Coding by Shape

Andy Scott

September 4, 2019

who am I?

who am I?

- contributor to a few Scala libs; maintainer of even fewer
- thumbs up to graphs, trees, and recursive structures
- also dogs, hiking, coffee, books, music
- work on Scala & Bazel at Stripe

who am I?

- contributor to a few Scala libs; maintainer of even fewer
- thumbs up to graphs, trees, and recursive structures
- also dogs, hiking, coffee, books, music
- work on Scala & Bazel at Stripe

- github https://github.com/andyscott
- twitter https://twitter.com/andy·g·scott

Goals

Goals

- use Scala to build intuition for some category theory

Goals

- use Scala to build intuition for some category theory

  code $\to$ diagrams

  diagrams $\to$ code

Goals

- use Scala to build intuition for some category theory

  code $\rightarrow$ diagrams

  diagrams $\rightarrow$ code

- keep it simple, cover a few common patterns (mostly)

Goals

- use Scala to build intuition for some category theory

  code $\rightarrow$ diagrams

  diagrams $\rightarrow$ code

- keep it simple, cover a few common patterns (mostly)

  folds!

Goals

- use Scala to build intuition for some category theory

  code → diagrams

  diagrams → code

- keep it simple, cover a few common patterns (mostly)

  folds!

- have fun

the real treasure is the friends we make along the way

— Jon Pretty

Goals

- use Scala to build intuition for some category theory

  code → diagrams

  diagrams → code

- keep it simple, cover a few common patterns (mostly)

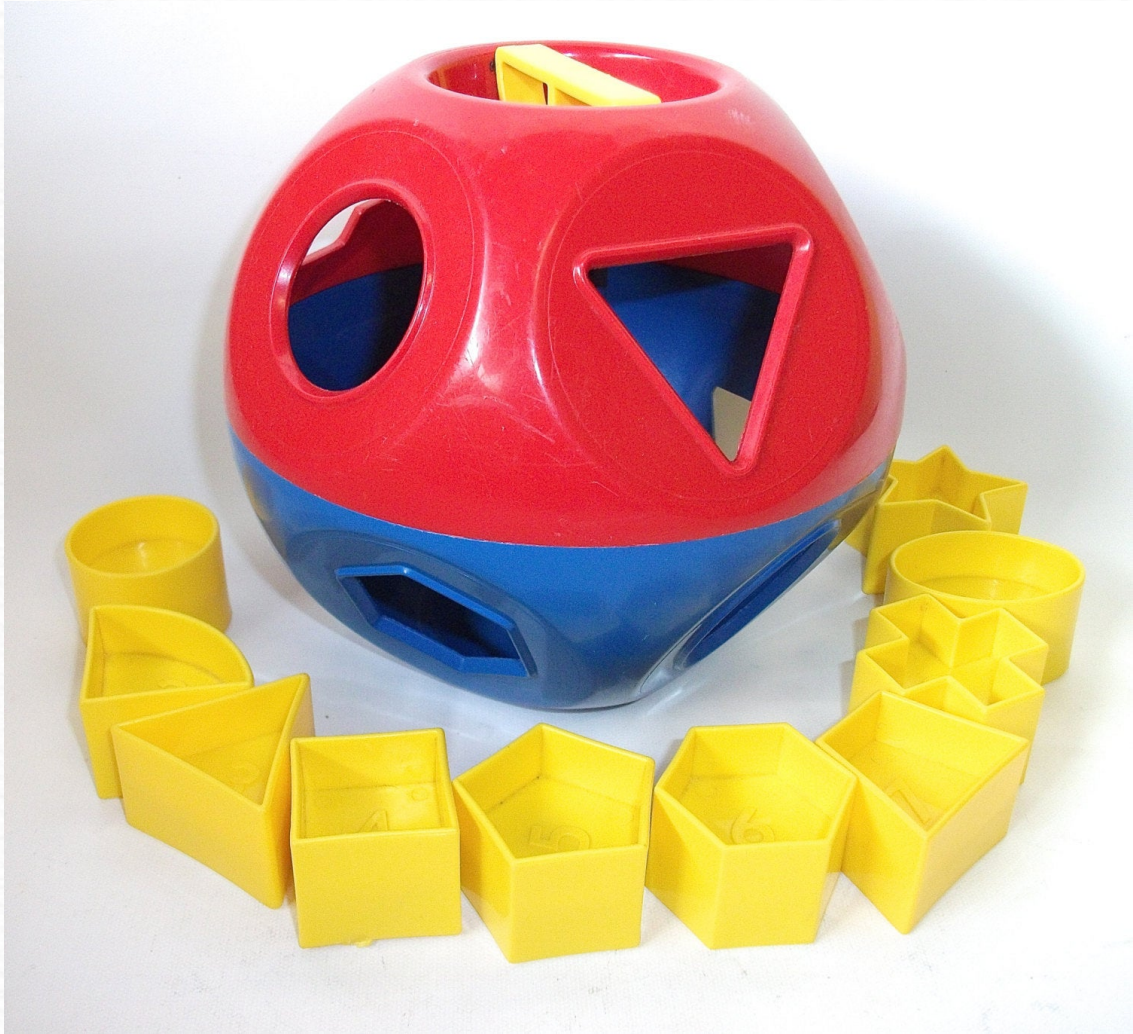
  folds!

- have fun

the real treasure is the ~~*friends*~~ *we* ~~*make*~~ *along the way*

folds *learn*

— Jon Pretty



Tupperware Shape-O

Outline

Start

↓

Intro

↓

Simple Diagrams

↓

Either, fold

↓

Option, fold

↓

List, foldRight

↓

List, hylo

↓

End

Commutative Diagrams

- strongly rooted in math and category theory
- useful for showing/proving/checking laws e.g. functor identity
- often used for "diagram chasing" i.e. theorem proving

Commutative Diagrams

- strongly rooted in math and category theory
- useful for showing/proving/checking laws e.g. functor identity
- often used for "diagram chasing" i.e. theorem proving

... we're not going to cover any of that in this talk

Commutative Diagram 101

Commutative Diagram 101

- nodes (objects)

Commutative Diagram 101

- nodes (objects)

😄

Commutative Diagram 101

- nodes (objects)
- edges (morphisms)

😄

Commutative Diagram 101

- nodes (objects)
- edges (morphisms)

$$\begin{tikzcd}
\text{Edinburgh} \arrow[d] \\
\text{😄} \\
\text{Manchester} \arrow[u] \\
\text{Amsterdam} \arrow[ul]
\end{tikzcd}$$

Commutative Diagram 101

- nodes (objects)
- edges (morphisms)

$$\begin{tikzcd}
\text{Edinburgh} \arrow[d] \\
\text{😄} \\
\text{Manchester} \arrow[u] \\
\text{Amsterdam} \arrow[ul]
\end{tikzcd}$$

Commutative Diagram 101

- nodes (objects)
- edges (morphisms)

$$
\begin{tikzcd}
 & \text{Edinburgh} \arrow[d] \\
 & \text{😄} \\
 & \text{Manchester} \arrow[u] \\
\text{Seattle} \arrow[r] \arrow[uur, bend left] \arrow[uurr, bend left] & \text{Chicago} \arrow[uur] & \text{Amsterdam} \arrow[ul]
\end{tikzcd}
$$

Commutative Diagram 101

- nodes (objects)
- edges (morphisms)

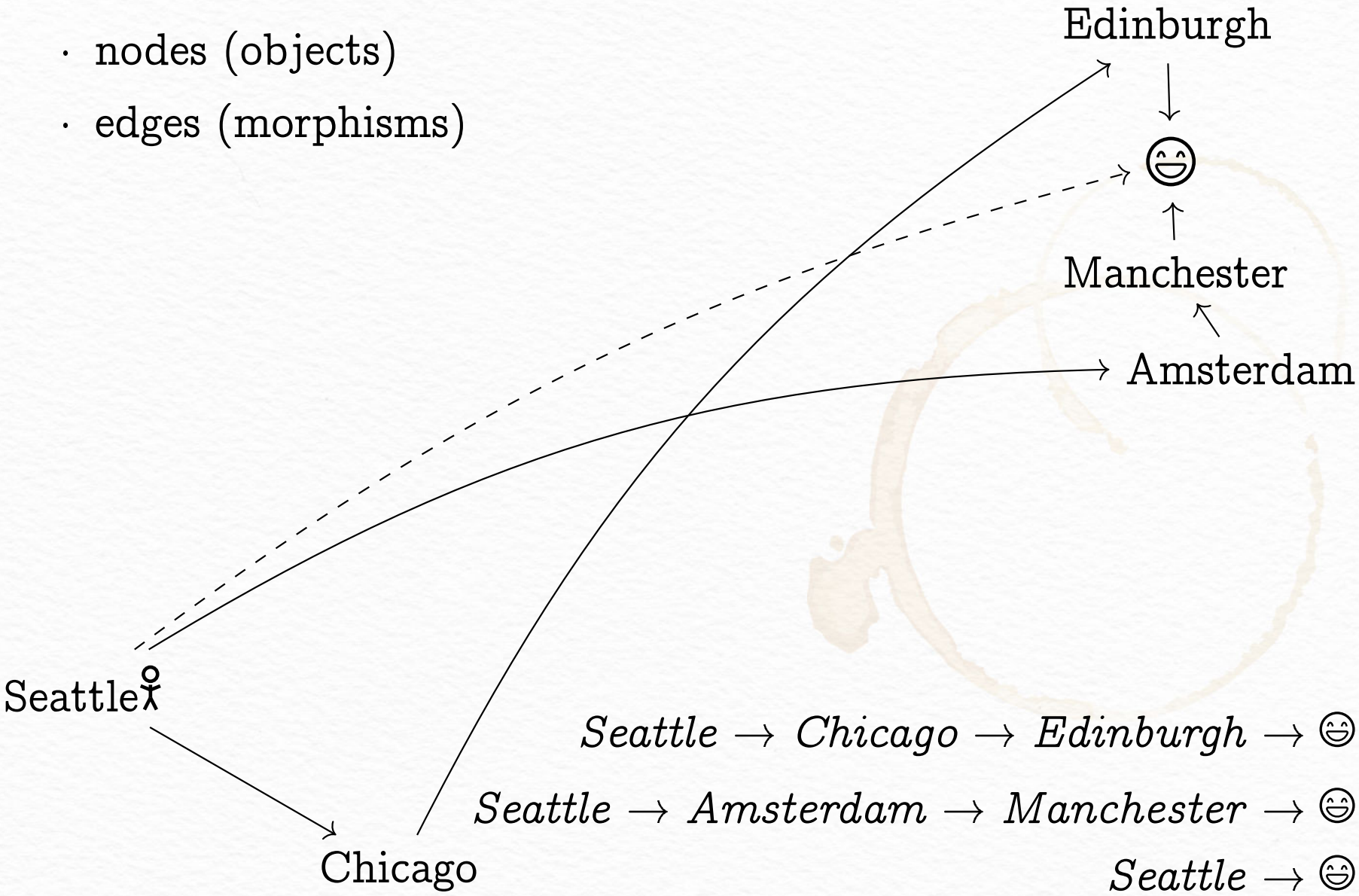
$$\begin{tikzcd}
 & & \text{Edinburgh} \arrow[d] \\
 & & \text{😄} \\
 & & \text{Manchester} \arrow[u] \\
 & & \text{Amsterdam} \arrow[ul] \\
\text{Seattle} \arrow[urr, bend left] \arrow[dr] & & \\
 & \text{Chicago} \arrow[uur] &
\end{tikzcd}$$

$$\textit{Seattle} \to \textit{Chicago} \to \textit{Edinburgh} \to \text{😄}$$

$$\textit{Seattle} \to \textit{Amsterdam} \to \textit{Manchester} \to \text{😄}$$

Commutative Diagram 101

- nodes (objects)
- edges (morphisms)



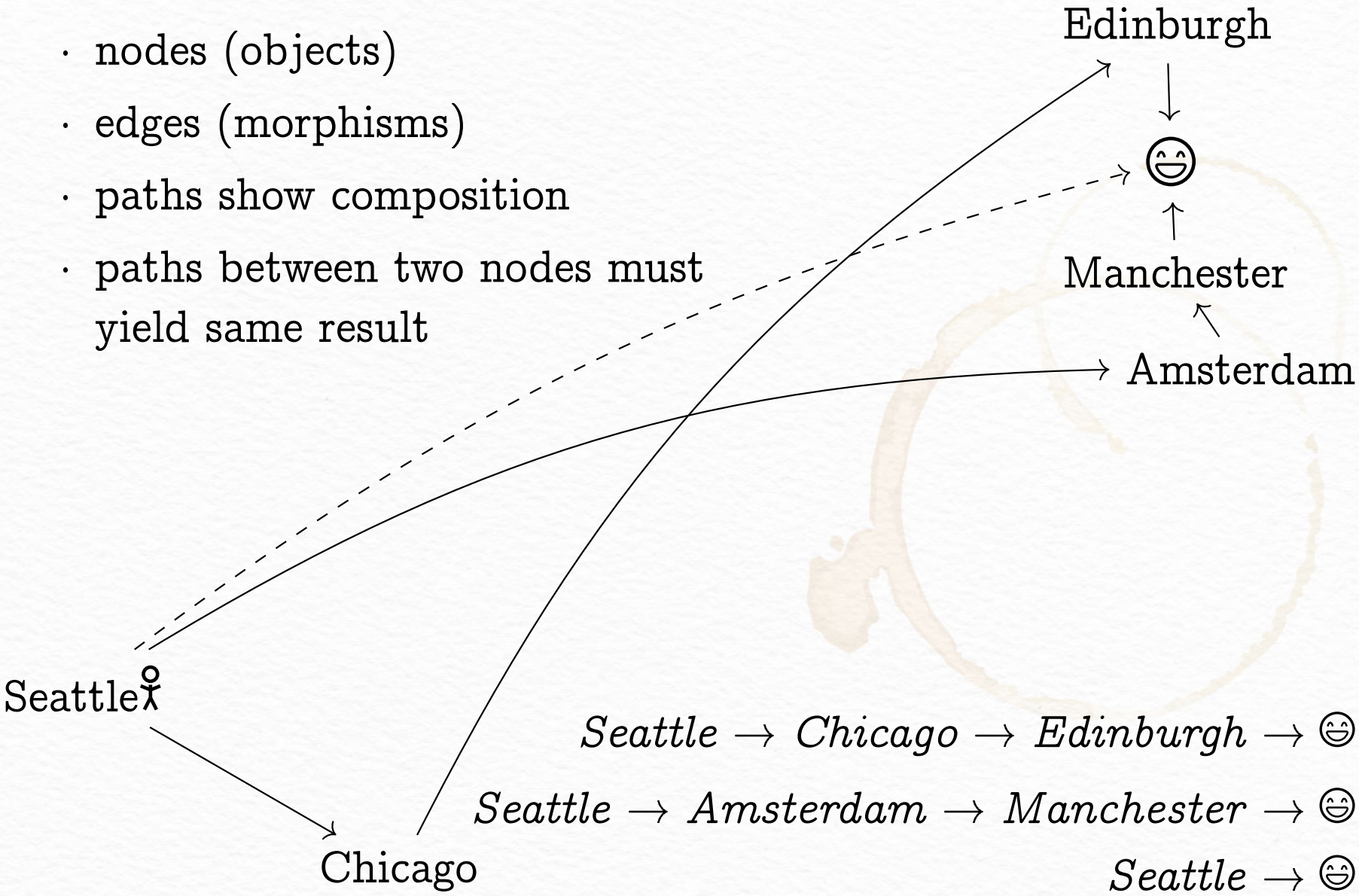
$$\mathit{Seattle} \to \mathit{Chicago} \to \mathit{Edinburgh} \to 😄$$

$$\mathit{Seattle} \to \mathit{Amsterdam} \to \mathit{Manchester} \to 😄$$

$$\mathit{Seattle} \to 😄$$

Commutative Diagram 101

- nodes (objects)
- edges (morphisms)
- paths show composition
- paths between two nodes must yield same result



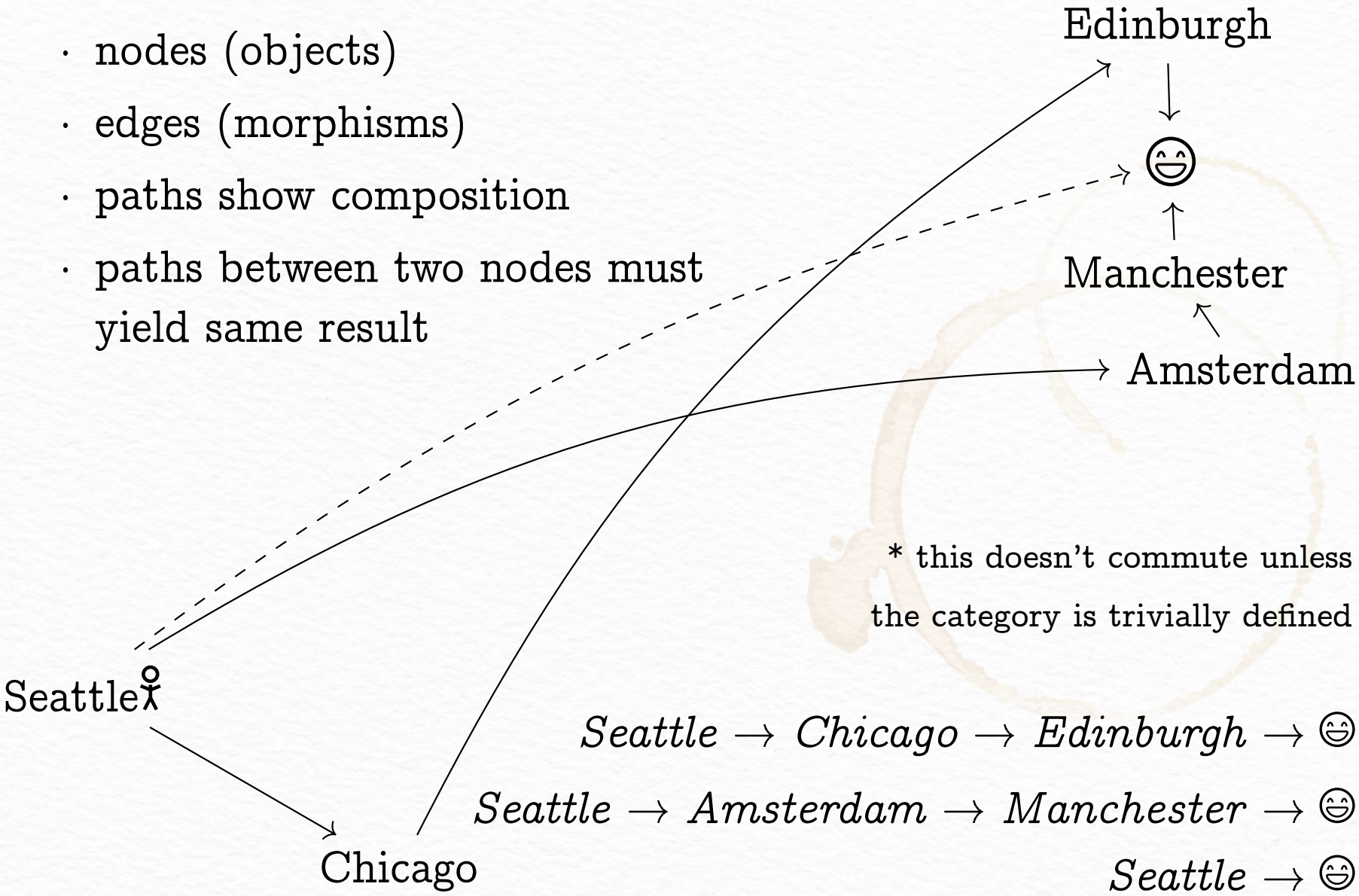
$$\mathit{Seattle} \to \mathit{Chicago} \to \mathit{Edinburgh} \to 😄$$

$$\mathit{Seattle} \to \mathit{Amsterdam} \to \mathit{Manchester} \to 😄$$

$$\mathit{Seattle} \to 😄$$

Commutative* Diagram 101

- nodes (objects)
- edges (morphisms)
- paths show composition
- paths between two nodes must yield same result



* this doesn't commute unless the category is trivially defined

$$Seattle \to Chicago \to Edinburgh \to 😄$$

$$Seattle \to Amsterdam \to Manchester \to 😄$$

$$Seattle \to 😄$$

A Scala Diagram

$$List[String]$$

A Scala Diagram

$List[String]$

```scala
val x: List[String] =
  List("hello", "scala", "world")

val y: Int = x
  .map(_.length)
  .foldLeft(0)(_ + _) // 15
```

A Scala Diagram

$$List[String] \longrightarrow Int$$

```scala
val x: List[String] =
  List("hello", "scala", "world")

val y: Int = x
  .map(_.length)
  .foldLeft(0)(_ + _) // 15
```

A Scala Diagram

$$List[String] \longrightarrow Int$$

```scala
val x: List[String] =
  List("hello", "scala", "world")

def f(x: List[String]): List[Int] =
  x.map(_.length)
def g(x: List[Int]): Int =
  x.foldLeft(0)(_ + _)

val y: Int = g(f(x)) // 15
```

A Scala Diagram

$$List[String] \longrightarrow Int$$

```scala
val x: List[String] =
  List("hello", "scala", "world")

def f(x: List[String]): List[Int] =
  x.map(_.length)
def g(x: List[Int]): Int =
  x.foldLeft(0)(_ + _)

val y: Int = g(f(x)) // 15
```

A Scala Diagram

$$List[String] \xrightarrow{f} List[Int] \xrightarrow{g} Int$$

```scala
val x: List[String] =
  List("hello", "scala", "world")

def f(x: List[String]): List[Int] =
  x.map(_.length)
def g(x: List[Int]): Int =
  x.foldLeft(0)(_ + _)

val y: Int = g(f(x)) // 15
```

A Scala Diagram

$$List[String] \xrightarrow{f} List[Int] \xrightarrow{g} Int$$

```scala
val x: List[String] =
  List("hello", "scala", "world")

def f(x: List[String]): List[Int] =
  x.map(_.length)
def g(x: List[Int]): Int =
  x.foldLeft(0)(_ + _)

def h(x: List[String]): Int = g(f(x))
// val h = g _ compose f
val y: Int = h(x) // 15
```

A Scala Diagram

$$\begin{tikzcd} List[String] \arrow[r, "f"] \arrow[rr, bend right, "h"'] & List[Int] \arrow[r, "g"] & Int \end{tikzcd}$$

```scala
def f(x: List[String]): List[Int] =
  x.map(_.length)
def g(x: List[Int]): Int =
  x.foldLeft(0)(_ + _)


def h(x: List[String]): Int = g(f(x))
// val h = g _ compose f
val y: Int = h(x) // 15
```

A Scala Diagram

$$\begin{tikzcd} List[String] \arrow[r, "f"] \arrow[rr, bend right, "g \circ f"'] & List[Int] \arrow[r, "g"] & Int \end{tikzcd}$$

```scala
def f(x: List[String]): List[Int] =
  x.map(_.length)
def g(x: List[Int]): Int =
  x.foldLeft(0)(_ + _)


def h(x: List[String]): Int = g(f(x))
// val h = g _ compose f
val y: Int = h(x) // 15
```

Review

- composition

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, bend right, "h = g \circ f"'] & B \arrow[r, "g"] & C \end{tikzcd}$$

```
def h(x: A): C = g(f(x))
```

Outline

Start

$\downarrow$

Intro

$\downarrow$

Simple Diagrams

$\downarrow$

Either, fold

$\downarrow$

Option, fold

$\downarrow$

List, foldRight

$\downarrow$

List, hylo

$\downarrow$

End

Outline

$$
\begin{array}{c}
\text{Start} \\
\downarrow \\
\text{Intro} \\
\downarrow \\
\text{Simple Diagrams} \\
\downarrow s \\
\text{Either, fold} \\
\downarrow o \\
\text{Option, fold} \\
\downarrow i \\
\text{List, foldRight} \\
\downarrow d \\
\text{List, hylo} \\
\downarrow a \\
\text{End}
\end{array}
$$

Outline

$$
\begin{tikzcd}
\text{Start} \arrow[d] \\
\text{Intro} \arrow[d] \\
\text{Simple Diagrams} \arrow[d, "s"] \\
\text{Either, fold} \arrow[d, "o"] \\
\text{Option, fold} \arrow[d, "i"] \\
\text{List, foldRight} \arrow[d, "d"] \\
\text{List, hylo} \arrow[d, "a"] \\
\text{End} \arrow[uuuuuuu, bend left, dashed, "\text{adios} = \mathtt{a} \circ \mathtt{d} \circ \mathtt{i} \circ \mathtt{o} \circ \mathtt{s}"]
\end{tikzcd}
$$

Thanks!

Outline

Start

↓

Intro

↓

Simple Diagrams

↓

Either, fold

↓

Option, fold

↓

List, foldRight

↓

List, hylo

↓

End

Folding Coproducts

$$Either[A, B]$$

Folding Coproducts

$$A \xrightarrow{\quad left \quad} Either[A, B]$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
```

Folding Coproducts

$$A \xrightarrow{\quad left \quad} Either[A, B]$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

$$B \xrightarrow{\quad right \quad} Either[A, B]$$

Folding Coproducts

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Coproducts

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

Folding Coproducts

$$
\begin{tikzcd}
A \arrow[r, "left"] & Either[A, B] \arrow[d, "fold"] & B \arrow[l, "right"'] \\
 & Z &
\end{tikzcd}
$$

Folding Coproducts

$$
\begin{tikzcd}
A \arrow[r, "left"] & Either[A, B] \arrow[d, "fold"] & B \arrow[l, "right"'] \\
 & Z &
\end{tikzcd}
$$

```
def fold[A, B, Z](e: Either[A, B])
  (f: A => Z, g: B => Z): Z = e match {
    case Left(a)  = f(a)
    case Right(b) = g(b)
  }
```

Folding Coproducts

$$
\begin{tikzcd}
A \arrow[r, "left"] \arrow[dr, "f"'] & Either[A, B] \arrow[d, "fold"] & B \arrow[l, "right"'] \\
& Z &
\end{tikzcd}
$$

```
def fold[A, B, Z](e: Either[A, B])
  (f: A => Z, g: B => Z): Z = e match {
    case Left(a)  = f(a)
    case Right(b) = g(b)
  }
```

Folding Coproducts

$$
\begin{tikzcd}
A \arrow[r, "left"] \arrow[dr, "f"'] & Either[A, B] \arrow[d, "fold"] & B \arrow[l, "right"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}
$$

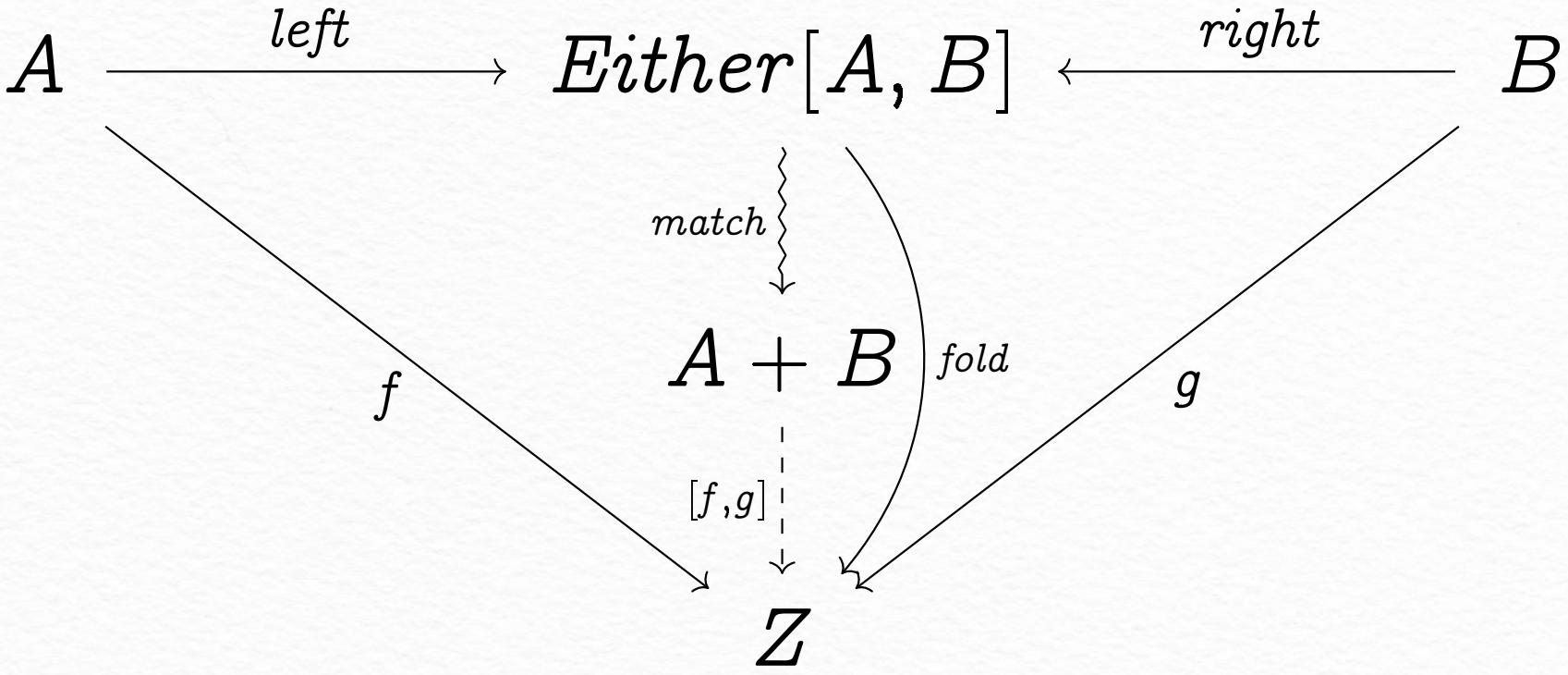
```
def fold[A, B, Z](e: Either[A, B])
  (f: A => Z, g: B => Z): Z = e match {
    case Left(a)  = f(a)
    case Right(b) = g(b)
  }
```

Folding Coproducts

$$
\begin{tikzcd}
A \arrow[r, "left"] \arrow[dr, "f"'] & Either[A, B] \arrow[d, "fold=[f,g]"] & B \arrow[l, "right"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}
$$

```
def fold[A, B, Z](e: Either[A, B])
  (f: A => Z, g: B => Z): Z = e match {
    case Left(a)  = f(a)
    case Right(b) = g(b)
  }
```

Folding Coproducts




```
def fold[A, B, Z](e: Either[A, B])
  (f: A => Z, g: B => Z): Z = e match {
    case Left(a)  = f(a)
    case Right(b) = g(b)
  }
```

Folding Coproducts

$$
\begin{tikzcd}
A \arrow[r, "i_A"] \arrow[dr, "f"'] & A + B \arrow[d, dashed, "{[f,g]}"'] & B \arrow[l, "i_B"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}
$$

```
def fold[A, B, Z](e: Either[A, B])
  (f: A => Z, g: B => Z): Z = e match {
    case Left(a)  = f(a)
    case Right(b) = g(b)
  }
```

Review

- two arrows

$$A \xrightarrow{\quad f \quad} C$$

$$B \xrightarrow{\quad g \quad} C$$

```
def f(a: A): C = //...
def g(b: B): C = //...
```

$$A + B \xrightarrow[\;[f, g]\;]{} C$$

- unpack

$$Either[A, B] \xrightarrow{\;match\;} A + B$$

Folding Option

$$Option[A]$$

Folding Option

$$Option[A]$$

```
sealed trait Option[A]
case class  Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

Folding Option

$$Option[A]$$

```
sealed trait Option[A]
case class  Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

$$Either[A, B]$$

Folding Option

$$Option[A]$$

```
sealed trait Option[A]
case class  Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

$$Either[A, B]$$

```
sealed trait Either[A, B]
case class Left [A, B](a: A) extends Either[A, B]
case class Right[A, B](b: B) extends Either[A, B]
```

Folding Option

$$Option[A]$$

```
sealed trait Option[A]
case class  Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

$$Either[A, B]$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$Option[A]$$

```
sealed trait Option[A]
case class  Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$Option[A]$$

```
def none[A]       : Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$Option[A] \xleftarrow{\quad some \quad} A$$

```
def none[A]       : Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$\mathit{Option}[A] \xleftarrow{\mathit{some}} A$$

```
def none[A](u: Unit): Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\mathit{left}} \mathit{Either}[A, B] \xleftarrow{\mathit{right}} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$Unit \xrightarrow{\quad none \quad} Option[A] \xleftarrow{\quad some \quad} A$$

```
def none[A](u: Unit): Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$1 \xrightarrow{\quad none \quad} Option[A] \xleftarrow{\quad some \quad} A$$

```
def none[A](u: Unit): Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$1 \xrightarrow{\quad none \quad} Option[A] \xleftarrow{\quad some \quad} A$$

```
def none[A]       : Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\quad left \quad} Either[A, B] \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$1 \xrightarrow{\quad none \quad} Option[A] \xleftarrow{\quad some \quad} A$$

```
def none[A]       : Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\quad left \quad} A + B \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$1 \xrightarrow{\quad none \quad} 1 + A \xleftarrow{\quad some \quad} A$$

```
def none[A]       : Option[A] = None
def some[A](a: A): Option[A] = Some(a)
```

$$A \xrightarrow{\quad left \quad} A + B \xleftarrow{\quad right \quad} B$$

```
def left [A, B](a: A): Either[A, B] = Left(a)
def right[A, B](b: B): Either[A, B] = Right(b)
```

Folding Option

$$A \xrightarrow{\quad some \quad} Option[A]$$

$$1 \xrightarrow{\quad none \quad} Option[A]$$

Folding Option

$$A \xrightarrow{\quad some \quad} Option[A]$$

$$1 \xrightarrow{\quad none \quad} Option[A]$$

$$1 + A \xrightarrow{[empty, some]} Option[A]$$

Folding Option

$$A \xrightarrow{\quad some \quad} Option[A]$$

$$1 \xrightarrow{\quad none \quad} Option[A]$$

$$1 + A \xrightarrow{\quad [empty, some] \quad} Option[A]$$

$$1 + A \xleftarrow{\quad match \quad} Option[A]$$

```scala
def fold[A, B](x: Option[A])
              (ifEmpty: B)(f: A => B): B =
  x match {
    case None     => ifEmpty
    case Some(a) => f(a)
  }
```

```scala
def fold[A, B](x: Option[A])
              (ifEmpty: B)(f: A => B): B =
  x match {
    case None     => ifEmpty
    case Some(a) => f(a)
  }
```

$$\mathit{Option}[A] \xrightarrow{\mathit{fold}} B$$

$$1 \xrightarrow{\mathit{ifEmpty}} B$$

$$A \xrightarrow{f} B$$

```scala
def fold[A, B](x: Option[A])
              (ifEmpty: B)(f: A => B): B =
  x match {
    case None     => ifEmpty
    case Some(a) => f(a)
  }
```

$$Option[A] \xrightarrow{\quad fold \quad} B$$

$$1 + A \xrightarrow{\quad [ifEmpty, f] \quad} B$$

```
def fold[A, B](x: Option[A])
              (ifEmpty: B)(f: A => B): B =
  x match {
    case None     => ifEmpty
    case Some(a) => f(a)
  }
```

$$\begin{tikzcd}
Option[A] \arrow[rr, "fold"] & & B \\
& 1 + A \arrow[ur, "{[ifEmpty, f]}"'] &
\end{tikzcd}$$

```scala
def fold[A, B](x: Option[A])
              (ifEmpty: B)(f: A => B): B =
  x match {
    case None     => ifEmpty
    case Some(a) => f(a)
  }
```

$$\begin{tikzcd}
Option[A] \arrow[rr, "fold"] \arrow[dr, "match"'] & & B \\
& 1 + A \arrow[ur, "{[ifEmpty, f]}"'] &
\end{tikzcd}$$

Review

- unpack

$$Either[A, B] \xrightarrow{\quad match \quad} A + B$$

$$Option[A] \xrightarrow{\quad match \quad} 1 + A$$

- constants

$$1 \xrightarrow{\quad a \quad} A$$

```
val a: A = //...
def a: A = //...
def a(unit: Unit => A): A = //...
```

List foldRight

```scala
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }

val list = "hello" :: "scala" :: "world" :: Nil
val f: (String, Int) => Int = _.length + _
foldRight(list)(0)(f)
// 15
```

creating/unpacking lists

$$1 \xrightarrow{\quad nil \quad} List[A]$$

$$A \times List[A] \xrightarrow{\quad cons \quad} List[A]$$

```
def nil[A]: List[A] = Nil
def cons[A](head: A, tail: List[A]): List[A] =
  head :: tail
```

$$1 + A \times List[A] \xrightarrow{\quad [nil, cons] \quad} List[A]$$

$$1 + A \times List[A] \xleftarrow{\quad match \quad} List[A]$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$List[A] \xrightarrow{foldRight} B$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$List[A] \xrightarrow{foldRight} B$$

$$1 \xrightarrow{z} B$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$List[A] \xrightarrow{foldRight} B$$

$$1 \xrightarrow{z} B$$

$$A \times B \xrightarrow{f} B$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$List[A] \xrightarrow{foldRight} B$$

$$1 + A \times B \xrightarrow{[z,f]} B$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$\begin{tikzcd}
List[A] \arrow[r, "foldRight"] & B \\
& 1 + A \times B \arrow[u, "{[z,f]}"']
\end{tikzcd}$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$
\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
1 + A \times List[A] & 1 + A \times B \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$
\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
1 + A \times List[A] \arrow[r, "id+\ldots"'] & 1 + A \times B \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$
\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
1 + A \times List[A] \arrow[r, "id+id\times\ldots"'] & 1 + A \times B \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

List foldRight

```
def foldRight[A, B](la: List[A])
  (z: B)(f: (A, B) => B): B = la match {
    case Nil => z
    case head :: tail => f(head,
        foldRight(tail)(z)(f))
  }
```

$$
\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
1 + A \times List[A] \arrow[r, "id + id \times foldRight"'] & 1 + A \times B \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

foldRight intuition

```scala
def foldRight[A, B](
  la: List[A]
)(z: B)(f: (A, B) => B): B = // ...

val list0 = "hello" :: "scala" :: "world" :: Nil
val f: (String, Int) => Int = _.length + _
foldRight(list0)(0)(f)
// 15
```

foldRight intuition

```scala
def foldRight[A, B](
  la: List[A]
)(z: B)(f: (A, B) => B): B = // ...

val list0 = "hello" :: "scala" :: "world" :: Nil
val f: (String, Int) => Int = _.length + _
foldRight(list0)(0)(f)
// 15

val list1 = ::("hello", ::("scala", ::("world", Nil)))
foldRight(list1)(0)(f)
// 15
```

foldRight intuition

```
::("hello", ::("scala", ::("world", Nil))) // list1
```

foldRight intuition

```
::("hello", ::("scala", ::("world", Nil))) // list1

 f("hello",  f("scala",  f("world", 0)))   // 15
```

foldRight intuition

```
::("hello", ::("scala", ::("world", Nil))) // list1

 f("hello",  f("scala",  f("world", 0)))   // 15
 f("hello",  f("scala",  5))               // 15
```

foldRight intuition

```
::("hello", ::("scala", ::("world", Nil))) // list1

 f("hello",  f("scala",  f("world", 0)))   // 15
 f("hello",  f("scala",  5))               // 15
 f("hello",  10)                           // 15
```

foldRight intuition

```
::("hello", ::("scala", ::("world", Nil))) // list1

 f("hello",  f("scala",  f("world", 0)))   // 15
 f("hello",  f("scala",  5))               // 15
 f("hello",  10)                           // 15
 15                                        // 15 :)
```

foldRight intuition

```
::("hello", ::("scala", ::("world", Nil))) // list1

 f("hello",  f("scala",  f("world", 0)))   // 15
 f("hello",  f("scala",  5))               // 15
 f("hello",  10)                           // 15
 15                                        // 15 :)

 f(list.head,                    // "hello".length +
   f(list.tail.head,             // "scala".length +
     f(list.tail.tail.head,      // "world".length +
       0)))                      // 0
```

from foldRight

```scala
// foldRight just for list
def foldRight[A, B](
  la: List[A]
)(z: B)(f: (A, B) => B): B = la match {
  case Nil => z
  case head :: tail => f(head, foldRight(tail)(z)(f))
}
```

from foldRight to hylo

```
// foldRight just for list
def foldRight[A, B](
  la: List[A]
)(z: B)(f: (A, B) => B): B = la match {
  case Nil => z
  case head :: tail => f(head, foldRight(tail)(z)(f))
}
```

```
// 'foldRight' (and lots more)
// for any recursive data structure
def hylo[F[_]: Functor, A, B](a: A)(
  alg  : F[B] =>   B,
  coalg:   A  => F[A]
): B =
  alg(coalg(a).map(hylo(_)(alg, coalg)))
```

from foldRight to hylo

```scala
val input = List("hello", "scala", "world")

val z: Int = 0
val f: (String, Int) => Int = _.length + _

foldRight(input)(z)(f)
// 15
```

from foldRight to hylo

```scala
val input = List("hello", "scala", "world")

val z: Int = 0
val f: (String, Int) => Int = _.length + _

foldRight(input)(z)(f)
// 15
```

```scala
def alg: Option[(String, Int)] => Int =
  _ match {
    case None => 0
    case Some((head, acc)) => head.length + acc
  }

hylo(input)(alg, coalg) // coalg unpacks a list
// 15
```

from foldRight to hylo

- We're not going to prove anything
- We're just going to use our understanding of Scala and foldRight to gain intuition for how recursion schemes work.

reshaping foldRight

$$
\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
1 + A \times List[A] \arrow[r, "id + id \times foldRight"'] & 1 + A \times B \arrow[u, "{[z, f]}"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
\boxed{List[A]} \arrow[rr, "foldRight"] \arrow[d, squiggly, "match"'] & & B \\
1 + A \times \boxed{List[A]} \arrow[rr, "id + id \times foldRight"'] & & 1 + A \times \boxed{B} \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
1 + A \times List[A] \arrow[r, "id + id \times foldRight"'] & 1 + A \times B \arrow[u, "{[z, f]}"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
F[List[A]] \arrow[r, "id+id\times foldRight"'] & F[B] \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
A \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
F[A] \arrow[r, "id+id\times foldRight"'] & F[B] \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, squiggly, "match"'] & B \\
F[A] \arrow[r, "id+id\times foldRight"'] & F[B] \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, squiggly, "match"'] & B \\
F[A] \arrow[r, "map\ \ hylo"'] & F[B] \arrow[u, "{[z,f]}"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, squiggly, "match"'] & B \\
F[A] \arrow[r, "map\ \ hylo"'] & F[B] \arrow[u, "algebra"']
\end{tikzcd}
$$

reshaping foldRight

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, "coalgebra"'] & B \\
F[A] \arrow[r, "map\ \ hylo"'] & F[B] \arrow[u, "algebra"']
\end{tikzcd}
$$

hylo

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, "coalgebra"'] & B \\
F[A] \arrow[r, "map\ hylo"'] & F[B] \arrow[u, "algebra"']
\end{tikzcd}
$$

```
def hylo[F[_]: Functor, A, B](a: A)(
  alg  : F[B] =>   B,
  coalg:   A  => F[A]
): B =
  //...
```

hylo

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, "coalgebra"'] & B \\
F[A] \arrow[r, "map\ hylo"'] & F[B] \arrow[u, "algebra"']
\end{tikzcd}
$$

```
def hylo[F[_]: Functor, A, B](a: A)(
  alg  : F[B] =>   B,
  coalg:   A  => F[A]
): B =
      coalg(a)
```

hylo

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, "coalgebra"'] & B \\
F[A] \arrow[r, "map\ hylo"'] & F[B] \arrow[u, "algebra"']
\end{tikzcd}
$$

```
def hylo[F[_]: Functor, A, B](a: A)(
  alg  : F[B] =>   B,
  coalg:   A  => F[A]
): B =
      coalg(a).map(hylo(...          ))
```

hylo

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, "coalgebra"'] & B \\
F[A] \arrow[r, "map\ hylo"'] & F[B] \arrow[u, "algebra"']
\end{tikzcd}
$$

```
def hylo[F[_]: Functor, A, B](a: A)(
  alg  : F[B] =>   B,
  coalg:   A  => F[A]
): B =
      coalg(a).map(hylo(_)(alg, coalg))
```

hylo

$$
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, "coalgebra"'] & B \\
F[A] \arrow[r, "map\ hylo"'] & F[B] \arrow[u, "algebra"']
\end{tikzcd}
$$

```
def hylo[F[_]: Functor, A, B](a: A)(
  alg  : F[B] =>   B,
  coalg:   A  => F[A]
): B =
  alg(coalg(a).map(hylo(_)(alg, coalg)))
```

hylo intuition

```scala
val input = List("hello", "scala", "world")
val f: (String, Int) => Int = _.length + _

foldRight(input)(0)(f) // 15
```

hylo intuition

```scala
val input = List("hello", "scala", "world")
// F[A] = Option[(String, A)] = 1 + String x A

val coalg: List[String] =>
  Option[(String, List[String])] =
  _ match {
    case Nil            => None
    case head :: tail => Some((head, tail))
  }

val alg: Option[(String, Int)] => Int =
  _ match {
    case None                 => 0
    case Some((head, acc)) => head.length + acc
  }

hylo(input)(alg, coalg) // 15
```

hylo intuition

```
f(list.head,                    // "hello".length +
  f(list.tail.head,             // "scala".length +
    f(list.tail.tail.head,      // "world".length +
      z)))                      // 0

alg(coalg(list).fmap(ist => // "hello".length +
  alg(coalg(ist).fmap(st => // "scala".length +
    alg(coalg(st).fmap(t => // "world".length +
      alg(coalg(t).map(_ => // 0
        ???)))))))))
```

List foldRight & hylo

$$\begin{tikzcd}
List[A] \arrow[r, "foldRight"] \arrow[d, squiggly, "match"'] & B \\
1 + A \times List[A] \arrow[r, "id + id \times foldRight"'] & 1 + A \times B \arrow[u, "{[z,f]}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "hylo"] \arrow[d, "coalg"'] & B \\
F[A] \arrow[r, "map\ \ hylo"'] & F[B] \arrow[u, "alg"']
\end{tikzcd}$$

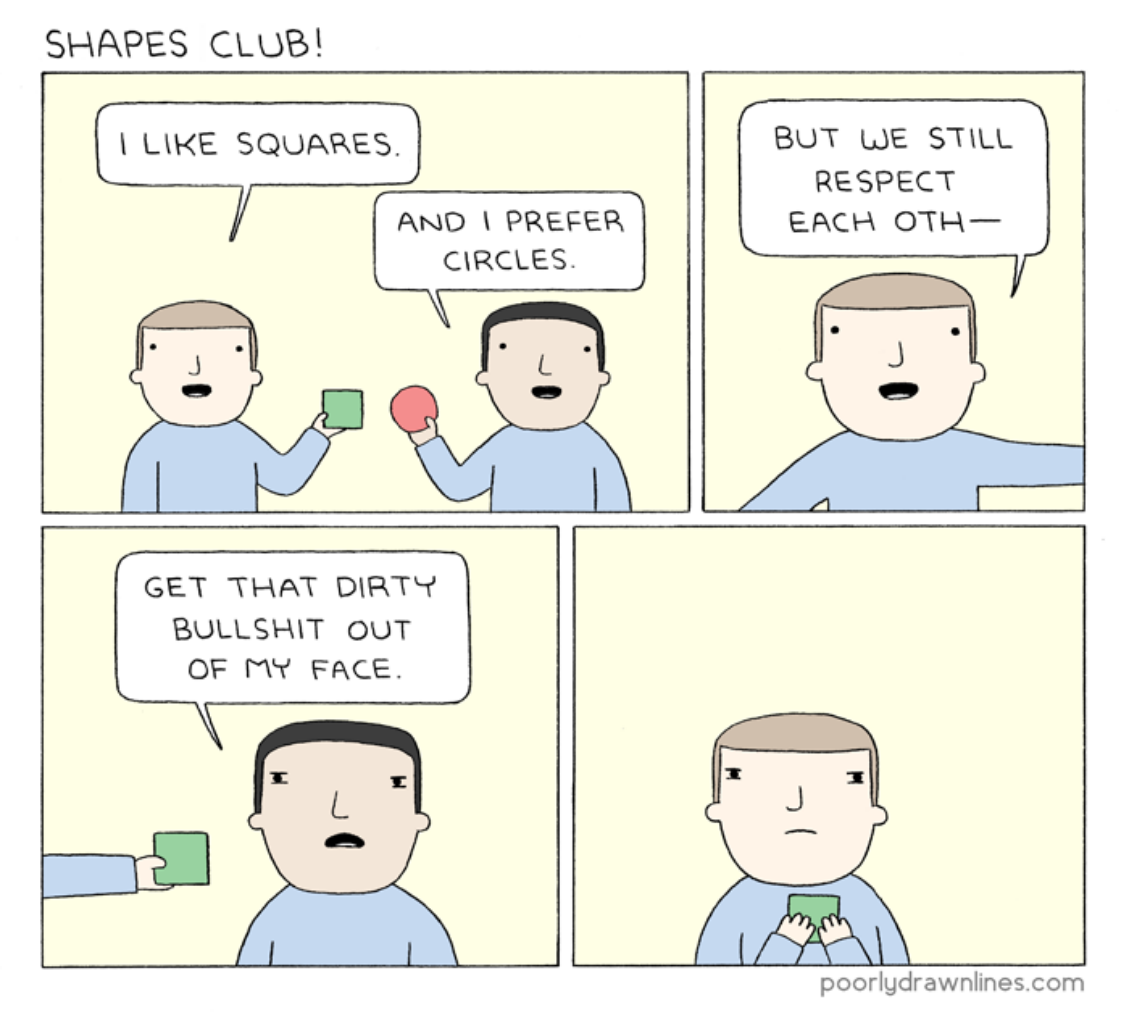
```scala
val list = "hello" :: "scala" :: "world" :: Nil

f(list.head,                      // "hello".length +
  f(list.tail.head,               // "scala".length +
    f(list.tail.tail.head,        // "world".length +
      z)))                        // 0

alg(coalg(list).fmap(ist => // "hello".length +
  alg(coalg(ist).fmap(st => // "scala".length +
    alg(coalg(st).fmap(t => // "world".length +
      alg(coalg(t).map(_ => // 0
        ???))))))))
```

things we learned

- a bit about categories & diagramming
  - composition
  - arrows to show product/coproduct of functions
  - arrows for constants
- a bit about folds in Scala

things we learned

- a bit about categories & diagramming
  - composition
  - arrows to show product/coproduct of functions
  - arrows for constants
- a bit about folds in Scala
- behind the curtain:
  - f-algebras
  - initial f-algebras
  - lambek's lemma
  - ana & cata


SHAPES CLUB!
I LIKE SQUARES.
AND I PREFER CIRCLES.
BUT WE STILL RESPECT EACH OTH—
GET THAT DIRTY BULLSHIT OUT OF MY FACE.
poorlydrawnlines.com

thank you!