Adjunctions in everyday life

What we talk about when we talk about monads

Rúnar Bjarnason (@runarorama)

TAKT

runar@takt.com

Functional Programming

IN

scala

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

$$
\begin{tikzcd}
a \arrow[r, "f"] \arrow[dr, "g\ .\ f"'] & b \arrow[d, "g"] \\
& c
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
a \arrow[r, "f"'] \arrow[rr, bend left, "g\ .\ f"] \arrow[rrr, bend left=60, "h\ .\ g\ .\ f"] & b \arrow[r, "g"'] \arrow[rr, bend right, "h\ .\ g"'] & c \arrow[r, "h"'] & d
\end{tikzcd}
$$

$$
\begin{tikzcd}
a \arrow[r, "id"] \arrow[dr, "f"'] & a \arrow[d, "f"] \\
& b
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
a \arrow[r, "f"] \arrow[dr, "f"'] & b \arrow[d, "id"] \\
& b
\end{tikzcd}
$$

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)

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

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

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

$$f\ a \to c \quad \Leftrightarrow \quad a \to g\ c$$

```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:

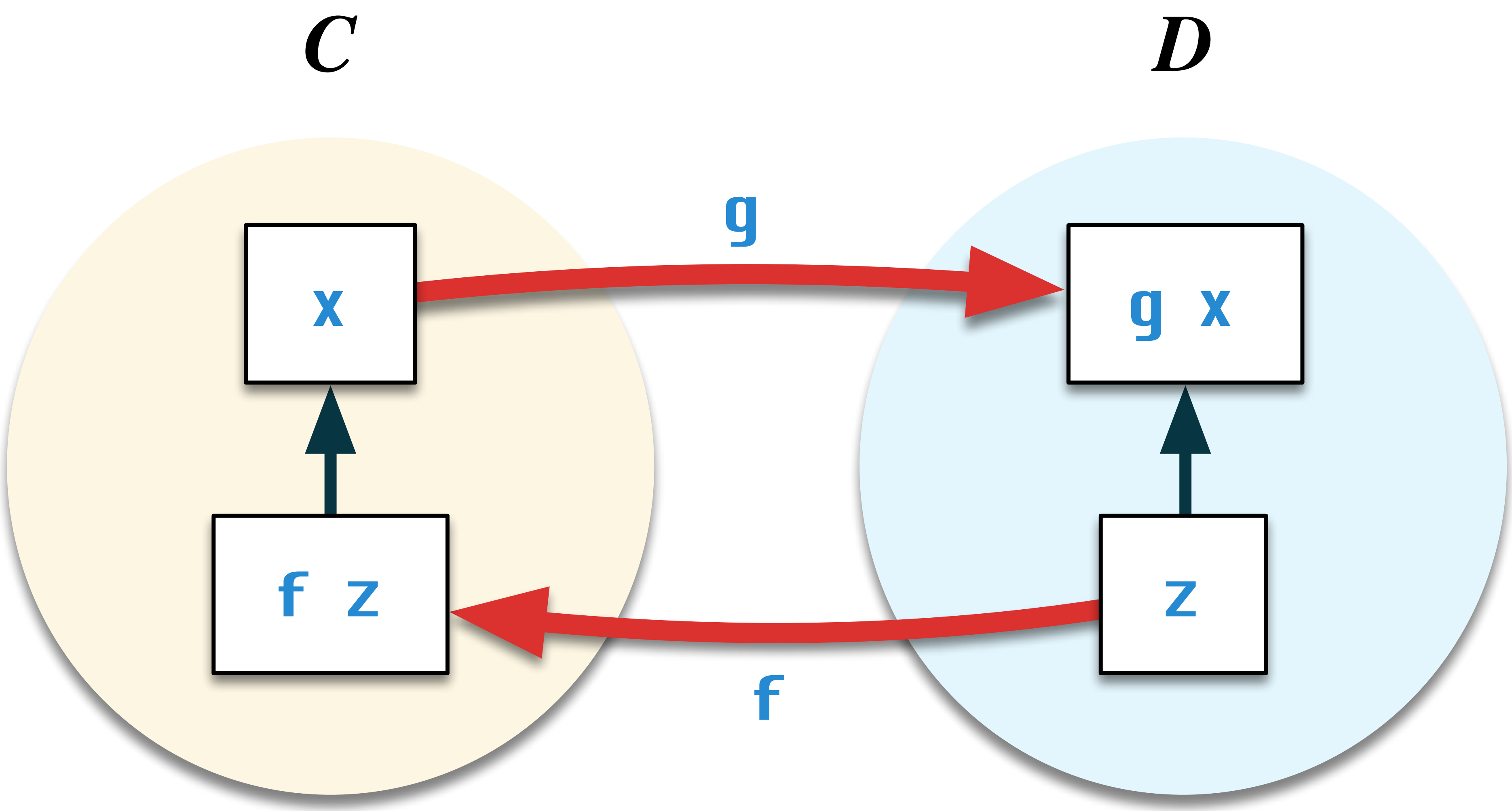
```haskell
leftAdjunct . rightAdjunct = id
rightAdjunct . leftAdjunct = id
```

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

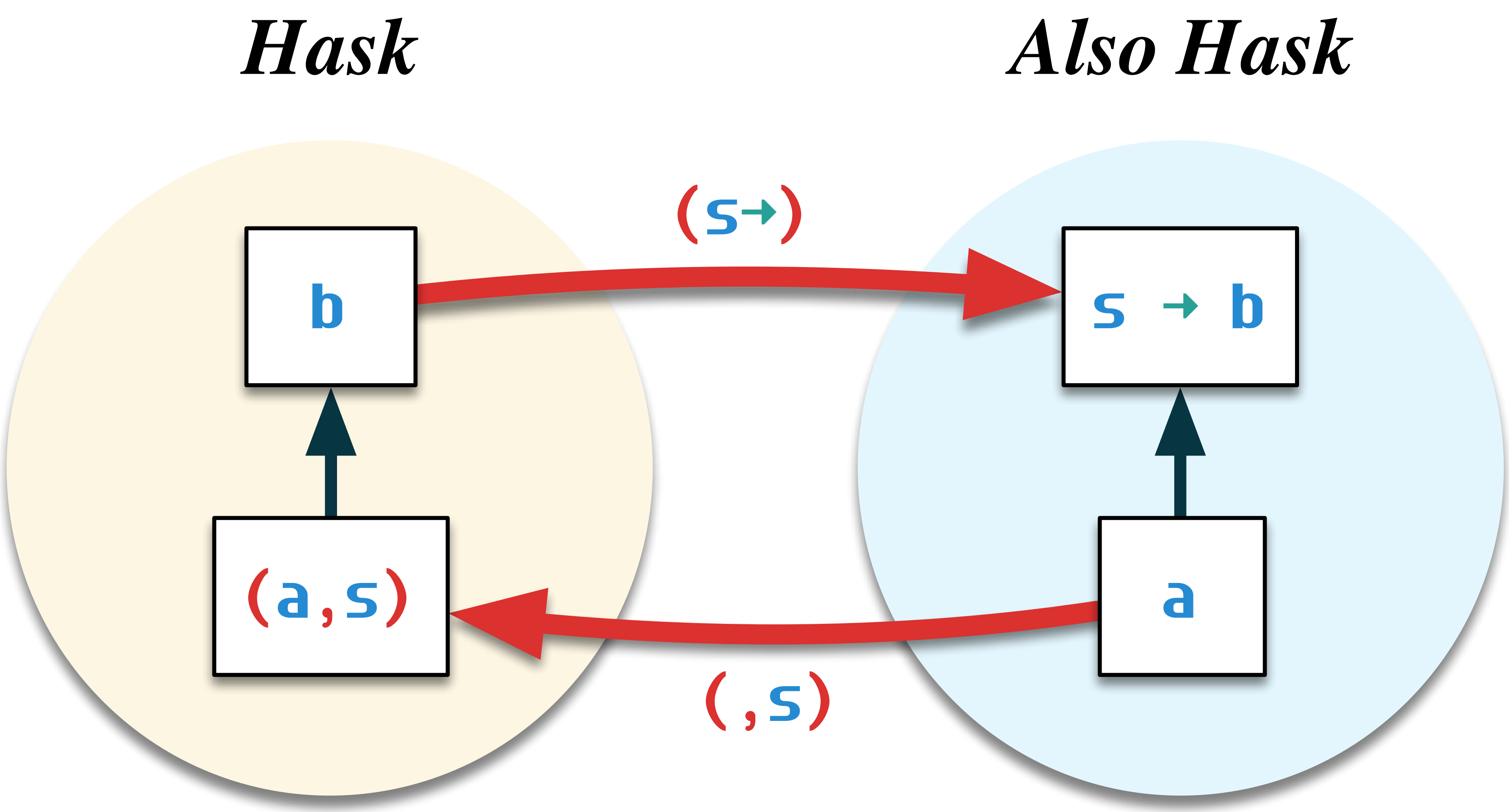
```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a -> b) -> a -> g b
    rightAdjunct :: (a -> g b) -> f a -> b

    unit   :: a -> g (f a)
    unit = leftAdjunct id

    counit :: f (g a) -> a
    counit = rightAdjunct id
```

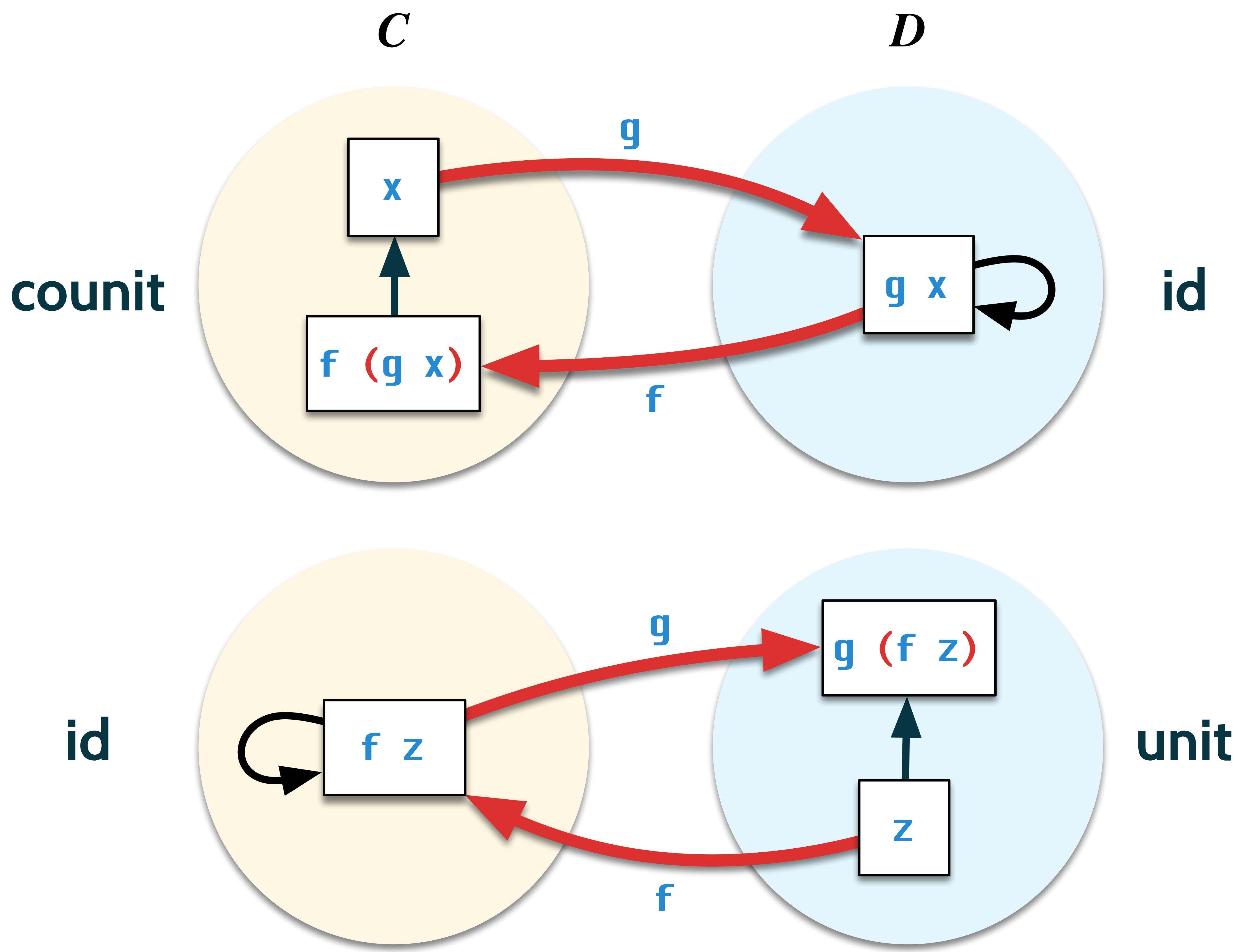
```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a -> b) -> a -> g b
    leftAdjunct h = fmap h . unit

    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
```

$$f \dashv g$$

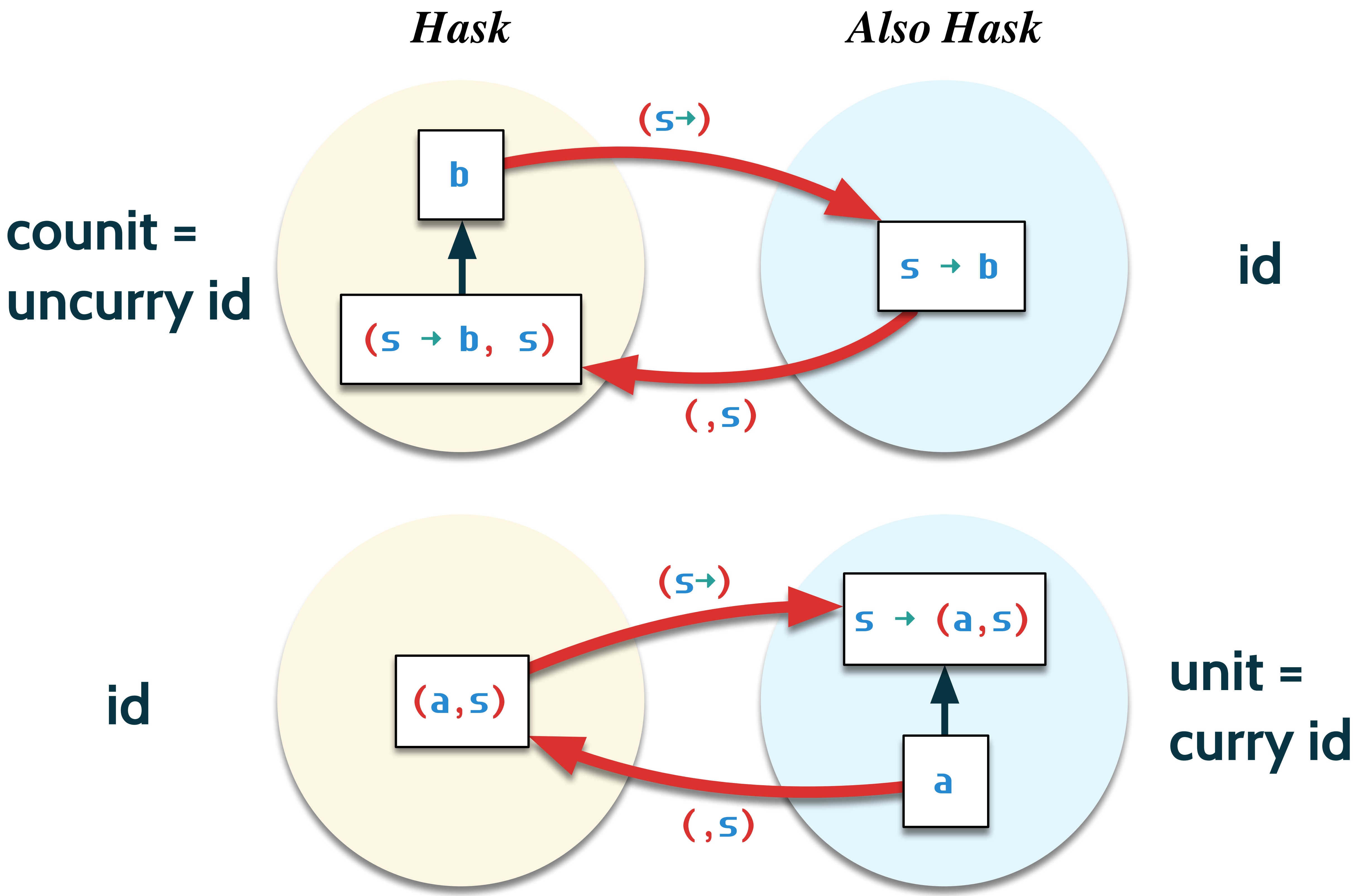


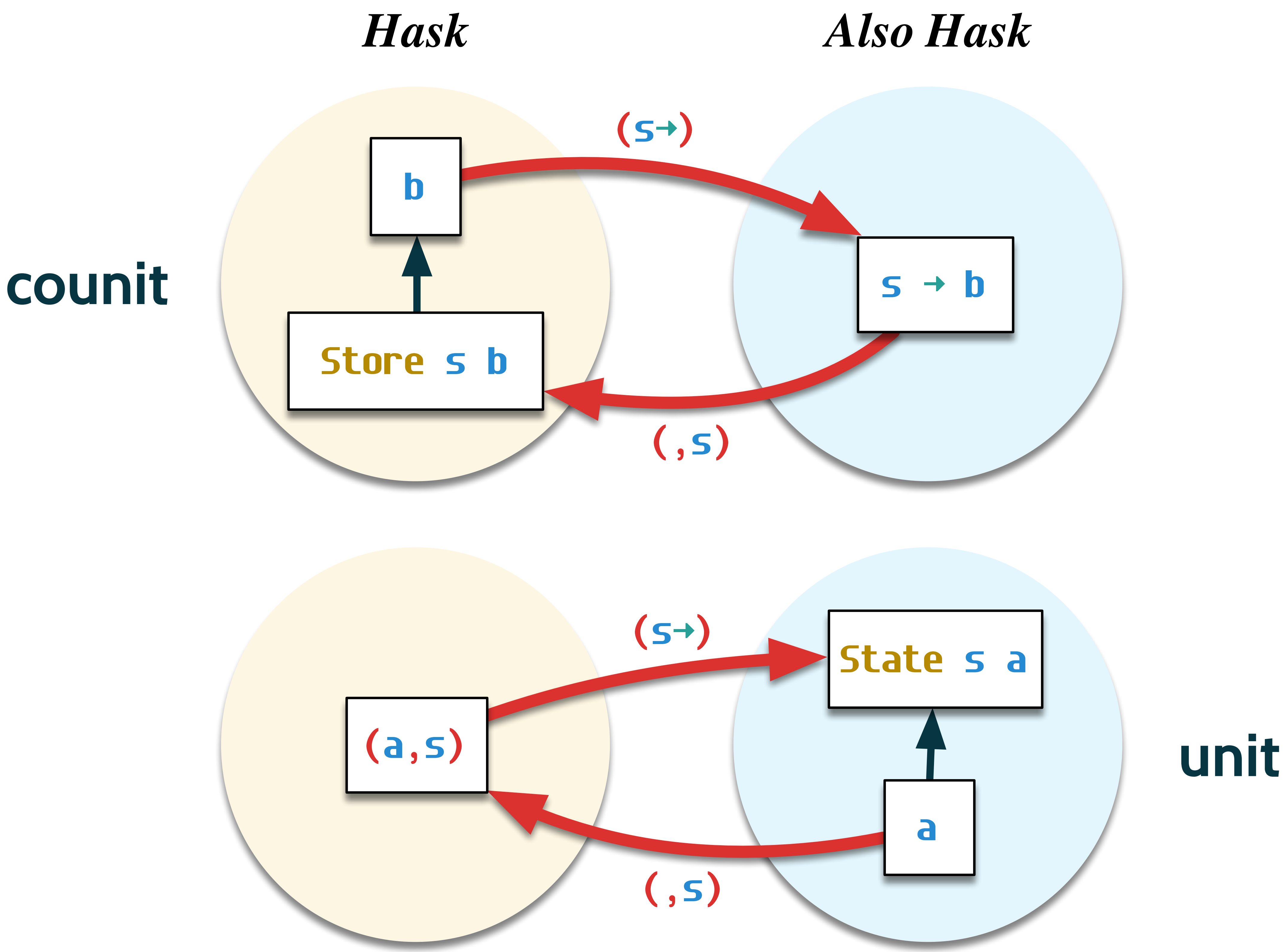
$$(,s) \dashv (s\to)$$





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


Hask
Also Hask
counit =
uncurry id
b
(s→)
s → b
(s → b, s)
(,s)
id
id
(a,s)
(s→)
s → (a,s)
a
(,s)
unit =
curry id



Hask
Also Hask
counit
(s→)
b
s → b
Store s b
(,s)
(s→)
State s a
(a,s)
unit
a
(,s)

state before state after

```haskell
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
```

$$\texttt{join} :: \overbrace{(s \to (s \to (a,s),s))}^{\substack{\texttt{s → Store s (a,s)} \\ \texttt{State s (State s a)}}} \to \overbrace{s \to (a,s)}^{\texttt{State s a}}$$

```
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
```

Store s (Store s a)

Store s a

(State s (s → a), s)

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

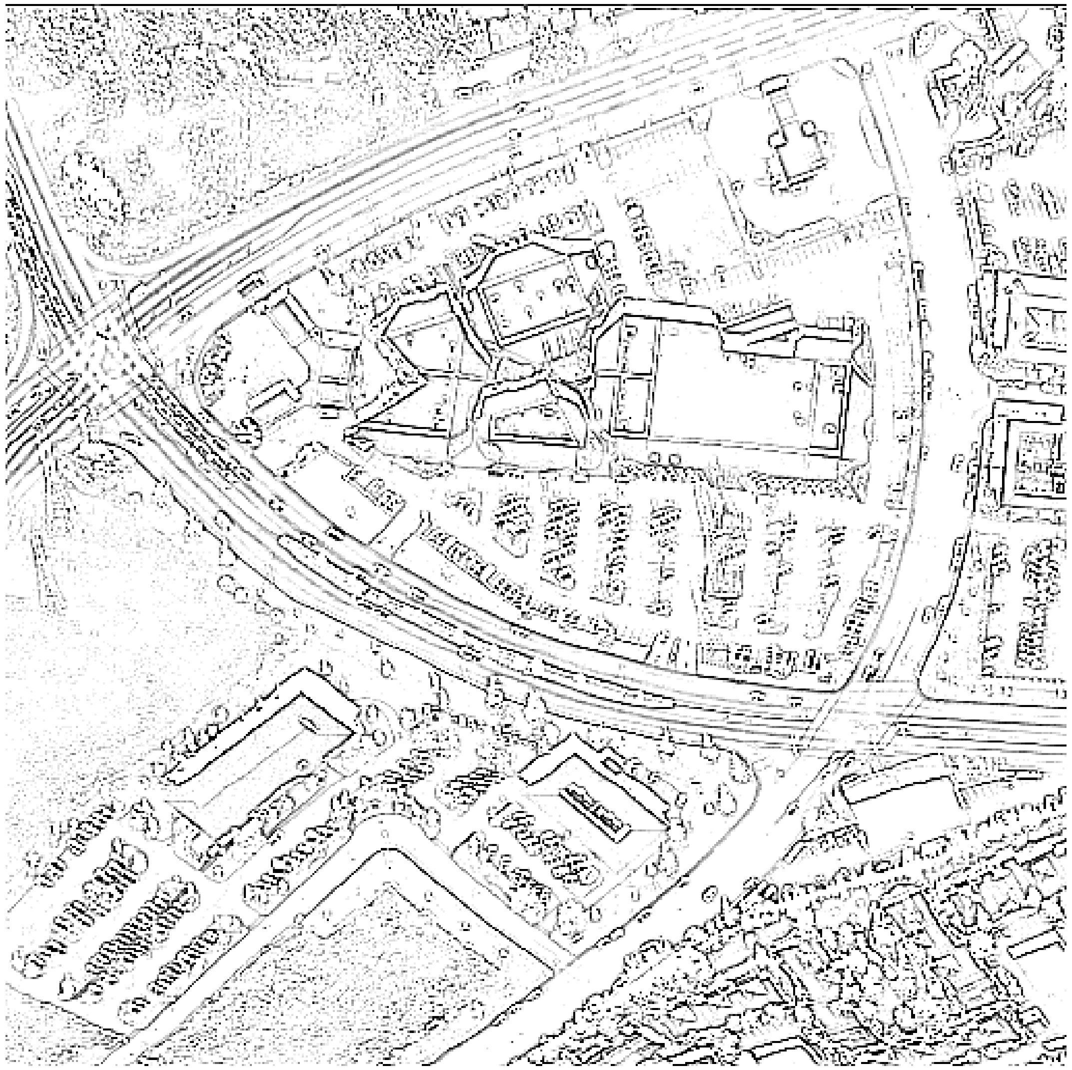
```haskell
class (Functor w) => Comonad w where
  extract   :: w a -> a
  duplicate :: w a -> w (w a)

  (=>>) :: 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
```

```haskell
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

$$\begin{tikzcd} a \arrow[r, "a \leq b"] \arrow[dr, "a \leq c"'] & b \arrow[d, "b \leq c"] \\ & c \end{tikzcd}$$

$$a \xrightarrow{a \leq a} a$$

$$
\begin{tikzcd}
a \arrow[r, "a \leq b"'] \arrow[rr, bend left, "a \leq c"] \arrow[rrr, bend left=60, "a \leq d"] & b \arrow[r, "b \leq c"'] \arrow[rr, bend right, "b \leq d"'] & c \arrow[r, "c \leq d"'] & d
\end{tikzcd}
$$

```
x,y,z :: Integer
given y > 0

(z*y ≤ x) ⇔ (z ≤ x/y)
```

```
(z*y ≤ x) ⇔ (z ≤ x/y)

unit:    (x/y)*y ≤ x
counit: z ≤ (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:

$c1 \subseteq c2$ when $c2$ contains all of $c1$

Descriptions:

$d1 \preceq d2$ 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*** $\to$ ***Types***

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

P $\dashv$ U

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

P: ***Types*** → ***PointedTypes***

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

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

$$\mathbf{P}\,a \to b \quad \Leftrightarrow \quad a \to \mathbf{U}\,b$$

$$P\ a \rightarrow b \Leftrightarrow a \rightarrow b$$

```
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
}

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


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

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

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

```haskell
indexOf :: Eq a => a -> [a] -> Maybe Integer
```

```
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

```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
```

```
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 $\dashv$ 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
```

$$\text{Free} \dashv \text{Forget}$$

Free $\dashv$ Forget $\dashv$ Cofree

For two objects **A** and **B** in a category, can we approximate a notion of *"both **A** and **B**"* ?

...that works universally and identically for any **A** and **B**

For any two categories ***C*** and ***D*** there's a product category ***C***x***D*** with

- Objects: Pairs of objects, one from ***C***, one from ***D***
- Arrows: Pairs of arrows, one from ***C***, one from ***D***

Diagonal functor

$\Delta: \boldsymbol{C} \to \boldsymbol{C} \times \boldsymbol{C}$

$\Delta c = [c, c]$

$\Delta f = [f, f]$

$$\Delta \dashv \Pi$$

$$\Delta a \Rightarrow [b,c] \Leftrightarrow a \to \Pi[b,c]$$

$$[a,a] \Rightarrow [b,c] \Leftrightarrow a \to \Pi[b,c]$$

$$[a,a] \Rightarrow [b,c] \Leftrightarrow a \rightarrow b \times c$$

$$(a \to b,\ a \to c) \Leftrightarrow a \to b{\times}c$$

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

$$(b \times c \to b,\ b \times c \to c)$$

$$
\begin{tikzcd}
& a \arrow[dl, "\delta_1"'] \arrow[d, dotted] \arrow[dr, "\delta_2"] & \\
b & b \times c \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & c
\end{tikzcd}
$$

```
fst :: (b,c) → b
snd :: (b,c) → c
```

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

$$(a \leq b) \wedge (a \leq c) \Leftrightarrow a \leq b \times c$$

$$(a \leq b) \wedge (a \leq c) \Leftrightarrow a \leq b \times c$$

counit: $(b \times c \leq b) \wedge (b \times c \leq c)$

unit: $a \leq a \times a$

$$(a \geq b) \wedge (a \geq c) \Leftrightarrow a \geq b+c$$

counit: $(b+c \geq b) \wedge (b+c \geq c)$

unit: $a \geq a+a$

$$\mathbf{LUB} \dashv \Delta \dashv \mathbf{GLB}$$

$$\mathtt{Either} \dashv \Delta \dashv (,)$$

$$\Sigma \dashv \Delta \dashv \Pi$$

$$\begin{array}{l} F \dashv G \\ P \dashv Q \\ \hdashline FP \dashv GQ \end{array}$$

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?