Adjunctions in everyday life
What we talk about when we talk about monads

Rúnar Bjarnason (@runarorama)
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.
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
  - $\lambda x \rightarrow f (g (h x))$
  - $\lambda x \rightarrow x$
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.
\[
\text{curry} :: ((a,b) \to c) \to a \to b \to c
\]
\[
\text{curry } f \ a \ b = f (a,b)
\]
\[
\text{uncurry} :: (a \to b \to c) \to (a,b) \to c
\]
\[
\text{uncurry } f \ (a,b) = f \ a \ b
\]
\[(a, b) \rightarrow c \iff a \rightarrow b \rightarrow c\]
class (Functor f, Functor g) => Adjunction f g where
    leftAdjunct :: (f a -> b) -> a -> g b
    rightAdjunct :: (a -> g b) -> f a -> b
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:

  leftAdjunct . rightAdjunct = id
  rightAdjunct . leftAdjunct = id
instance Adjunction (,s) (s→) where
  leftAdjunct = curry
  rightAdjunct = uncurry
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
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
\((a, s) \rightarrow (s \rightarrow )\)

**Hask**

- Node: \((a, s)\)
- Node: \(b\)

**Also Hask**

- Node: \(a\)
- Node: \(s \rightarrow b\)

Arrows:
- \((a, s) \rightarrow b\)
- \(s \rightarrow \)
- \((s \rightarrow) \rightarrow (a, s)\)
- \((s \rightarrow) \rightarrow (s, s)\)
counit = uncurry id

unit = curry id
$\text{Store } s \ b$

$counit$

$\text{State } s \ a$

$\text{unit}$
\texttt{type State s a = s \rightarrow (a, s)}
type Store 𝑠 𝑎 = (𝑠 → 𝑎, 𝑠)
type Bitmap2D = Store (Int, Int) Color
join :: (s → (s → (a,s),s)) → s → (a,s)
join = fmap counit

State s (State s a)

s → Store s (a,s)

State s a
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
\[ \text{Store } s \ (\text{Store } s \ a) \]

\[ \text{Store } s \ a \]

\[ (\text{State } s \ (s \rightarrow a), s) \]

\[ \text{duplicate :: } (s \rightarrow a, s) \rightarrow (s \rightarrow (s \rightarrow a, s), s) \]

\[ \text{duplicate } = \text{fmap unit} \]
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
lowPass :: Bitmap2D → Bitmap2D
lowPass bmp = bmp >>> mean
edges :: Bitmap2D → Bitmap2D
edges bmp = bmp >>= \b →
    extract b - extract (lowPass b)
Category of integers
\[ a \leq b \]
\[ a \leq c \]
\[ b \leq c \]
$a \leq a$
$x, y, z :: \text{Integer}$

given $y > 0$

$$(z \cdot y \leq x) \iff (z \leq x/y)$$
\[(z \times y \leq x) \iff (z \leq x/y)\]

**unit:** \[(x/y) \times y \leq x\]

**counit:** \[z \leq (z \times y)/y\]
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:
\[ c_1 \subseteq c_2 \] when \( c_2 \) contains all of \( c_1 \)

Descriptions:
\[ d_1 \preceq d_2 \] when \( d_1 \) is more specific than \( d_2 \)
describe (examples d) \subseteq d

e \subseteq examples (describe e)
describe ← examples

describe e ≤ c ⇔ e ⊆ examples c
a simple API design problem
indexOf :: Eq a ⇒ a → [a] → Integer
\((-1)\)

Infinity

\((-\text{Infinity})\)

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

\[ U: \text{PointedTypes} \rightarrow \text{Types} \]

\[ U[x] \] “forgets” the point of \( x \) and gives the underlying type.
U: \textit{PointedTypes} \rightarrow \textit{Types} has a left adjoint:

\textbf{P}: \textit{Types} \rightarrow \textit{PointedTypes}

for any type \( x \), \( P[x] \) is a pointed type.
$P \rightarrow U$

$P \ a \rightarrow \ b \ \Leftrightarrow \ a \rightarrow U \ b$
\( P \ a \rightarrow b \iff a \rightarrow b \)
rightAdjunct :: Pointed b ⇒ (a → b) → P a → b

leftAdjunct :: (P a → b) → a → b
rightAdjunct :: Pointed b ⇒ (a → b) → P a → b

leftAdjunct :: (P a → b) → a → b

counit :: Pointed b ⇒ P b → b

unit :: a → P a
\text{rightAdjunct} :: b \to (a \to b) \to P a \to b

\text{leftAdjunct} :: (P a \to b) \to a \to b

\text{counit} :: b \to P b \to b

\text{unit} :: a \to P a
rightAdjunct :: (a \rightarrow\ b) \rightarrow P\ a \rightarrow b \rightarrow b

leftAdjunct :: (P\ a \rightarrow b) \rightarrow a \rightarrow b

counit :: P\ b \rightarrow b \rightarrow b

unit :: a \rightarrow P\ a
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
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
join :: Maybe (Maybe a) → Maybe a
join = counit

duplicate :: Maybe a → Maybe (Maybe a)
duplicate = fmap unit
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?
U: Monoids $\to$ Types has a left adjoint:

$M: Types \to Monoids$

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
rightAdjunct :: Monoid b \Rightarrow (a \to b) \to M a \to b

leftAdjunct :: (M a \to b) \to a \to b

counit :: Monoid b \Rightarrow M b \to b

unit :: a \to M a
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: \textit{Monads} $\to$ \textit{Functors} has a \textit{left adjoint}:

Forget: \textit{Functors} $\to$ \textit{Monads}

for any functor $F$, Free[$F$] is a monad
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
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 $C \times D$ with

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

\[ \Delta: C \rightarrow C \times C \]

\[ \Delta c = [c,c] \]

\[ \Delta f = [f,f] \]
Δa ⇒ [b, c] ⇔ a → Π[b, c]
\([a,a] \Rightarrow [b,c] \iff a \rightarrow \Pi[b,c]\)
[a,a] \Rightarrow [b,c] \iff a \rightarrow b \times c
(a → b, a → c) ⇔ a → b\times c
(a → b, a → c) ⇔ a → b\times c

(b\times c → b, b\times c → c)
\texttt{fst} :: (b,\texttt{c}) \rightarrow \texttt{b}

\texttt{snd} :: (b,\texttt{c}) \rightarrow \texttt{c}
(a → b, a → c) \iff a → b \times c
\((a \leq b) \land (a \leq c) \iff a \leq b \times c\)
\[(a \leq b) \land (a \leq c) \iff a \leq b \times c\]

counit: \[(b \times c \leq b) \land (b \times c \leq c)\]

unit: \[a \leq a \times a\]
\((a \geq b) \land (a \geq c) \iff a \geq b+c\)

counit: \((b+c \geq b) \land (b+c \geq c)\)

unit: \(a \geq a+a\)
LUB $\rightarrow$ $\Delta$ $\rightarrow$ GLB
Either ⊢ Δ ⊢ ( , )
Σ ─ Δ ─ Π
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?