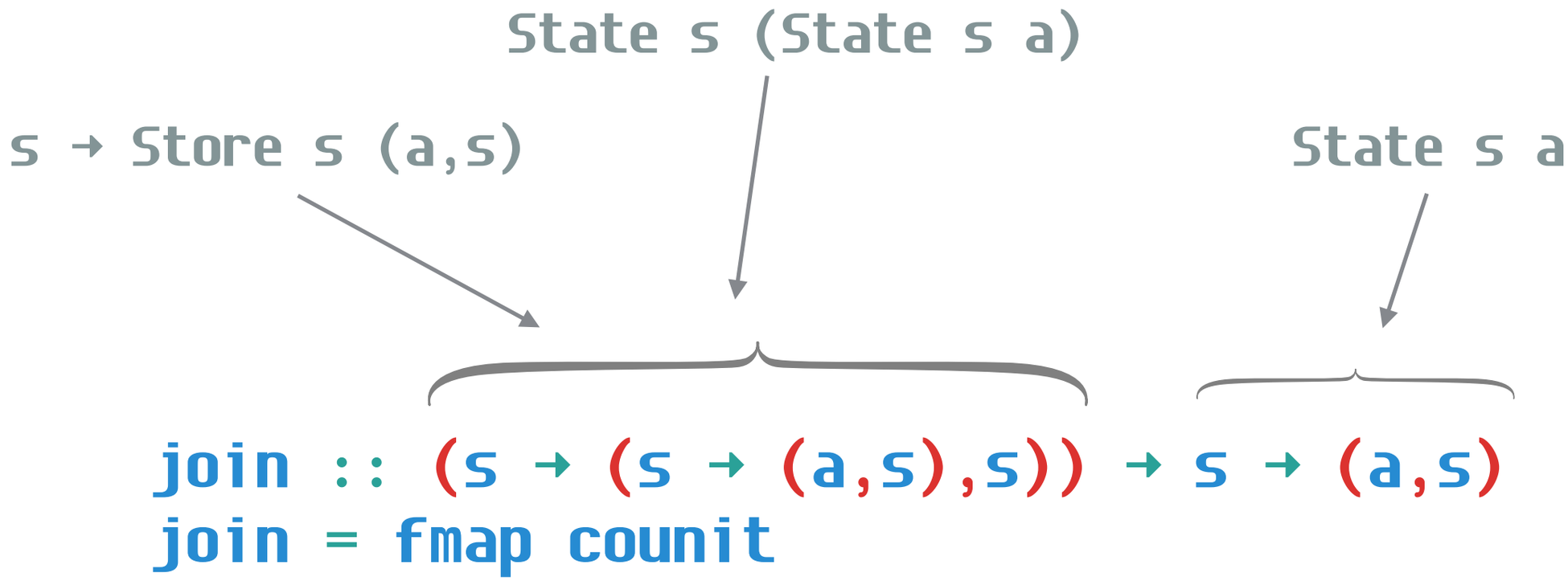
index

cursor

```
type Store s a = (s → a,s)
```

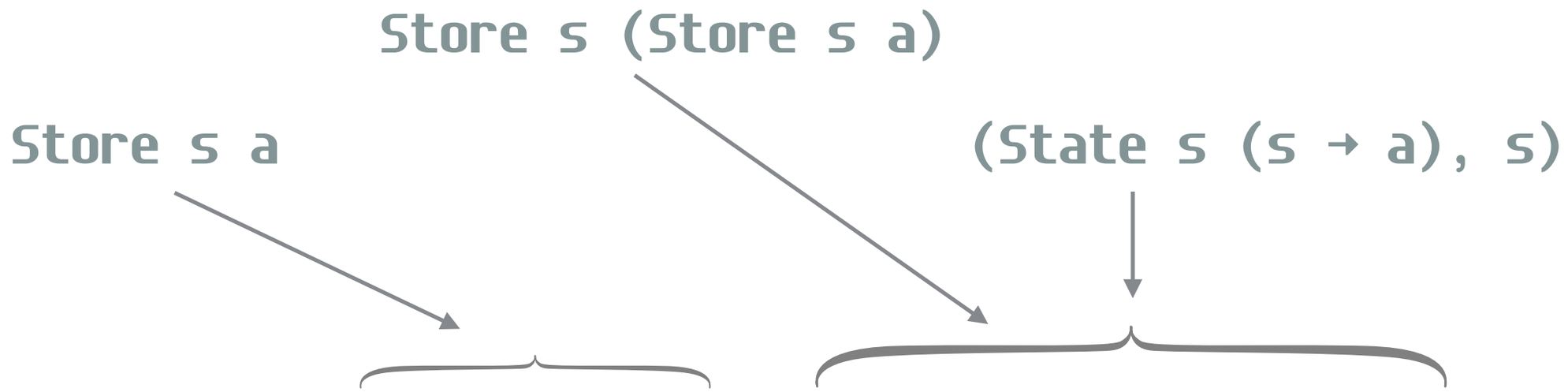
stored value

```
type Bitmap2D = Store (Int,Int) Color
```




```
join :: (s → (s → (a,s),s)) → s → (a,s)
join = fmap counit
```

```haskell
class (Functor m) => Monad m where
  return :: a → m a
  join :: m (m a) → m a
  (>>=) :: m a → (a → m b) → m b
  m >>= f = join $ fmap (unit . f) m
```




```
duplicate :: (s → a, s) → (s → (s → a, s), s)
duplicate = fmap unit
```

```haskell
class (Functor w) => Comonad w where
  extract   :: w a → a
  duplicate :: w a → w (w a)
```

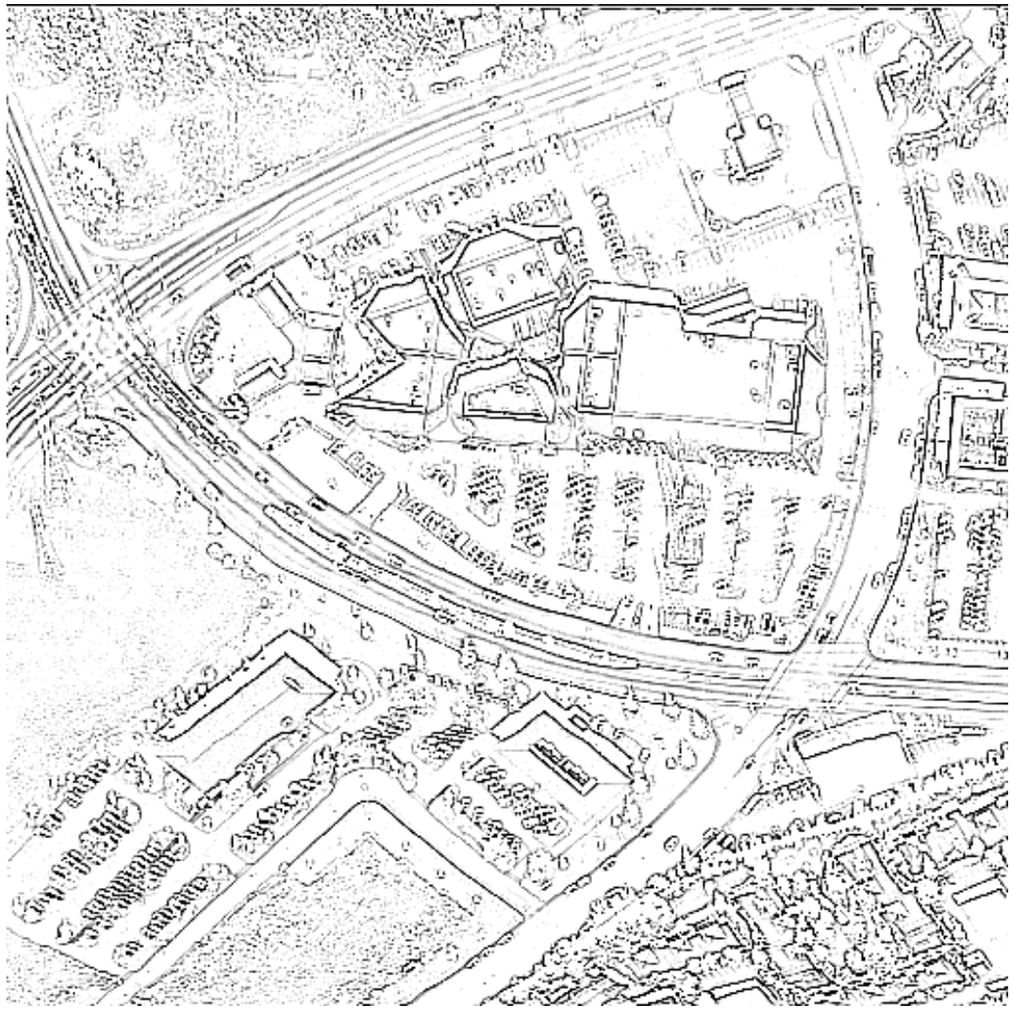
```haskell
(=>>) :: w a → (w a → b) → w b
w =>> f = fmap f (duplicate w)
```

A comonad extends a local computation to a global one.

```
type Bitmap2D = Store (Int,Int) Int
```

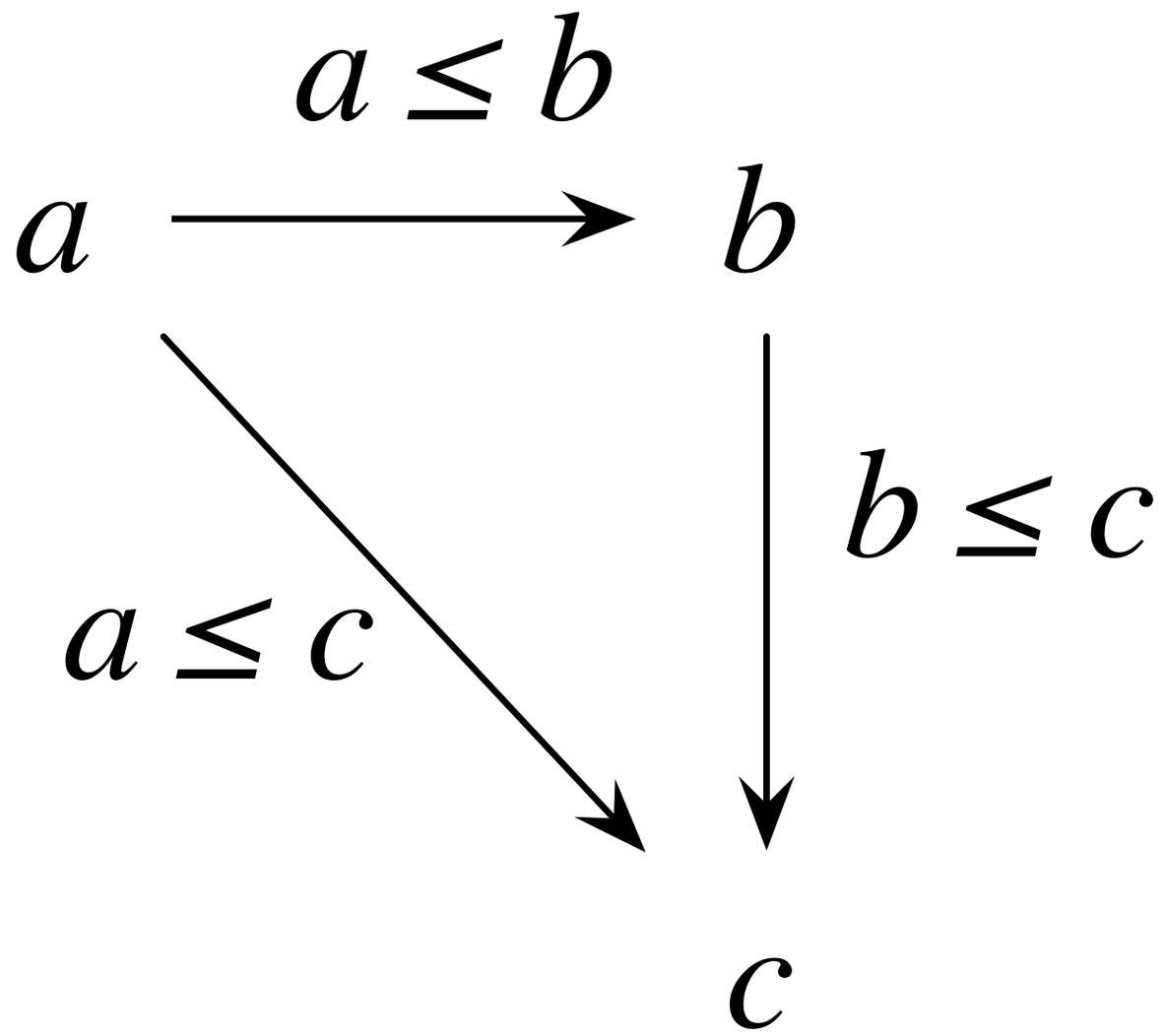
```
lowPass :: Bitmap2D → Bitmap2D
lowPass bmp = bmp =>> mean
```

```haskell
edges :: Bitmap2D → Bitmap2D
edges bmp = bmp >>= \b →
  extract b - extract (lowPass b)
```



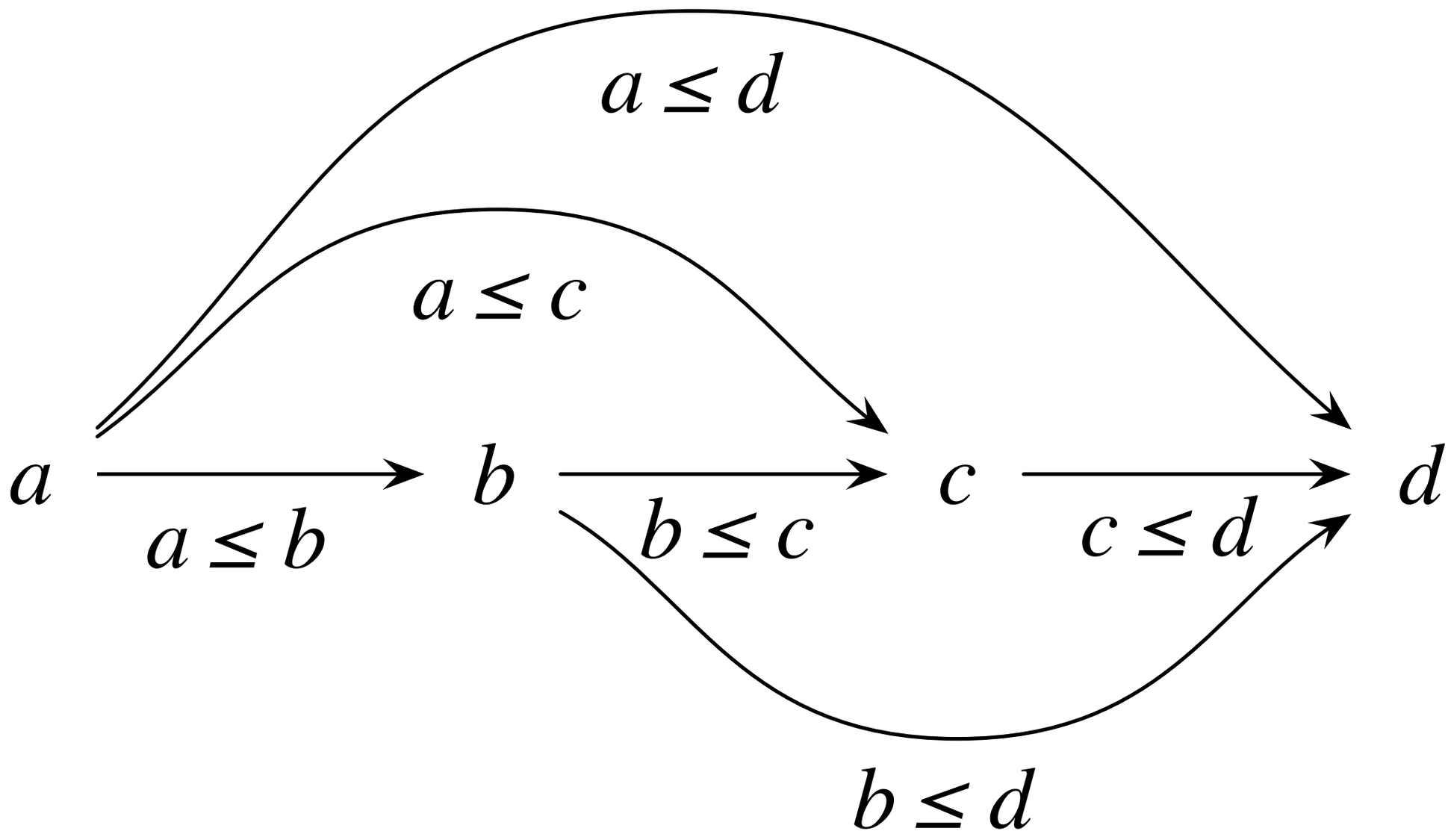
$$(a,b) \to c \quad \Leftrightarrow \quad a \to b \to c$$

Category of integers


$a \leq b$
a
b
$b \leq c$
$a \leq c$
c

$$a \leq a$$

$$a \longrightarrow a$$


$a \leq d$
$a \leq c$
a
$a \leq b$
b
$b \leq c$
c
$c \leq d$
d
$b \leq d$

$$x,y,z :: \mathbb{Z}$$
given $y > 0$

$$(z*y \leq x) \Leftrightarrow (z \leq x/y)$$

$$(z * y \leq x) \Leftrightarrow (z \leq x/y)$$

unit: $(x/y) * y \leq x$

counit: $z \leq (z * y)/y$

Galois Connection

$$z * y \leq x \iff z \leq x / y$$

$$f\ z \leq x \iff z \leq g\ x$$

unit: $f\ (g\ x) \leq x$

counit: $z \leq g\ (f\ z)$

Conceptualization as an adjunction

Collections:

c1 ⊆ **c2** when **c2** contains all of **c1**

Descriptions:

d1 ≤ **d2** when **d1** is more specific than **d2**

$$\mathtt{describe}\ (\mathtt{examples}\ d) \preceq d$$

$$e \subseteq \mathtt{examples}\ (\mathtt{describe}\ e)$$

$$\texttt{describe} \dashv \texttt{examples}$$

$$\texttt{describe}\ e \preceq c \Leftrightarrow e \subseteq \texttt{examples}\ c$$

a simple API
design problem

```
indexOf :: Eq a ⇒ a → [a] → Integer
```

```
(-1)

Infinity

(-Infinity)

NaN

null :: forall a. a
```

```
class Pointed a where
  point :: a
```

Can we turn any type into a pointed type in a formulaic, universal way?

Making no ad hoc choices?

There's a *forgetful* functor
U*:* ***PointedTypes*** → ***Types***

U[x] "forgets" the point of x and gives the underlying type.

$\mathbf{P} \dashv \mathbf{U}$

U: ***PointedTypes*** → ***Types*** has a *left adjoint:*

P: ***Types*** → ***PointedTypes***

for any type x, P[x] is a pointed type

$$P \dashv U$$

$$P\ a \to b \iff a \to U\ b$$

$$P\ a \to b \iff a \to b$$

```haskell
rightAdjunct :: Pointed b => (a → b) → P a → b

leftAdjunct  :: (P a → b) → a → b
```

```haskell
rightAdjunct :: Pointed b => (a → b) → P a → b

leftAdjunct  :: (P a → b) → a → b


counit :: Pointed b => P b → b

unit   :: a → P a
```

```
rightAdjunct :: b → (a → b) → P a → b

leftAdjunct  :: (P a → b) → a → b


counit :: b → P b → b

unit   :: a → P a
```

```
rightAdjunct :: (a → b) → P a → b → b

leftAdjunct  :: (P a → b) → a → b


counit :: P b → b → b

unit   :: a → P a
```

```haskell
newtype P a = P {
  foldP :: (a → b) → b → b
}
```

```haskell
counit = flip foldP id
unit a = P $ λf _ → f a
```

```
foldP :: P a → (a → b) → b → b
```

```
maybe :: b → (a → b) → Maybe a → b
```

```haskell
data Maybe a = Nothing | Just a

maybe :: b → (a → b) → Maybe a → b
maybe b f Nothing = b
maybe b f (Just a) = f a
```

```haskell
counit b = maybe b id
unit = Just
```

```haskell
join :: Maybe (Maybe a) → Maybe a
join = counit
```

```haskell
duplicate :: Maybe a → Maybe (Maybe a)
duplicate = fmap unit
```

```
indexOf :: Eq a ⇒ a → [a] → Maybe Integer
```

```haskell
class Monoid a where
  mempty :: a
  mappend :: a → a → a
```

Can we turn any type into a monoid in a formulaic, universal way?

Making no ad hoc choices?

$\mathbf{M} \dashv \mathbf{U}$

U: ***Monoids*** → ***Types*** has a *left adjoint:*

M: ***Types*** → ***Monoids***

for any type x, M[x] is a monoid

```
rightAdjunct :: Monoid b => (a → b) → M a → b

leftAdjunct  :: (M a → b) → a → b

counit :: Monoid b => M b → b

unit   :: a → M a
```

```haskell
rightAdjunct :: Monoid b => (a → b) → M a → b

leftAdjunct  :: (M a → b) → a → b

counit :: Monoid b => M b → b

unit   :: a → M a
```

```haskell
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

```
foldMap :: Monoid m => (a → m) → [a] → m
```

Can we turn any functor into a monad in a formulaic, universal way?

Making no ad hoc choices?

Free ⊣ Forget

Free: ***Monads*** → ***Functors*** has a *left adjoint:*

Forget: ***Functors*** → ***Monads***

for any functor F, Free[F] is a monad

```haskell
rightAdjunct :: Monad m =>
  (forall a. f a → m a) → Free f a → m a

leftAdjunct ::
  (forall a. Free f a → g a) → f a → g a

counit :: Monad m => Free m a → m a

unit :: a → Free f a
```

Free ⊣ Forget

Free ⊣ Forget ⊣ Cofree

$$F \dashv G$$

$$P \dashv Q$$

$$FP \dashv GQ$$

Generic Programming with Adjunctions
Ralf Hinze

Galculator: functional prototype of a Galois-connection based proof assistant
Paulo Silva, José Oliveira

What does adjunction *mean*?

- Generates a solution that naturally fits the problem.
- Resolves tension between tradeoffs.
- Finds an optimal surface between a problem space and solution space.

What does adjunction *mean*?

- Compares two categories.
- Simulates one category in another.

Whenever we're looking for a general, natural, elegant, and efficient solution, we can express the problem as a functor and find its adjoint.

Adjunctions are everywhere.

Let’s find them.

Questions?