Function Applicative for Great Good of Leap Year Function

Polyglot FP for Fun and Profit – Haskell and Scala

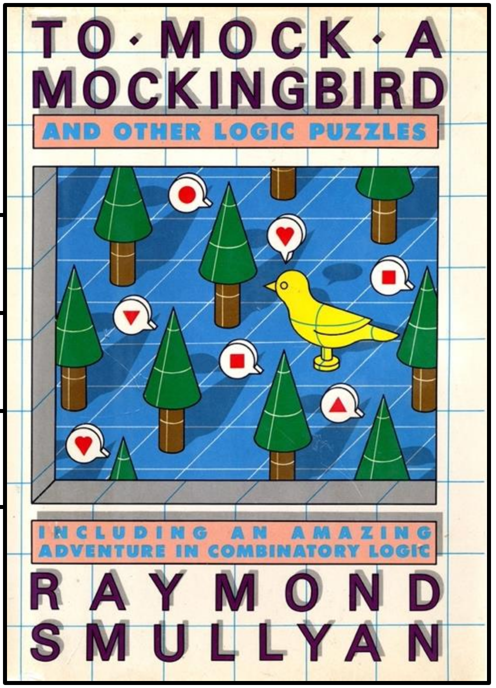
```haskell
leap_year :: Integral a => a -> Bool
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

```haskell
liftA2 = Φ
```

```haskell
liftA2 f g = S (B f g)
```

```haskell
liftA2 f g h = S (S (K f) g) h
```

Φ	*Phoenix*
S	*Starling*
B	*Bluebird*
K	*Kestrel*



slides by  @philip_schwarz FP Illuminated https://fpilluminated.com/

This deck is about the **leap_year** function shown in the tweet below. It is being defined in a **Haskell REPL**.

Given an integer representing a **year**, the function returns a boolean indicating if that **year** is a **leap year.**

@philip_schwarz

https://x.com/Iceland_jack/status/1802659835642528217

The **leap_year** function uses built-in functions **(>)** and **gcd**

```
-- Greater Than function
> :type (>)
(>) :: Ord a => a -> a -> Bool

> (>) 2 3
False

> (>) 3 2
True

-- Greatest Common Divisor function
> :type gcd
gcd :: Integral a => a -> a -> a

> gcd 10 15
5

> gcd 10 16
2

> gcd 10 17
1

> gcd 10 18
2

> gcd 10 19
1

> gcd 10 20
10
```

```
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

leap_year also uses liftA2, which is a function provided by Applicative.

```
> import Control.Applicative

> :type liftA2
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
```

Let’s define leap_year and take it for a quick spin.

```haskell
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

```
> leap_year = liftA2 (>) (gcd 80) (gcd 50)

> :type leap_year
leap_year :: Integral a => a -> Bool

> leap_year 2024
True

> leap_year 2025
False

> fmap leap_year [1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400]
[True,False,False,False,True,False,False,False,True]
```

The following slide uses logic operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\Leftrightarrow$.

Here is a reminder of their definition.

Operator	**Definition**
$\neg P$	$not\ P$
$P \vee Q$	$P\ or\ Q$
$P \wedge Q$	$P\ and\ Q$
$P \rightarrow Q$	$if\ P\ then\ Q$
$P \Leftrightarrow Q$	$P\ if\ and\ only\ if\ Q$

It looks like **leap_year** is exploiting the algorithm described in the following twitter/x thread.

https://x.com/chordbug/status/1497912619784720384

```
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

Post

Lynn (so is she gone or not??)
@chordbug

I don't know who needs to hear this but year "n" is a leap year iff

gcd(80, n) > gcd(50, n)

11:02 PM · Mar 22, 2021

https://x.com/chordbug/status/1497912619784720384/photo/1

The traditional definition of a leap year is something like:

$$n \text{ is a leap year} \iff 4 \mid n \wedge \neg(100 \mid n \wedge \neg(400 \mid n))$$

But we'll work with a slightly simpler logically equivalent form:

$$n \text{ is a leap year} \iff 4 \mid n \wedge (100 \mid n \rightarrow 400 \mid n)$$

The magic formula states that

$$n \text{ is a leap year} \iff \boxed{\gcd(n, 80) > \gcd(n, 50)}$$

and we'd like to prove these are equivalent:

$$\boxed{4 \mid n \wedge (100 \mid n \rightarrow 400 \mid n) \iff \gcd(n, 80) > \gcd(n, 50)}$$

By the way, in case you are asking yourself how the previous slide refactored this

$$n\ is\ a\ leap\ year \iff 4 \mid n \land \boxed{\neg(100 \mid n \land \neg(400 \mid n))}$$

to this

$$n\ is\ a\ leap\ year \iff 4 \mid n \land (100 \mid n \rightarrow 400 \mid n)$$

see below for how I explained it to myself.

If we apply **De Morgan's law**, i.e.

refactor

$$\neg(P \land Q) = \neg P \lor \neg Q$$

to

$$\neg(100 \mid n \land \neg(400 \mid n))$$

we get

$$n\ is\ a\ leap\ year \iff 4 \mid n \land \boxed{(\neg(100 \mid n) \lor 400 \mid n)}$$

Next, if we apply $\neg P \lor Q = P \rightarrow Q$

refactor

to

$$\neg(100 \mid n) \lor (400 \mid n)$$

we get

$$n\ is\ a\ leap\ year \iff 4 \mid n \land \boxed{(100 \mid n \rightarrow 400 \mid n)}$$

which reads as follows:

n is a leap year if and only if both of the following are true

- it is divisible by 4
- if it is divisible by 100, then it is also divisible by 400

Why is it that, given function definition

```
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

and given some input **year**

e.g. **2024**

evaluating `leap_year year`, amounts to evaluating

```
(gcd 80 year) > (gcd 50 year)
```

e.g.

```
(gcd 80 2024) > (gcd 50 2024)
```

?

Here is the definition of **Applicative** function **liftA2**

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Source

Lift a binary function to actions.

Some functors support an implementation of `liftA2` that is more efficient than the default one. In particular, if `fmap` is an expensive operation, it is likely better to use `liftA2` than to `fmap` over the structure and then use `<*>`.

This became a typeclass method in 4.10.0.0. Prior to that, it was a function defined in terms of `<*>` and `fmap`.

https://hackage.haskell.org/package/base-4.20.0.1/docs/Prelude.html#liftA2

But given that the signature of `liftA2` is

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

how does

```
liftA2 (>) (gcd 80) (gcd 50) 2024
```

map to

```
(gcd 80 2024) > (gcd 50 2024)
```

?

```
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

The first step that we are going to take to answer this question, is to consider the actual parameters of `liftA2` in

```
liftA2 (>) (gcd 80) (gcd 50) 2024
```

The first one, i.e. `(>)`, is a function with type `Int -> Int -> Bool`.

The second one, i.e. `(gcd 80)` is the result of applying a function of type `Int -> Int -> Int` to `80`, which results in a function `Int -> Int`.

The third one, i.e. `(gcd 50)` is the result of applying a function of type `Int -> Int -> Int` to `50`, which also results in a function `Int -> Int`.

Given that the type of `leap_year` is `Int -> Bool`, it follows that in

```
liftA2 (>) (gcd 80) (gcd 50) 2024
```

the signature of `liftA2` is

```
liftA2 :: (Int -> Int -> Bool) -> (Int -> Int) -> (Int -> Int) -> (Int -> Bool)
```

But what abstraction does f need to be in order for

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

to become

```
liftA2 :: (Int -> Int -> Boolean) -> (Int -> Int) -> (Int -> Int) -> (Int -> Boolean)
```

?

Let us define f to be a function of type `r -> ?`, where r is some specific type, and ? is some yet to be specified type.

Now let's update the signature of `liftA2` to reflect the above definition of f:

```
liftA2 :: (a -> b -> c) -> (r -> ?1) -> (r -> ?2) -> (r -> ?3)
```

In the case at hand, i.e.

```
liftA2 (>) (gcd 80) (gcd 50) 2024
```

```
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

we already know that

1. `(a -> b -> c)` is `(Int -> Int -> Bool)`, i.e. the type of `(>)`,
2. `(r -> ?3) is (Int -> Boolean)`, i.e. the type of `leap_year`
3. `(r -> ?1)` and `(r -> ?2)` are `(Int -> Int)`, i.e. the type of both `(gcd 80)` and `(gcd 50)`

so we see that with `r = Int`, `?1 = Int`, `?2 = Int` and `?3 = Bool`, the signature of `liftA2` is indeed the sought one:

```
liftA2 :: (Int -> Int -> Bool) -> (Int -> Int) -> (Int -> Int) -> (Int -> Bool)
```

So, to arrive at the **liftA2** signature that is applicable in

```
liftA2 (>) (gcd 80) (gcd 50) 2024
```

i.e.

```
liftA2 :: (Int -> Int -> Bool) -> (Int -> Int) -> (Int -> Int) -> (Int -> Bool)
```

we first take the minimal definition of **Functor**

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

and a minimal definition of `Applicative`, but with **liftA2** added to it

```
class Functor f => Applicative f where
    pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

We then take the **Function** `Functor` and **Function** `Applicative`, i.e. the `Functor` and `Applicative` instances for `((->) r)`, in which `f` is defined to be a function from some specific type `r` to some yet unspecified type. Here are the function signatures of the resulting instances:

```
  fmap :: (a -> b) -> (r -> a) -> (r -> b)
  pure :: a -> (r -> a)
 (<*>) :: (r -> a -> b) -> (r -> a) -> (r -> b)
liftA2 :: (a -> b -> c) -> (r -> a) -> (r -> b) -> (r -> c)
```

If we define `r = Int`, `a = Int`, `b = Int` and `c = Bool`, then `liftA2` takes on the desired signature:

```
liftA2 :: (Int -> Int -> Bool) -> (Int -> Int) -> (Int -> Int) -> (Int -> Bool)
```

Now let's get back to the question that we are looking to answer.

Why is it that given function definition

```
leap_year = liftA2 (>) (gcd 80) (gcd 50)
```

and given some input **year**, evaluating `leap_year year`, amounts to evaluating

```
(gcd 80 year) > (gcd 50 year)
```

?

Or in other words, since `leap_year` is defined in terms of **liftA2**, why is it that

```
liftA2 (>) (gcd 80) (gcd 50) year
```

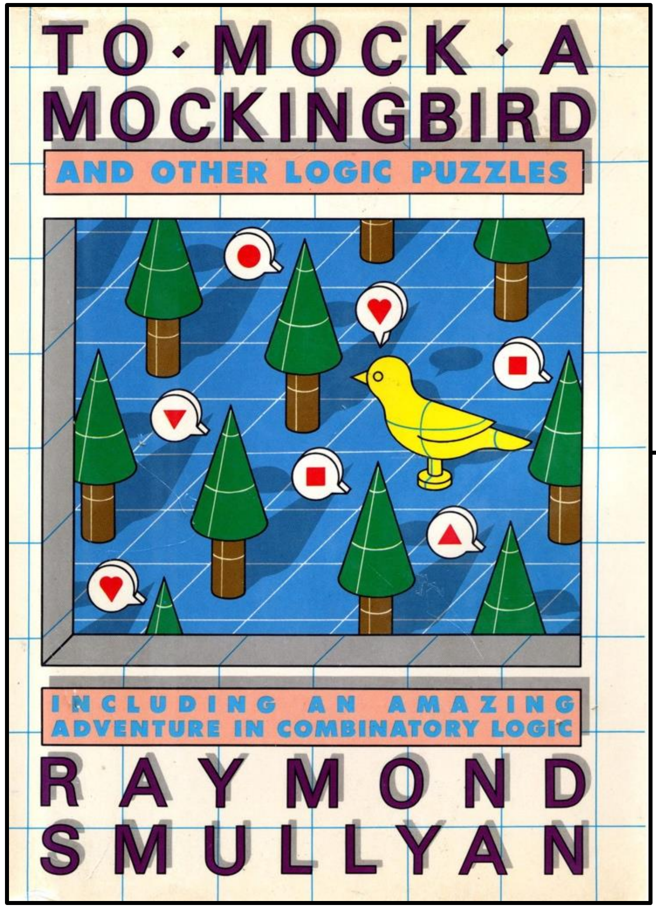
evaluates to

```
(gcd 80 year) > (gcd 50 year)
```

?

It turns out that the `liftA2` function of the **Function** `Applicative` is a **combinatory logic** function (a **combinator**) called the **phoenix**.

To answer the question restated in the previous slide, instead of looking at the code for `liftA2`, in the next slide we are going to exploit the fact that `liftA2` = **phoenix**.




```
liftA2 :: (a -> b -> c) -> (r -> a) -> (r -> b) -> (r -> c)
```

Phoenix

$$\Phi\, x\, y\, z\, w = x(y\, w)(z\, w)$$

rename variables for additional clarity

$$\Phi\, f\;\, g\; h\; x = f(g\; x)(h\; x)$$

make 'point free'

$$\Phi = \lambda f.\lambda g.\lambda h.\lambda x.f(g\; x)(h\; x)$$

```
phoenix :: (b -> c -> d) -> (a -> b) -> (a -> c) -> a -> d
```

(Big) Phi combinator - phoenix - Haskell `liftM2`.

This is the same function as `starling'`.

https://hackage.haskell.org/package/data-aviary-0.4.0/docs/Data-Aviary-Birds.html

```
starling' :: (b -> c -> d) -> (a -> b) -> (a -> c) -> a -> d
```

S' combinator - starling prime - Turner's big phi. Haskell: Applicative's `liftA2` on functions (and similarly Monad's `liftM2`).

This is the same function as `phoenix`.

$$\boxed{\Phi = \lambda f.\lambda g.\lambda h.\lambda x.f(g\ x)(h\ x)}$$

Equation	**Action**
$leap\_year = liftA2\ (>)\ (gcd\ 80)\ (gcd\ 50)$	$liftA2 = \Phi$
$leap\_year = \Phi\ (>)\ (gcd\ 80)\ (gcd\ 50)$	$\Phi = \lambda f.\lambda g.\lambda h.\lambda x.f(g\ x)(h\ x)$
$leap\_year = \big(\lambda f.\lambda g.\lambda h.\lambda x.f(g\ x)(h\ x)\big)(>)\ (gcd\ 80)\ (gcd\ 50)$	$f = (>)$
$leap\_year = (\lambda g.\lambda h.\lambda x.(>)(g\ x)(h\ x))\ (gcd\ 80)\ (gcd\ 50)$	$g = gcd\ 80$
$leap\_year = (\lambda h.\lambda x.(>)(gcd\ 80\ x)(h\ x))\ (gcd\ 50)$	$h = gcd\ 50$
$leap\_year = \lambda x.(>)(gcd\ 80\ x)(gcd\ 50\ x)$	$(>)\ x\ y = x > y$
$leap\_year = \lambda x.(gcd\ 80\ x) > (gcd\ 50\ x)$	*apply leap_year to* 2024
$leap\_year\ 2024 = (\lambda x.(gcd\ 80\ x) > (gcd\ 50\ x))\quad 2024$	$x = 2024$
$leap\_year\ 2024 = (gcd\ 80\ 2024) > (gcd\ 50\ 2024)$	Q.E.D.

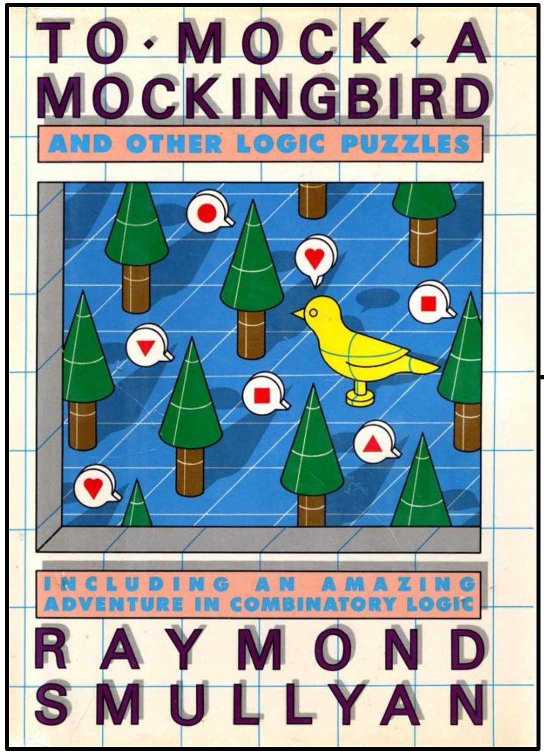
Thanks to the **phoenix**, we have not needed to look at the implementation of `liftA2` in order to understand how it works. Still, what does the implementation look like? It uses `<*>` and `fmap`:

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2 f x = (<*>) (fmap f x)
```

It turns out that in the **Function Functor**, **fmap** is the **Bluebird combinator**, and in the **Function Applicative**, **<*>** is the **Starling combinator**. So again, instead of looking at the code for **fmap** and **<*>**, in the next slide we are going to exploit the fact that **fmap** = **bluebird** and **<*>** = **starling**.

```
liftA2 f x = (<*>)(fmap f x)
```

```
liftA2 f g = S(B f g)
```




```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Bluebird — $B\ x\ y\ z = x(y\ z)$ → $B\ f\ g\ x = f(g\ x)$ → $B = \lambda f.\lambda g.x.f(g\ x)$

```
bluebird :: (b -> c) -> (a -> b) -> a -> c
```

B combinator - bluebird - Haskell (.).

The function composition function. Given two functions f and g, it returns a function h that is f composed with g, i.e. $h(x) = f(g(x))$. Also known as **Compositor**.

https://hackage.haskell.org/package/data-aviary-0.4.0/docs/Data-Aviary-Birds.html

```
starling :: (a -> b -> c) -> (a -> b) -> a -> c
```

S combinator - starling.

Haskell: Applicative's (<*>) on functions.

Substitution.

Also known as **Distributor**.

Starling — $S\ x\ y\ z = (x\ z)(y\ z)$ → $S\ f\ g\ x = (f\ x)(g\ x)$ → $S = \lambda f.\lambda g.\lambda x.(f\ x)(g\ x)$

```
(<*>) :: (r -> a -> b) -> (r -> a) -> (r -> b)
```

$B = \lambda f.\lambda g.x.f(g\ x)$ $S = \lambda f.\lambda g.\lambda x.(f\ x)(g\ x)$ `liftA2 f g = S(B f g)`

Equation	Action
$leap\_year = liftA2\ (>)\ (gcd\ 80)\ (gcd\ 50)$	$liftA2\ f\ g = S(B\ f\ g)$
$leap\_year = S(B\ (>)\ (gcd\ 80))\ (gcd\ 50)$	$S = \lambda f.\lambda g.\lambda x.(f\ x)(g\ x)$
$leap\_year = (\lambda f.\lambda g.\lambda x.(f\ x)(g\ x))(B\ (>)\ (gcd\ 80))\ (gcd\ 50)$	$f = B\ (>)\ (gcd\ 80)$
$leap\_year = (\lambda g.\lambda x.(B\ (>)\ (gcd\ 80)\ x)(g\ x))\ (gcd\ 50)$	$g = gcd\ 50$
$leap\_year = (\lambda x.(B\ (>)\ (gcd\ 80)\ x)(gcd\ 50\ x))$	$B = \lambda f.\lambda g.\lambda x.f(g\ x) = \lambda f.\lambda g.\lambda y.f(g\ y)$
$leap\_year = (\lambda x.((\lambda f.\lambda g.\lambda y.f(g\ y))\ (>)\ (gcd\ 80)\ x)(gcd\ 50\ x))$	$f = (>)$
$leap\_year = (\lambda x.((\lambda g.\lambda y.(>)(g\ y))\ (gcd\ 80)\ x)(gcd\ 50\ x))$	$g = gcd\ 80$
$leap\_year = (\lambda x.((\lambda y.(>)(gcd\ 80\ y))\ x)(gcd\ 50\ x))$	$y = x$
$leap\_year = (\lambda x.(>)(gcd\ 80\ x)(gcd\ 50\ x))$	$(>)\ x\ y = x > y$
$leap\_year = (\lambda x.(gcd\ 80\ x) > (gcd\ 50\ x))$	$apply\ \ leap\_year\ \ to\ 2024$
$leap\_year\ 2024 = \big(\lambda x.(gcd\ 80\ x) > (gcd\ 50\ x)\big)\ 2024$	$x = 2024$
$leap\_year\ 2024 = (gcd\ 80\ 2024) > (gcd\ 50\ 2024)$	Q.E.D.

In previous slides, we saw this definition of **liftA2**

```haskell
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2 f x = (<*>) (fmap f x)
```

Here is the same definition, but using the **infix operator** equivalent of function **fmap**, and the **infix operator** equivalent of function (**<*>**).

```haskell
liftA2 f g h = f <$> g <*> h
```

```haskell
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

The above is a more convenient version of the following:

```haskell
liftA2 f g h = pure f <*> g <*> h
```

Applicative functors

Synopsis

```haskell
class Functor f => Applicative (f :: Type -> Type) where
```

Source

A functor with application, providing operations to

- embed pure expressions (`pure`), and
- sequence computations and combine their results (`<*>` and `liftA2`).

A minimal complete definition must include implementations of `pure` and of either `<*>` or `liftA2`. If it defines both, then they must behave the same as their default definitions:

```haskell
(<*>) = liftA2 id
```

```haskell
liftA2 f x y = f <$> x <*> y
```

https://hackage.haskell.org/package/base-4.20.0.1/docs/Control-Applicative.html

On the previous slide we saw the following possible implementation of **liftA2**

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2 f g h = pure f <*> g <*> h
```

In the **Function Applicative**, **<*>** is the **Starling combinator** that we saw earlier, and **pure** is the **Kestrel combinator**. So again, instead of looking at the code for **pure** and **<*>**, in the next slide we are going to exploit the fact that **pure** = **kestrel** and **<*>** = **starling**.

```
liftA2 f x = (<*>) (fmap f x)
```

```
liftA2 f g h = f <$> g <*> h
```

```
liftA2 f g h = pure f <*> g <*> h
```

```
liftA2 f g h = S (S (K f) g) h
```

```
pure :: a -> (r -> a)
```

Kestrel | $K x\, y = x$ | $K\, f\, g = f$ | $K = \lambda f.\lambda g.f$

```
kestrel :: a -> b -> a
```

K combinator - kestrel - Haskell `const`. Corresponds to the encoding of `true` in the lambda calculus.

Also known as **Cancellator**.

https://hackage.haskell.org/package/data-aviary-0.4.0/docs/Data-Aviary-Birds.html

```
starling :: (a -> b -> c) -> (a -> b) -> a -> c
```

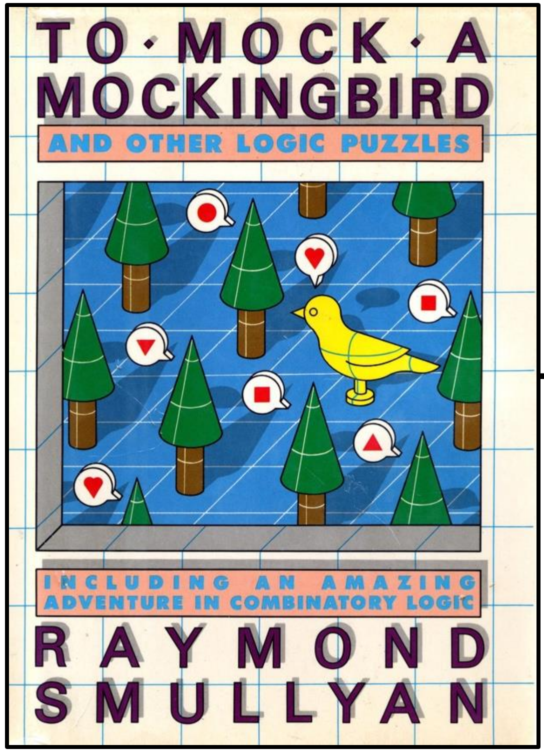
S combinator - starling.

Haskell: Applicative's `(<*>)` on functions.

Substitution.

Starling | $S\, x\, y\, z = (x\, z)(y\, z)$ | $S\, f\, g\, x = (f\, x)(g\, x)$ | $S = \lambda f.\lambda g.\lambda x.(f\, x)(g\, x)$

```
(<*>) :: (r -> a -> b) -> (r -> a) -> (r -> b)
```



$K = \lambda f.\lambda g.f$

$S = \lambda f.\lambda g.\lambda x.(f\ x)(g\ x)$

```
liftA2 f g h = S (S (K f) g) h
```

Equation	Action
$leap\_year = liftA2\ (>)\ (gcd\ 80)\ (gcd\ 50)$	$liftA2\ f\ g\ h = S\ (S\ (K\ f)\ g)\ h$
$leap\_year = S\ \left(S\ (K\ (>))(gcd\ 80)\right)(gcd\ 50)$	$S = \lambda f.\lambda g.\lambda x.(f\ x)(g\ x)$
$leap\_year = (\lambda f.\lambda g.\lambda x.(f\ x)(g\ x))\left(S\ (K\ (>))(gcd\ 80)\right)(gcd\ 50)$	$f = \left(S\ (K\ (>))(gcd\ 80)\right)$
$leap\_year = (\lambda g.\lambda x.\left(S\ (K\ (>))(gcd\ 80)\ x\right)(g\ x))\ (gcd\ 50)$	$g = gcd\ 50$
$leap\_year = \lambda x.\left(S\ (K\ (>))(gcd\ 80)\ x\right)(gcd\ 50\ x)$	$S = \lambda f.\lambda g.\lambda x.(f\ x)(g\ x) = \lambda f.\lambda g.\lambda y.(f\ y)(g\ y)$
$leap\_year = \lambda x.\left((\lambda f.\lambda g.\lambda y.(f\ y)(g\ y))(K\ (>))(gcd\ 80)\ x\right)(gcd\ 50\ x)$	$f = (K\ (>))$
$leap\_year = \lambda x.((\lambda g.\lambda y.(K\ (>)\ y)(g\ y))\ (gcd\ 80)\ x)(gcd\ 50\ x)$	$g = gcd\ 80$
$leap\_year = \lambda x.((\lambda y.(K\ (>)\ y)(gcd\ 80\ y))\ x)(gcd\ 50\ x)$	$y = x$
$leap\_year = \lambda x.(K\ (>)\ x)(gcd\ 80\ x)(gcd\ 50\ x)$	$\mathrm{K} = \lambda f.\lambda g.f$
$leap\_year = \lambda x.((\lambda g.(>))\ x)(gcd\ 80\ x)(gcd\ 50\ x)$	$f = (>)$
$leap\_year = \lambda x.((\lambda g.(>))\ x)(gcd\ 80\ x)(gcd\ 50\ x)$	$g = x$
$leap\_year = \lambda x.(>)(gcd\ 80\ x)(gcd\ 50\ x)$	$(>)\ x\ y = x > y$
$leap\_year = \lambda x.(gcd\ 80\ x) > (gcd\ 50\ x)$	*apply leap_year to* 2024
$leap\_year\ 2024 = \left(\lambda x.(gcd\ 80\ x)(> gcd\ 50\ x)\right)\ 2024$	$x = 2024$
$leap\_year\ 2024 = (gcd\ 80\ 2024) > (gcd\ 50\ 2024)$	Q.E.D.

We had a go at understanding the following functions of the **Function Applicative**, without the need to look at their code: **fmap**, **pure**, **<*>** and **liftA2**.

We did this by looking at their equivalent **combinators**: **Bluebird**, **Kestrel**, **Starling** and **Phoenix**.

While we have now seen the code for **liftA2**, we have not yet seen that for **fmap**, **pure** and **<*>**.

Now the we are familiar with the **combinators**, the code for **fmap**, **pure** and **<*>** does not present any surprises, and can be seen on the following slide, which acts as a recap of the correspondence between the functions and the **combinators**.

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```haskell
class Functor f => Applicative f where
   pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
 liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Bluebird	B	$B\ f\ g\ x = f\,(g\ x)$	fmap (<$>)
Kestrel	K	$K\ x\ y = x$	pure
Starling	S	$S\ f\ g\ x = (f\ x)(g\ x)$	(<*>)
Phoenix	Φ	$\Phi\ f\ g\ h\ x = f\,(g\ x)(h\ x)$	liftA2

```haskell
instance Functor ((->) r) where
  fmap = (.)
```

```haskell
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

```haskell
instance Applicative ((->) r) where
  pure x = (\_ -> x)
  f <*> g = \x -> f x (g x)
  liftA2 f x = (<*>) (fmap f x)
```

The next slide shows the **Scala** code for the definition of **leap_year** in terms of the following alternative equivalent implementations of **liftA2**:

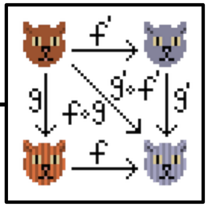
```
liftA2 f x = (<*>) (fmap f x)

liftA2 f g h = f <$> g <*> h

liftA2 f g h = pure f <*> g <*> h
```

```scala
import scala.math.BigInt.int2bigInt
```

```scala
import cats.*
import cats.implicits.*
```



```scala
val gcd: Int => Int => Int =
  x => y => x.gcd(y).intValue

val `(>)`: Int => Int => Boolean =
  x => y => x > y
```

```scala
extension [A,B](f: A => B)
  def `<$>`[F[_]: Functor](fa: F[A]): F[B] = fa.map(f)
```

```scala
def liftA2_v1[A,B,C,F[_]: Applicative](f: A => B => C)(fa: F[A], fb: F[B]): F[C] =
  fa.map(f) <*> fb
```

```scala
def liftA2_v2[A,B,C,F[_]: Applicative](f: A => B => C)(fa: F[A], fb: F[B]): F[C] =
  f `<$>` fa <*> fb
```

```scala
def liftA2_v3[A,B,C,F[_]: Applicative](f: A => B => C)(fa: F[A], fb: F[B]): F[C] =
  f.pure <*> fa <*> fb
```

```scala
val leapYear1: Int => Boolean =
  liftA2_v1(`(>)`)(gcd(80), gcd(50))
```

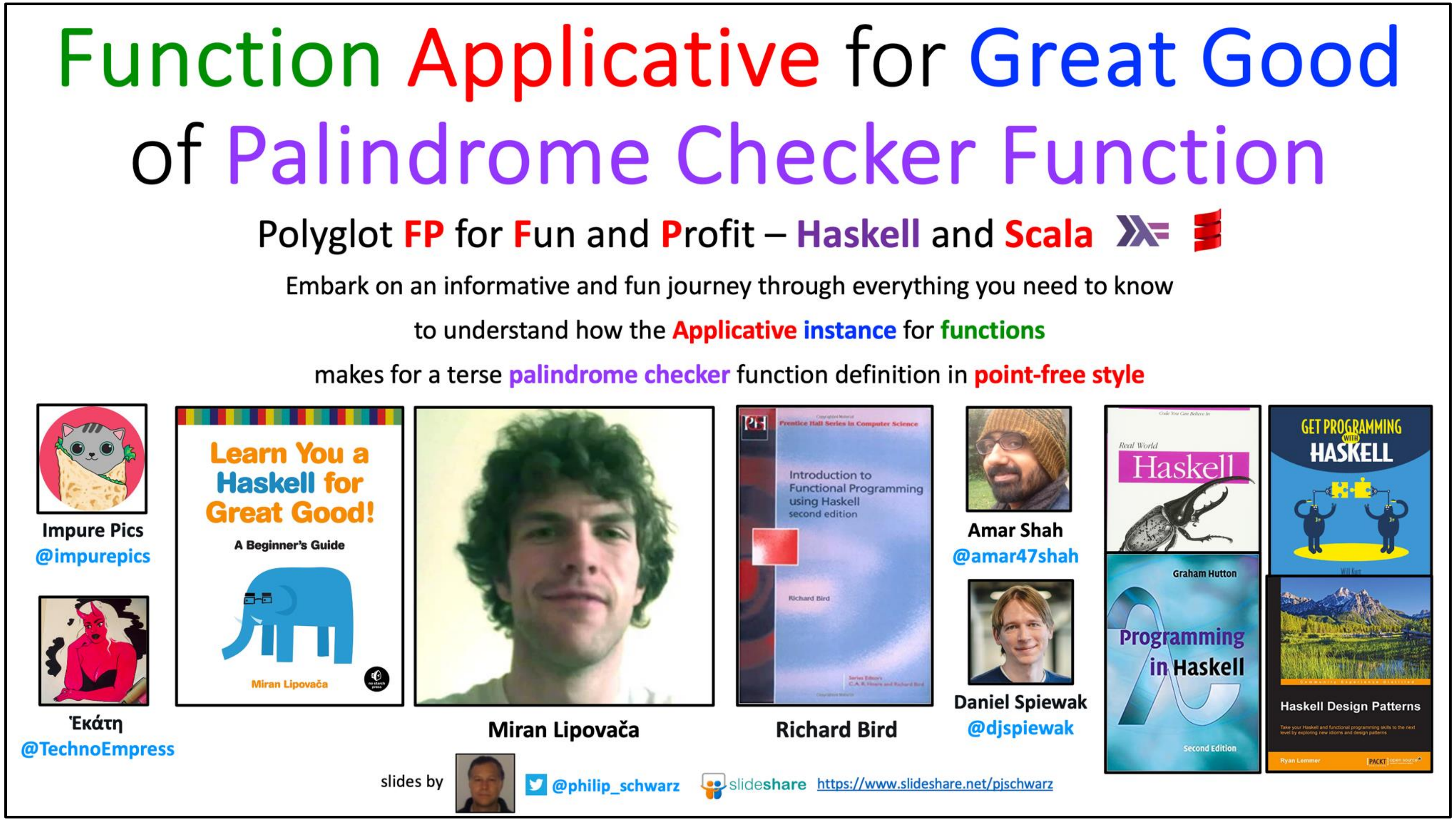
```scala
val leapYear2: Int => Boolean =
 liftA2_v2(`(>)`)(gcd(80), gcd(50))
```

```scala
val leapYear3: Int => Boolean =
  liftA2_v3(`(>)`)(gcd(80), gcd(50))
```

```scala
for
  leapYear <- List(leapYear1, leapYear2, leapYear3)
  _ = assert(List.range(2000,2025).filter(leapYear) == List(2000, 2004, 2008, 2012, 2016, 2020, 2024))
  _ = assert(List(1600, 1700, 1800, 1900, 2000).filter(leapYear) == List(1600, 2000))
yield ()
```

That’s all. I hope you found it useful.

If you would like a more comprehensive introduction to the **Function Applicative**, consider checking out the following deck.



FP Iλλuminated https://fpilluminated.com/