Adjunctions in everyday life

Or:
What We Talk About When We Talk About Monads

Rúnar Bjarnason
BayHac 2017, **San Francisco**

About Rúnar

- Lead engineer, **Takt**
- Author, *Functional Programming in Scala*

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.”
- Or dually, of finding the most difficult problem that a formula solves.

Goals

- I'm going to show you this pattern over and over
- You're going to start seeing it everywhere
- I want to hear about all the adjunctions that you discover

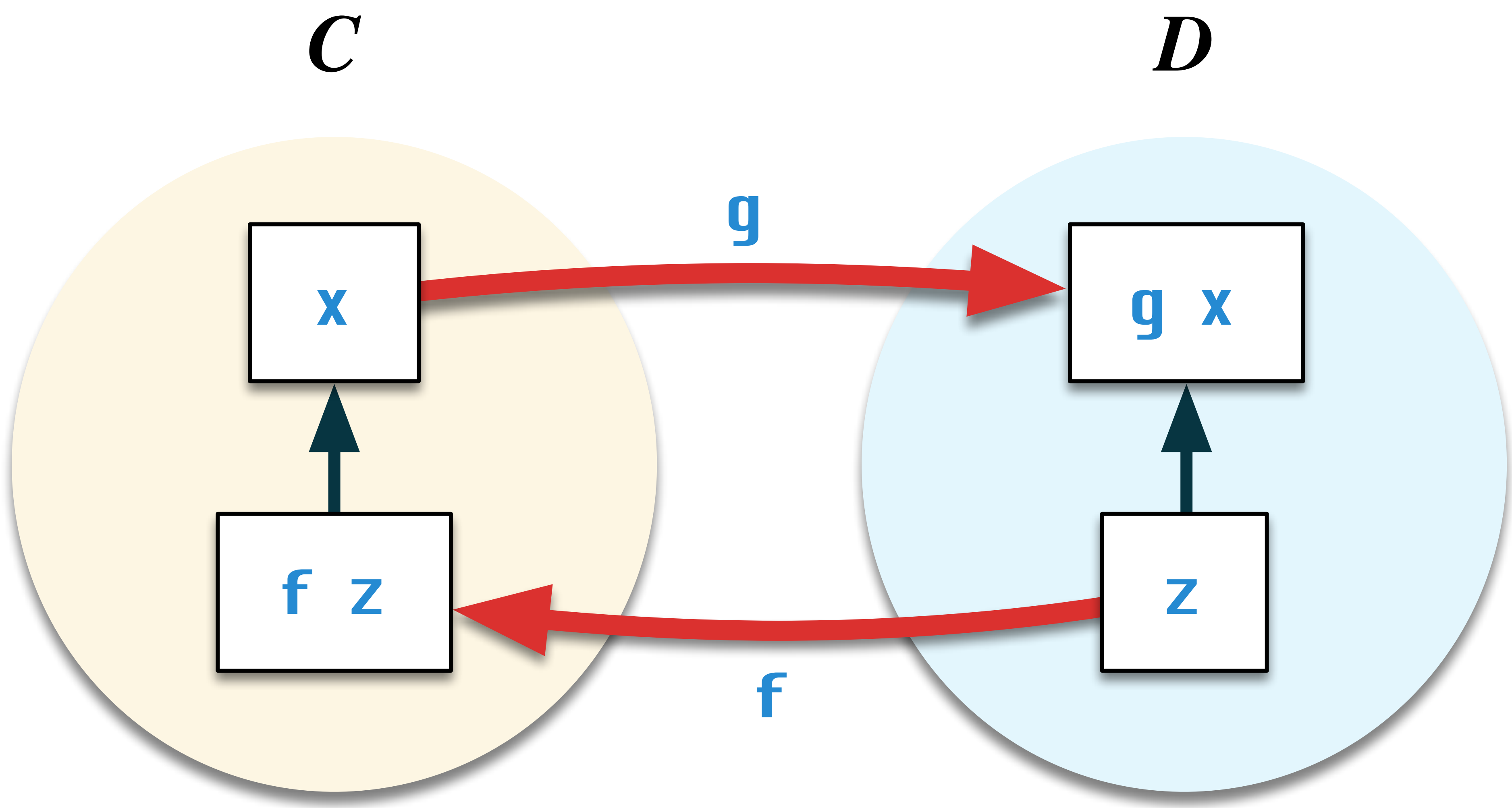
```
curry :: ((a,b) → c) → a → b → c
curry f a b = f (a,b)

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

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

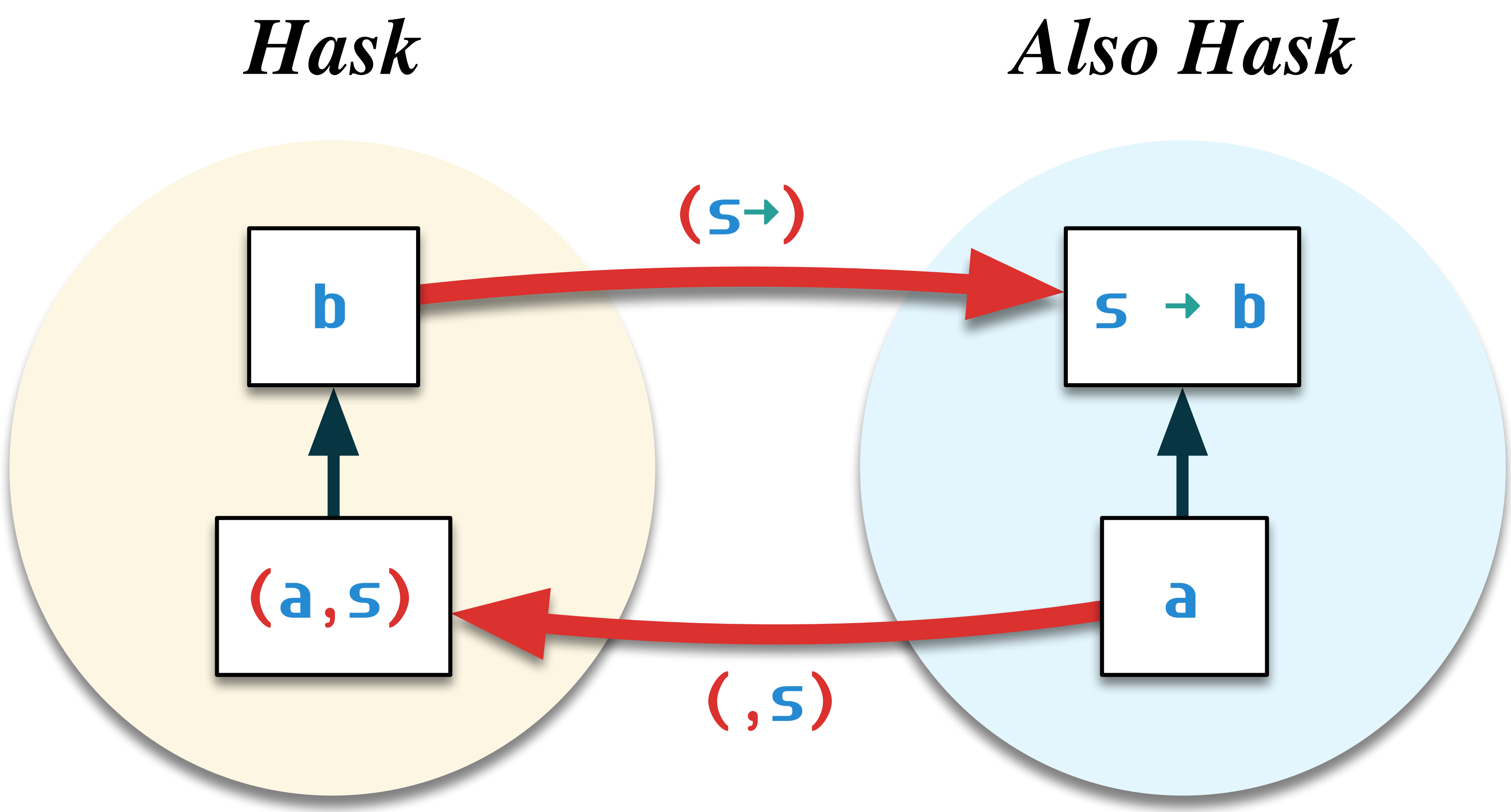
$$(a, b) \to c \quad \Leftrightarrow \quad a \to b \to c$$

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

$$f \dashv g$$



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

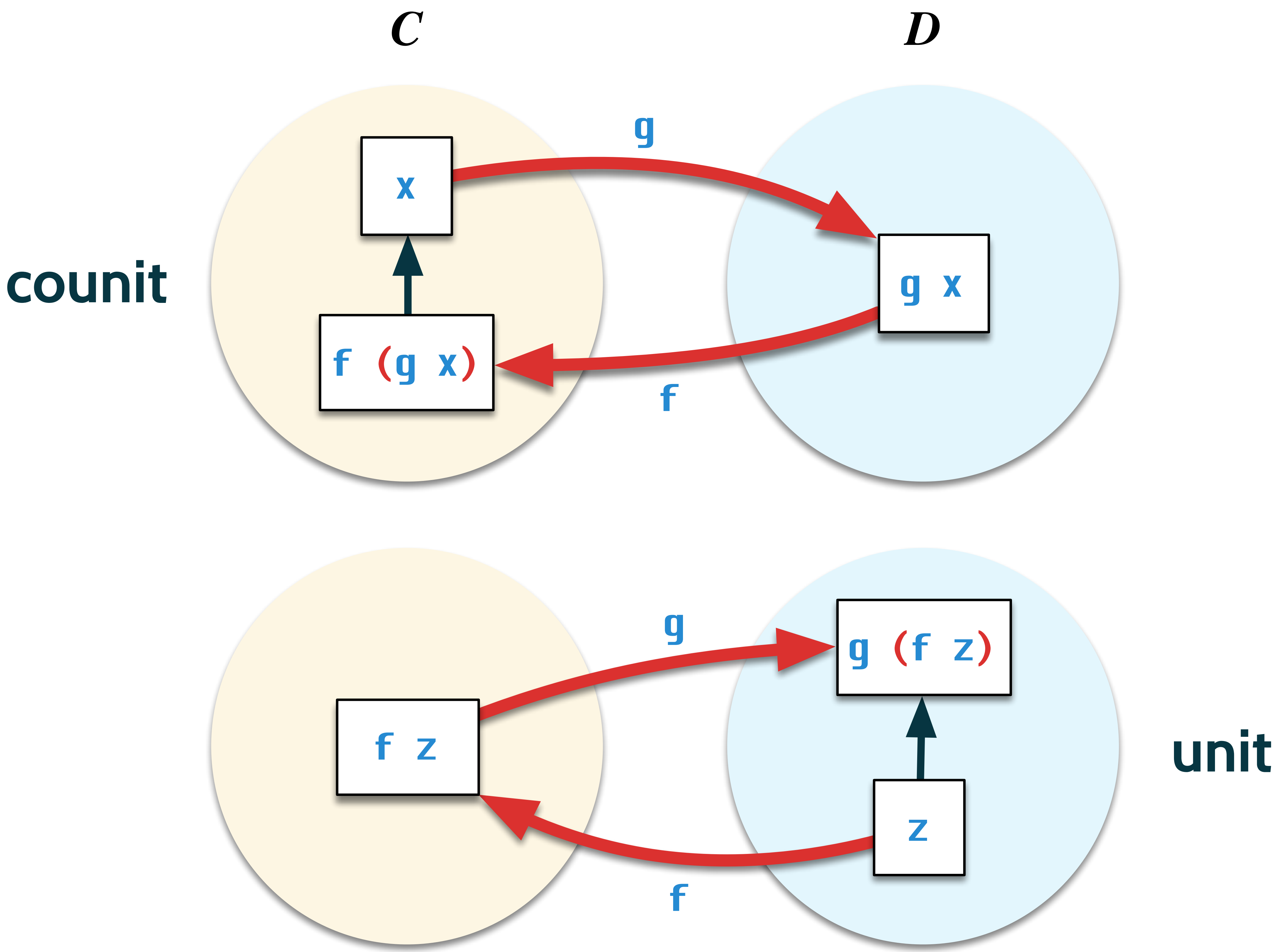


```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a -> b) -> a -> g b
    rightAdjunct :: (a -> g b) -> f a -> b
```

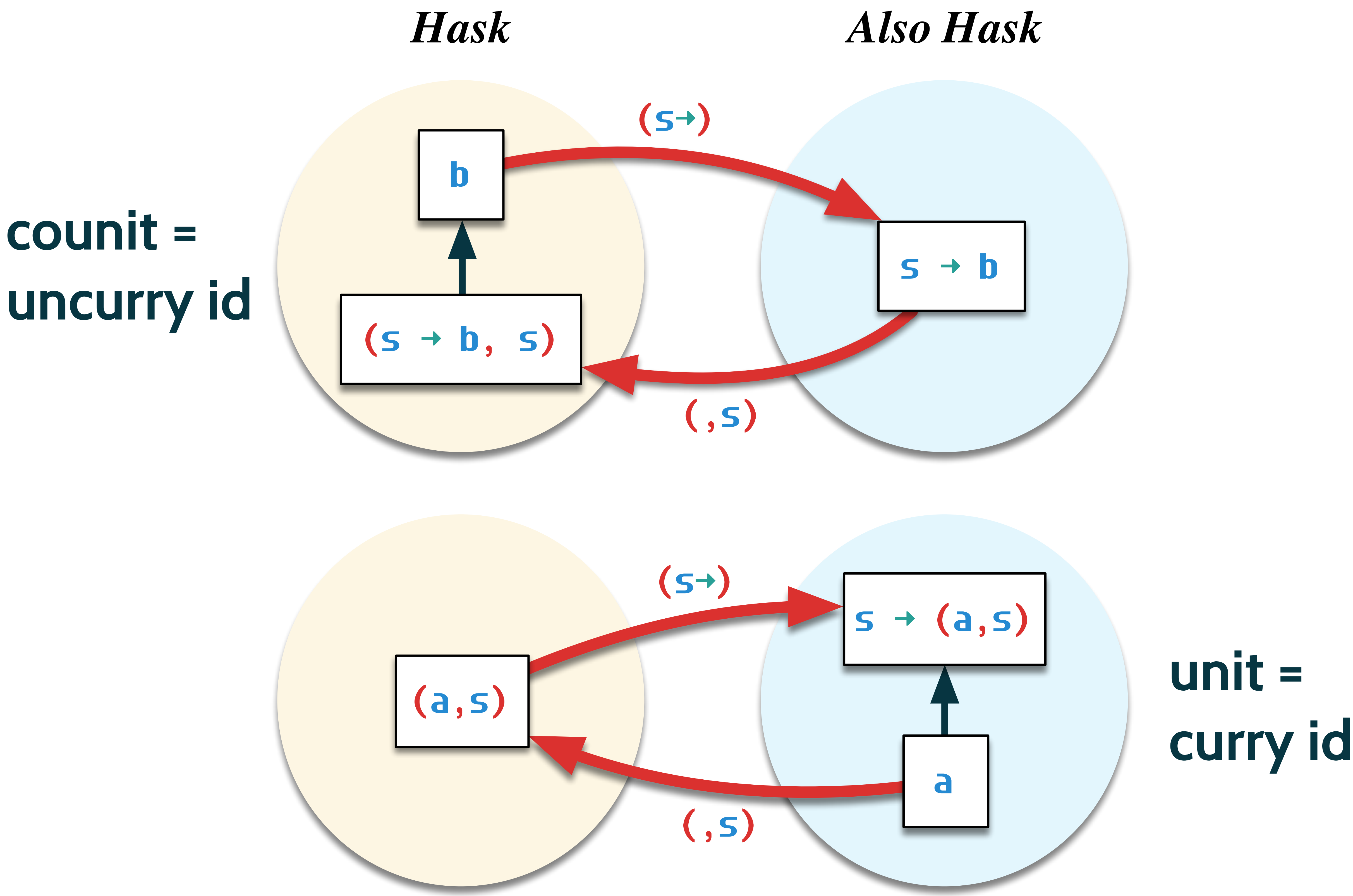
```haskell
instance Adjunction (,s) (s→) where
  leftAdjunct  = curry
  rightAdjunct = uncurry
```

```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)
    counit :: f (g a) -> a

    unit = leftAdjunct id
    counit = rightAdjunct id
```

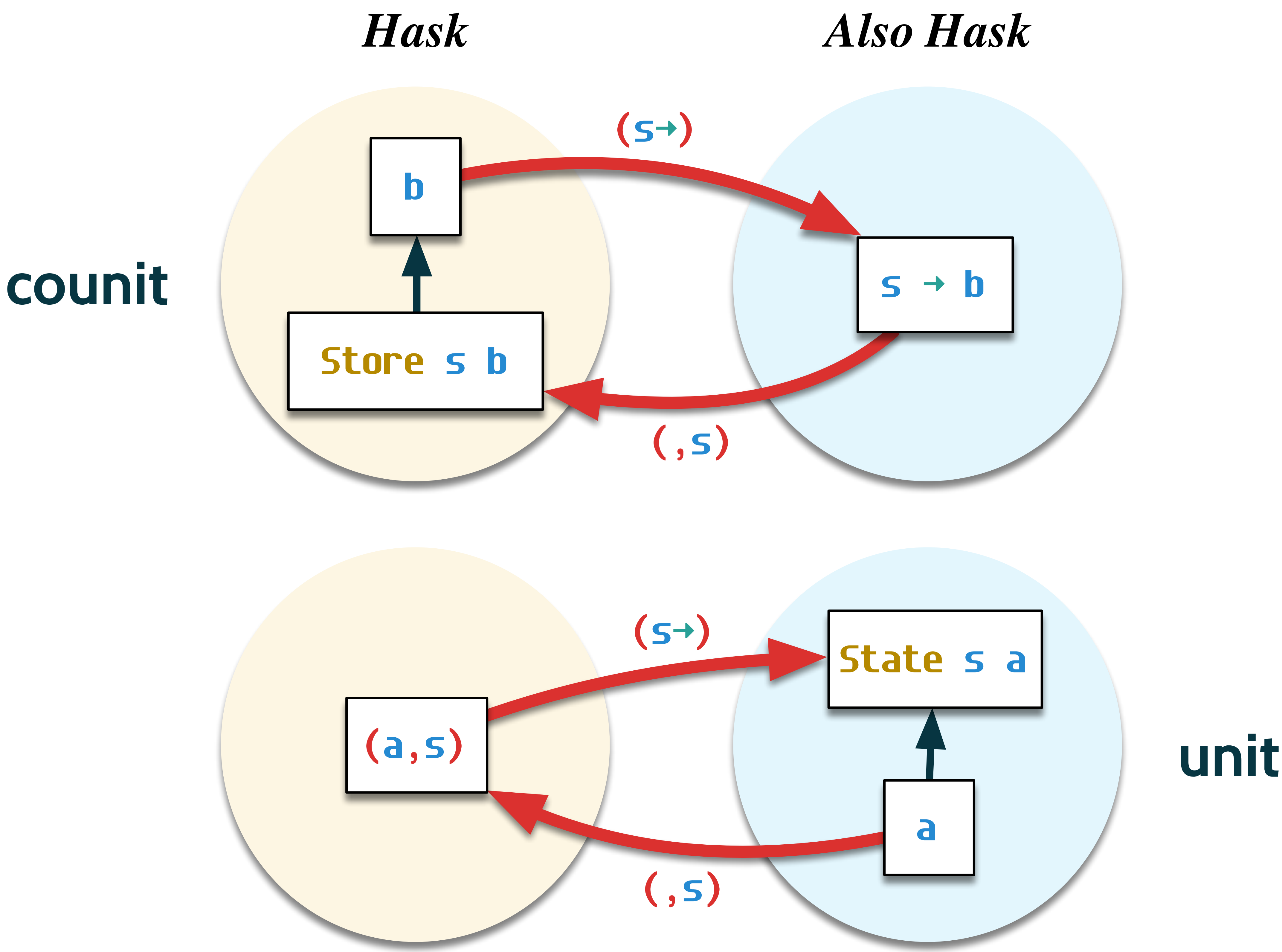
```haskell
class (Functor f, Functor g) =>
  Adjunction f g where
    leftAdjunct  :: (f a -> b) -> a -> g b
    rightAdjunct :: (a -> g b) -> f a -> b

    unit   :: a -> g (f a)
    counit :: f (g a) -> a

    unit = leftAdjunct id
    counit = rightAdjunct id
    leftAdjunct f = fmap f . unit
    rightAdjunct f = counit . fmap f
```



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



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


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

state before

state after

```
type State s a = s → (a,s)
```

output

index

cursor

```
type Store s a = (s → a, s)
```

stored value

State s (State s a)

s → Store s (a,s)

State s a

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

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 m) => Monad m where
  return :: a -> m a
  join   :: m (m a) -> m a
```

```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 f m)
```

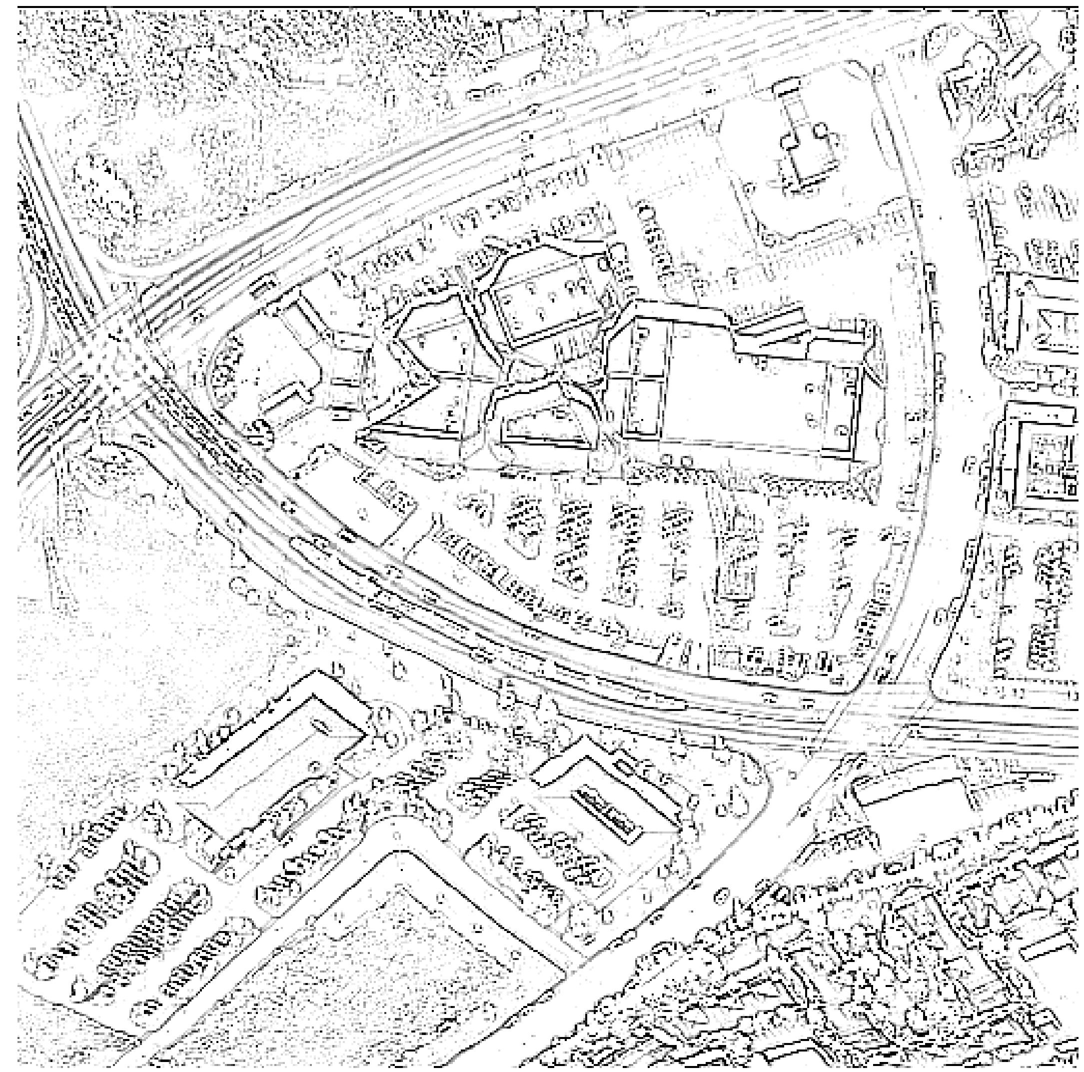
```haskell
class (Functor w) => Comonad w where
  extract   :: w a -> a
  duplicate :: w a -> w (w a)

  (=>>) :: 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)
```

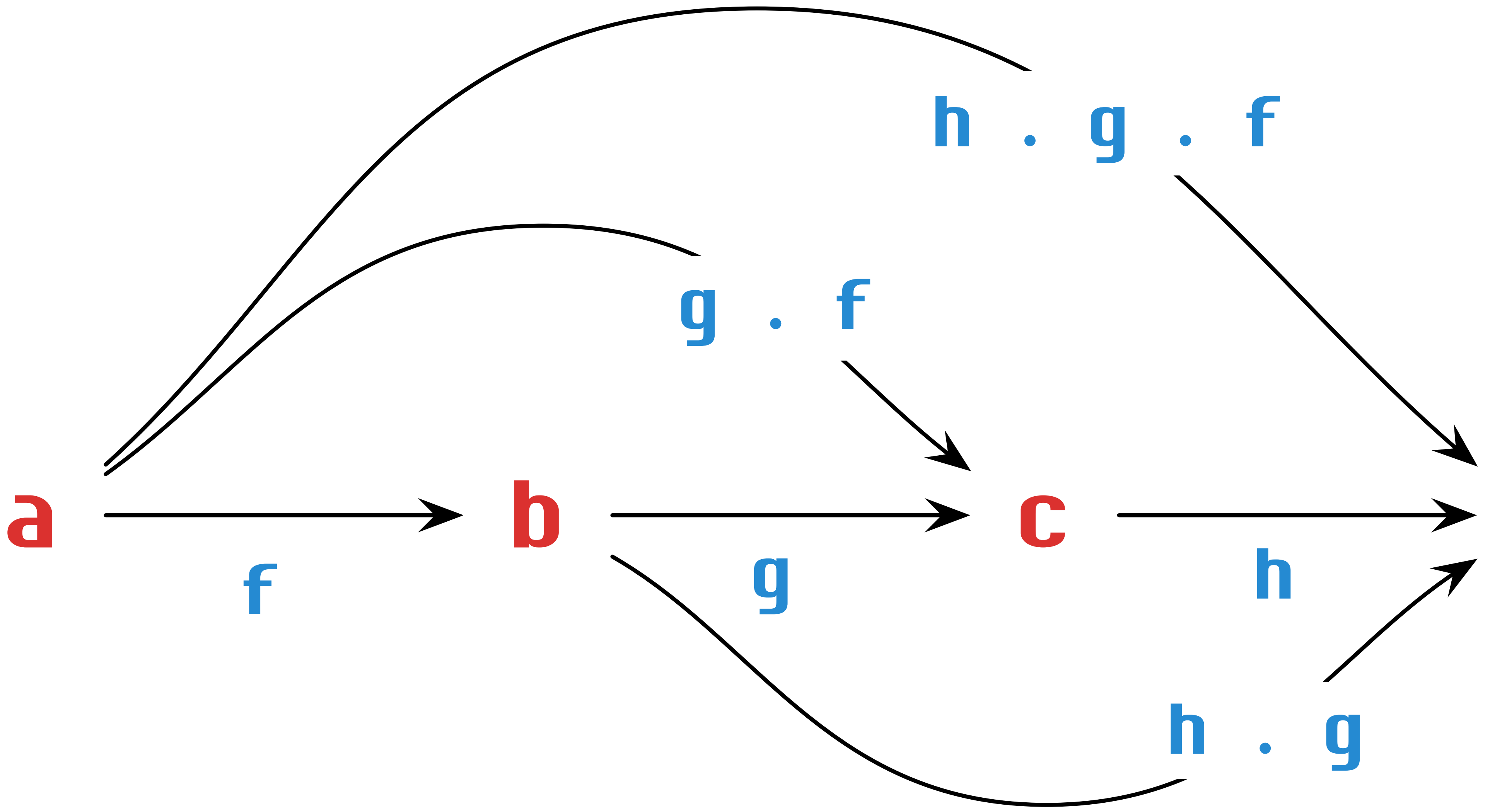


$$(a, b) \to c \quad \Leftrightarrow \quad a \to b \to c$$

Categories

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

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


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

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

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

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

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

$$
\begin{tikzcd}[column sep=large]
a \arrow[r, "a \leq b"'] \arrow[rr, bend left=50, "a \leq c"'] \arrow[rrr, bend left=60, "a \leq d"'] & b \arrow[r, "b \leq c"'] \arrow[rr, bend right=50, "b \leq d"'] & c \arrow[r, "c \leq d"'] & d
\end{tikzcd}
$$

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

(z*y ≤ x) ⇔ (z ≤ x/y)
```

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

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

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

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

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

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

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

Collections:

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

Descriptions:

d1 $\preceq$ **d2** when **d1** is more specific than **d2**

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

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

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

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

```
(-1)

Infinity

(-Infinity)

NaN

null :: forall a. a
```

```haskell
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

$$P \dashv U$$

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

$$P\ \ a \rightarrow b \iff a \rightarrow b$$

```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
```

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


counit :: b → P b → b
unit   :: a → P a
```

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

```haskell
counit b p = foldP p b id
unit a = P $ \_ f → f a
```

```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


radj = maybe
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
```

Free $\dashv$ Forget

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

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

$\Delta c = [c, c]$

$\Delta f = [f, f]$

$$\Delta \dashv \Pi$$

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

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

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

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

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

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

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

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

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

$$(a \le b) \land (a \le c) \Leftrightarrow a \le b \times c$$

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

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: $a \geq a+a$

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

$\texttt{Either} \vdash \Delta \vdash (,)$

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

$$F \dashv G$$

$$P \dashv Q$$

$$\text{- - - - - - -}$$

$$FP \dashv GQ$$

Generic Programming with Adjunctions
Ralf Hinze

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

- Look for adjunctions to generate solutions that naturally fit the problem
- Adjunctions resolve tension between tradeoffs
- Finds an optimal surface between a problem space and solution space

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?

$$\begin{tikzcd} M \arrow[r, "x"] \arrow[dr, "y \diamond x"'] & M \arrow[d, "y"] \\ & M \end{tikzcd}$$

$$\begin{tikzcd} M \arrow[r, "x"] \arrow[dr, "x"'] & M \arrow[d, "1"] \\ & M \end{tikzcd}$$

$$\begin{tikzcd} M \arrow[r, "1"] \arrow[dr, "x"'] & M \arrow[d, "x"] \\ & M \end{tikzcd}$$

$$
\begin{tikzcd}
M \arrow[r, "x"'] \arrow[rr, bend left, "y \diamond x"] \arrow[rrr, bend left=60, "z \diamond y \diamond x"] & M \arrow[r, "y"'] \arrow[rr, bend right, "z \diamond y"'] & M \arrow[r, "z"'] & M
\end{tikzcd}
$$

$$
\begin{tikzcd}
C \arrow[r, "F"] \arrow[dr, "GF"'] & D \arrow[d, "G"] \\
& E
\end{tikzcd}
$$