Typeclass

Type and Operations

```
case class Person(name: String, age: Int)

def setAge(p: Person, newAge: Int): Person =
  p.copy(age = newAge)

def isAdult(p: Person): Boolean =
  p.age >= 18
```

Polymorphism(s)

Inheritance

```
sealed trait Shape

case class Circle(radius: Double) extends Shape
case class Rectangle(width: Double, height: Double) extends Shape

def double(shape: Shape): Shape =
  shape match {
    case Circle(x)       => Circle(2 * x)
    case Rectangle(w, h) => Rectangle(2 * w, 2 * h)
  }
```

Parametric Polymorphism

```
def headOption[A](xs: List[A]): Option[A] =
  xs match {
    case Nil    => None
    case x :: _ => Some(x)
  }
```

```
scala> headOption(List(1,2,3))
res0: Option[Int] = Some(1)

scala> headOption(List("Hello", "World"))
res1: Option[String] = Some(Hello)
```

Row Polymorphism

```
case class Person(name: String, age: Int)

case class Employee(name: String, age: Int, companyName: String)

def isAdult(x: {val age: Int}): Boolean =
  x.age >= 18
```

```
scala> isAdult(Person("John", 14))
res2: Boolean = false

scala> isAdult(Employee("Eva", 22, "Foo Inc"))
res3: Boolean = true
```

Ad hoc Polymorphism

```
scala> 2 + 3
res4: Int = 5

scala> 2.0 + 3.0
res5: Double = 5.0

scala> BigDecimal(2) + BigDecimal(3)
res6: scala.math.BigDecimal = 5
```

```
scala> List(1,2,3).map(_ + 1)
res7: List[Int] = List(2, 3, 4)

scala> Vector(1,2,3).map(_ + 1)
res8: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

```
scala> Person("John", 34).toString
res9: String = Person(John,34)

scala> false.toString
res10: String = false
```

Plan

- Typeclass definition and comparison with other approaches
- Most common usage for **application developers**
- Use case: folding data

Use case: Equality

Top Level

```
class Object {
  def equals(other: Any): Boolean =
    this == other
}
```

```
case class Foo(i: Int, s: String)
```

Top Level

```
class Object {
  def equals(other: Any): Boolean =
    this == other
}
```

```
case class Foo(i: Int, s: String) extends Object {
  override def equals(obj: Any): Boolean =
    obj match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

Top Level

```
class Object {
  def equals(other: Any): Boolean =
    this == other
}
```

```
case class Foo(i: Int, s: String) extends Object {
  override def equals(obj: Any): Boolean =
    obj match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

```
scala> Foo(5, "Hello").equals(Foo(5, "Hello"))
res11: Boolean = true

scala> 5.equals(6)
res12: Boolean = false
```

Top Level

```
class Object {
  def equals(other: Any): Boolean =
    this == other
}
```

```
case class Foo(i: Int, s: String) extends Object {
  override def equals(obj: Any): Boolean =
    obj match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

```
scala> Foo(5, "Hello").equals(3)
res13: Boolean = false
```

Top Level

```
class Object {
  def equals(other: Any): Boolean =
    this == other
}
```

```
case class Foo(i: Int, s: String) extends Object {
  override def equals(obj: Any): Boolean =
    obj match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

```
val inc1: Int => Int = _ + 1
val inc2: Int => Int = _ + 1
```

```
scala> inc1.equals(inc2)
res14: Boolean = false
```

Mixin Trait

```
trait Eq {
  def eqv(other: Any): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq {
  def eqv(other: Any): Boolean =
    other match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

Mixin Trait

```
trait Eq {
  def eqv(other: Any): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq {
  def eqv(other: Any): Boolean =
    other match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

```
def contains[A <: Eq](xs: List[A], value: A): Boolean =
  xs.foldRight(false)((x, acc) => x.eqv(value) || acc)
```

Mixin Trait

```
trait Eq {
  def eqv(other: Any): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq {
  def eqv(other: Any): Boolean =
    other match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

```
scala> Foo(5, "Hello").eqv(7)
res15: Boolean = false
```

Mixin Trait

```
trait Eq {
  def eqv(other: Any): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq {
  def eqv(other: Any): Boolean =
    other match {
      case x: Foo => i == x.i && s == x.s
      case _      => false
    }
}
```

```
scala> Foo(5, "Hello").eqv(7)
res15: Boolean = false
```

```
scala> 5.eqv(3)
       5.eqv(3)
         ^
On line 2: error: value eqv is not a member of Int
```

F-Bounded Trait

```
trait Eq[A <: Eq[A]]{
  def eqv(other: A): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq[Foo] {
  def eqv(other: Foo): Boolean =
    i == other.i && s == other.s
}
```

F-Bounded Trait

```
trait Eq[A <: Eq[A]]{
  def eqv(other: A): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq[Foo] {
  def eqv(other: Foo): Boolean =
    i == other.i && s == other.s
}
```

```
def contains[A <: Eq[A]](xs: List[A], value: A): Boolean =
  xs.foldRight(false)((x, acc) => x.eqv(value) || acc)
```

F-Bounded Trait

```
trait Eq[A <: Eq[A]]{
  def eqv(other: A): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq[Foo] {
  def eqv(other: Foo): Boolean =
    i == other.i && s == other.s
}
```

```
scala> Foo(5, "Hello").eqv(7)
       Foo(5, "Hello").eqv(7)
                       ^
On line 2: error: type mismatch;
        found   : Int(7)
        required: Foo
```

F-Bounded Trait

```
trait Eq[A <: Eq[A]]{
  def eqv(other: A): Boolean
}
```

```
case class Foo(i: Int, s: String) extends Eq[Foo] {
  def eqv(other: Foo): Boolean =
    i == other.i && s == other.s
}
```

```
scala> Foo(5, "Hello").eqv(7)
       Foo(5, "Hello").eqv(7)
                       ^
On line 2: error: type mismatch;
        found   : Int(7)
        required: Foo
```

```
scala> 5.eqv(7)
       5.eqv(7)
         ^
On line 2: error: value eqv is not a member of Int
```

Overloading

```
def eqv(x: Int   , y: Int   ): Boolean = x == y
def eqv(x: String, y: String): Boolean = x == y
def eqv(x: Foo   , y: Foo   ): Boolean = eqv(x.i, x.i) && eqv(x.s, y.s)
```

Overloading

```
def eqv(x: Int   , y: Int   ): Boolean = x == y
def eqv(x: String, y: String): Boolean = x == y
def eqv(x: Foo   , y: Foo   ): Boolean = eqv(x.i, x.i) && eqv(x.s, y.s)
```

```
def contains[A](xs: List[A], value: A): Boolean = ???
```

Overloading

```
def eqv(x: Int   , y: Int   ): Boolean = x == y
def eqv(x: String, y: String): Boolean = x == y
def eqv(x: Foo   , y: Foo   ): Boolean = eqv(x.i, x.i) && eqv(x.s, y.s)
```

```
def contains[A](xs: List[A], value: A)(compare: (A, A) => Boolean): Boolean =
  xs.foldRight(false)((x, acc) => compare(x, value) || acc)
```

Overloading

```
def eqv(x: Int   , y: Int   ): Boolean = x == y
def eqv(x: String, y: String): Boolean = x == y
def eqv(x: Foo   , y: Foo   ): Boolean = eqv(x.i, x.i) && eqv(x.s, y.s)
```

```
def contains[A](xs: List[A], value: A)(compare: (A, A) => Boolean): Boolean =
  xs.foldRight(false)((x, acc) => compare(x, value) || acc)
```

```
scala> contains(List(1,2,3,4,5), 6)(eqv)
res19: Boolean = false

scala> contains(List("hello","world"), "world")(eqv)
res20: Boolean = true
```

```
scala> contains(List(1,2,3,4,5), "hello")(eqv)
       contains(List(1,2,3,4,5), "hello")(eqv)
                                          ^
On line 2: error: type mismatch;
        found   : (x: Foo, y: Foo)Boolean <and> (x: String, y: String)Boolean <and> (x: Int, y: Int)Boolean
        required: (Any, Any) => Boolean
```

Overloading

```
def exact(x: Double, y: Double): Boolean =
  x == y

def approx(error: Double)(x: Double, y: Double): Boolean =
  (x - y).abs < error
```

Overloading

```
def exact(x: Double, y: Double): Boolean =
  x == y

def approx(error: Double)(x: Double, y: Double): Boolean =
  (x - y).abs < error
```

```
scala> val xs = List(1.0, 2.5, 3.3)
xs: List[Double] = List(1.0, 2.5, 3.3)
```

```
scala> contains(xs, 3.2)(exact)
res22: Boolean = false

scala> contains(xs, 3.2)(approx(0.2))
res23: Boolean = true

scala> contains(xs, 3.2)(approx(0.001))
res24: Boolean = false
```

Interface

```
trait Eq[A]{
  def eqv(x: A, y: A): Boolean
}
```

```
val fooEq: Eq[Foo] = new Eq[Foo]{
  def eqv(x: Foo, y: Foo): Boolean = x.i == y.i && x.s == y.s
}

val intEq: Eq[Int] = new Eq[Int]{
  def eqv(x: Int, y: Int): Boolean = x == y
}
```

Interface

```
trait Eq[A]{
  def eqv(x: A, y: A): Boolean
}
```

```
val fooEq: Eq[Foo] = new Eq[Foo]{
  def eqv(x: Foo, y: Foo): Boolean = x.i == y.i && x.s == y.s
}

val intEq: Eq[Int] = new Eq[Int]{
  def eqv(x: Int, y: Int): Boolean = x == y
}
```

```
scala> fooEq.eqv(Foo(1, "Hello"), Foo(5, "Hi"))
res25: Boolean = false

scala> intEq.eqv(5, 5)
res26: Boolean = true
```

Interface

```
trait Eq[A]{
  def eqv(x: A, y: A): Boolean
}
```

```
val fooEq: Eq[Foo] = new Eq[Foo]{
  def eqv(x: Foo, y: Foo): Boolean = x.i == y.i && x.s == y.s
}

val intEq: Eq[Int] = new Eq[Int]{
  def eqv(x: Int, y: Int): Boolean = x == y
}
```

```
scala> fooEq.eqv(Foo(1, "Hello"), 3)
       fooEq.eqv(Foo(1, "Hello"), 3)
                                  ^
On line 2: error: type mismatch;
        found   : Int(3)
        required: Foo
```

Interface

```
def contains[A](xs: List[A], value: A)(eq: Eq[A]): Boolean =
  xs.foldRight(false)((x, acc) => eq.eqv(x, value) || acc)

val exact: Eq[Double] = new Eq[Double] {
  def eqv(x: Double, y: Double): Boolean = x == y
}

def approx(error: Double): Eq[Double] = new Eq[Double] {
  def eqv(x: Double, y: Double): Boolean = (x - y).abs < error
}

val xs = List(1.0, 2.5, 3.3)
```

```
scala> contains(xs, 3.2)(exact)
res28: Boolean = false

scala> contains(xs, 3.2)(approx(0.2))
res29: Boolean = true

scala> contains(xs, 3.2)(approx(0.001))
res30: Boolean = false
```

Interface

```
trait Eq[A] {
  def eqv (x: A, y: A): Boolean
  def neqv(x: A, y: A): Boolean = !eqv(x, y)
}
```

Interface

```
trait Eq[A] {
  def eqv (x: A, y: A): Boolean
  def neqv(x: A, y: A): Boolean = !eqv(x, y)
}
```

```
trait Hash[A] extends Eq[A] {
  def hash(a: A): Int
}

val fooHash: Hash[Foo] = new Hash[Foo]{
  def eqv(x: Foo, y: Foo): Boolean = x.i == y.i && x.s == y.s
  def hash(a: Foo): Int = a.hashCode
}

val intHash: Hash[Int] = new Hash[Int]{
  def eqv(x: Int, y: Int): Boolean = x == y
  def hash(a: Int): Int = a
}
```

Interface or record of functions

```
case class Hash[A](
  eqv: (A, A) => Boolean,
  hash: A => Int
)

val hashFoo: Hash[Foo] = Hash[Foo](
  eqv  = (x, y) => x.i == y.i && x.s == y.s,
  hash = _.hashCode
)

val intHash: Hash[Int] = Hash[Int](
  eqv  = _ == _,
  hash = identity
)
```

Interface or record of functions

```scala
trait Eq[A] {
  def eqv (x: A, y: A): Boolean
  def neqv(x: A, y: A): Boolean = !eqv(x, y)
}

trait Hash[A] extends Eq[A] {
  def hash(a: A): Int
}

val intHash: Hash[Int] = new Hash[Int]{
  def eqv(x: Int, y: Int): Boolean = x == y
  def hash(a: Int): Int = a
}

def hashToEq[A](hash: Hash[A]): Eq[A] = hash

def reverseHash[A](current: Hash[A]): Hash[A] =
  new Hash[A]{
    def eqv(x: A, y: A): Boolean = current.eqv(x, y)
    def hash(a: A): Int = - current.hash(a)
  }
```

```scala
case class Eq[A](
  eqv: (A, A) => Boolean
)

case class Hash[A](
  eqv: (A, A) => Boolean,
  hash: A => Int
)

val intHash: Hash[Int] = Hash[Int](
  eqv  = _ == _,
  hash = identity
)

def hashToEq[A](hash: Hash[A]): Eq[A] = Eq(hash.eqv)

def reverseHash[A](current: Hash[A]): Hash[A] =
  current.copy(
    hash = (x: A) => - current.hash(x)
  )
```

Typeclass

```
trait Eq[A]{
  def eqv(x: A, y: A): Boolean
}
```

```
implicit val intEq: Eq[Int] = new Eq[Int]{
  def eqv(x: Int, y: Int): Boolean = x == y
}

implicit val stringEq: Eq[String] = new Eq[String]{
  def eqv(x: String, y: String): Boolean = x == y
}

implicit val fooEq: Eq[Foo] = new Eq[Foo]{
  def eqv(x: Foo, y: Foo): Boolean =
    implicitly[Eq[Int]].eqv(x.i, y.i) && implicitly[Eq[String]].eqv(x.s, y.s)
}
```

Typeclass: Summoning

```
object Global {
  val eqDictionary: Map[x: Type, Eq[x]] := Map(
    Int    -> intEq,
    String -> stringEq,
    Foo    -> fooEq
  )
}
```

Typeclass: Summoning

```
object Global {
  val eqDictionary: Map[x: Type, Eq[x]] := Map(
    Int    -> intEq,
    String -> stringEq,
    Foo    -> fooEq
  )
}
```

```
scala> implicitly[Eq[Int]].eqv(5, 6)
res31: Boolean = false

scala> implicitly[Eq[String]].eqv("Hello", "Hello")
res32: Boolean = true
```

```
scala> implicitly[Eq[Double]].eqv(5.5, 5.6)
       implicitly[Eq[Double]].eqv(5.5, 5.6)
                 ^
On line 2: error: could not find implicit value for parameter e: Eq[Double]
```

Typeclass: Constrain

```
trait Eq[A]{
  def eqv(x: A, y: A): Boolean
}
```

```
def contains[A](xs: List[A], value: A)(implicit ev: Eq[A]): Boolean =
  xs.foldRight(false)((x, acc) => ev.eqv(x, value) || acc)
```

Typeclass: Constrain

```
trait Eq[A]{
  def eqv(x: A, y: A): Boolean
}
```

```
def contains[A](xs: List[A], value: A)(implicit ev: Eq[A]): Boolean =
  xs.foldRight(false)((x, acc) => ev.eqv(x, value) || acc)
```

```
scala> contains(List(1,2,3,4,5), 4)
res34: Boolean = true

scala> contains(List("hello","world"), "world")
res35: Boolean = true

scala> contains(List("hello","world"), "foo")
res36: Boolean = false
```

```
scala> contains(List(1.0, 2.5, 3.3), 3.2)
       contains(List(1.0, 2.5, 3.3), 3.2)
                ^
On line 2: error: could not find implicit value for parameter ev: Eq[Double]
```

Syntactic sugar

Syntactic sugar: Summoning implicit

```
object Eq {
  def apply[A](implicit ev: Eq[A]): Eq[A] = ev
}
```

```
scala> implicitly[Eq[Int]]
res38: Eq[Int] = $anon$1@1aeb0e23

scala> Eq[Int]
res39: Eq[Int] = $anon$1@1aeb0e23
```

```
scala> Eq[Int].eqv(5, 5)
res40: Boolean = true
```

Syntactic sugar: Context bound

```scala
def contains[A](xs: List[A], value: A)(implicit ev: Eq[A]): Boolean =
  xs.foldRight(false)((x, acc) => ev.eqv(x, value) || acc)

def contains[A: Eq](xs: List[A], value: A): Boolean =
  xs.foldRight(false)((x, acc) => Eq[A].eqv(x, value) || acc)
```

Syntactic sugar: Extension methods

```
implicit class EqSyntax[A](self: A){
  def eqv(other: A)(implicit ev: Eq[A]): Boolean =
     ev.eqv(self, other)

  def ===(other: A)(implicit ev: Eq[A]): Boolean =
     eqv(other)
}
```

Syntactic sugar: Extension methods

```
implicit class EqSyntax[A](self: A){
  def eqv(other: A)(implicit ev: Eq[A]): Boolean =
     ev.eqv(self, other)

  def ===(other: A)(implicit ev: Eq[A]): Boolean =
     eqv(other)
}
```

```
scala> 5.eqv(5)
res41: Boolean = true

scala> 5 === 6
res42: Boolean = false

scala> "foo" === "hello"
res43: Boolean = false
```

```
implicit val fooEq: Eq[Foo] = new Eq[Foo]{
  def eqv(x: Foo, y: Foo): Boolean =
    x.i === y.i && x.s === y.s
}
```

	Mixin			Values		
Features	**Top level**	**Trait**	**F-Bounded**	**Overload**	**Interface**	**Typeclass**
Define comparable type	✅	✅	✅	✅	✅	✅
Same type equality	✅	✅	✅	✅	✅	✅
Support foreign type	✅	✅	✅	✅	✅	✅
Customizable / Unique	U	U	U	C	C	U
Bundle + Hierarchy	✅	✅	✅	✅	✅	✅
Code Generation	✅	✅	✅	✅	✅	✅

Folding

Folding

```
def monoFoldLeft[A](fa: List[A])(initial: A)(f: (A, A) => A): A =
  fa match {
    case Nil     => initial
    case x :: xs => monoFoldLeft(xs)(f(initial, x))(f)
  }
```

Folding

```
def monoFoldLeft[A](fa: List[A])(initial: A)(f: (A, A) => A): A =
  fa match {
    case Nil     => initial
    case x :: xs => monoFoldLeft(xs)(f(initial, x))(f)
  }
```

```
def monoFoldLeft[A](fa: List[A])(initial: A)(f: (A, A) => A): A = {
  var res: A = initial
  val it = fa.iterator
  while(it.hasNext) res = f(res, it.next())
  res
}
```

Folding

```
scala> monoFoldLeft(List(1, 2, 3, 4, 5))(0)(_ + _)
res44: Int = 15
```

Folding

```
scala> monoFoldLeft(List(1, 2, 3, 4, 5))(0)(_ + _)
res44: Int = 15
```

```
scala> monoFoldLeft(List("foo", "bar"))("")(_ + _)
res45: String = foobar

scala> monoFoldLeft(List(List(1,2,3), List(4,5)))(Nil)(_ ++ _)
res46: List[Int] = List(1, 2, 3, 4, 5)
```

Folding

```
scala> monoFoldLeft(List(1, 2, 3, 4, 5))(0)(_ + _)
res44: Int = 15
```

```
scala> monoFoldLeft(List("foo", "bar"))("")(_ + _)
res45: String = foobar

scala> monoFoldLeft(List(List(1,2,3), List(4,5)))(Nil)(_ ++ _)
res46: List[Int] = List(1, 2, 3, 4, 5)
```

```
val inc   : Int => Int = _ + 1
val double: Int => Int = _ * 2

val func = monoFoldLeft(List(inc, double))(identity)(_ andThen _)
```

```
scala> func(5)
res47: Int = 12
```

Monoid

```
trait Monoid[A] {

  def combine(x: A, y: A): A

  def empty: A

}
```

```
def monoFoldLeft[A](fa: List[A])(implicit ev: Monoid[A]): A = {
  var res: A = ev.empty
  val it = fa.iterator
  while(it.hasNext) res = ev.combine(res, it.next())
  res
}
```

Monoid instances

```
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def combine(x: Int, y: Int): Int = x + y
  def empty: Int = 0
}

implicit val stringMonoid: Monoid[String] = new Monoid[String] {
  def combine(x: String, y: String): String = x + y
  def empty: String = ""
}
```

```
scala> monoFoldLeft(List(1, 2, 3, 4, 5))
res48: Int = 15

scala> monoFoldLeft(List("foo", "bar"))
res49: String = foobar
```

Monoid instances

```
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def combine(x: Int, y: Int): Int = x + y
  def empty: Int = 0
}

implicit val stringMonoid: Monoid[String] = new Monoid[String] {
  def combine(x: String, y: String): String = x + y
  def empty: String = ""
}
```

```
scala> monoFoldLeft(List(1, 2, 3, 4, 5))
res48: Int = 15

scala> monoFoldLeft(List("foo", "bar"))
res49: String = foobar
```

```
scala> monoFoldLeft(List(true, true, false))
       monoFoldLeft(List(true, true, false))
                    ^
On line 2: error: could not find implicit value for parameter ev: Monoid[Boolean]
```

Exercises 1 (up to 1h)

Semigroup

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

Semigroup

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

```
import cats.data.NonEmptyList

def fold[A: Monoid](fa: List[A]): A = ???

def reduce[A: Semigroup](fa: NonEmptyList[A]): A = ???
```

Exercises 1 and 2

Uniqueness

```
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def combine(x: Int, y: Int): Int = x + y
  def empty: Int = 0
}
```

```
scala> intMonoid
res51: Monoid[Int] = $anon$1@3328025d

scala> implicitly[Monoid[Int]]
res52: Monoid[Int] = $anon$1@3328025d
```

Uniqueness

```
implicit val intMonoid2: Monoid[Int] = new Monoid[Int]{
  def combine(x: Int, y: Int): Int = (x + y).abs
  def empty: Int = 0
}

implicit val intMonoid3: Monoid[Int] = new Monoid[Int]{
  def combine(x: Int, y: Int): Int = x
  def empty: Int = 99
}
```

Uniqueness

```
implicit val intMonoid2: Monoid[Int] = new Monoid[Int]{
  def combine(x: Int, y: Int): Int = (x + y).abs
  def empty: Int = 0
}

implicit val intMonoid3: Monoid[Int] = new Monoid[Int]{
  def combine(x: Int, y: Int): Int = x
  def empty: Int = 99
}
```

```
scala> implicitly[Monoid[Int]]
       implicitly[Monoid[Int]]
                 ^
On line 2: error: ambiguous implicit values:
        both value intMonoid of type => Monoid[Int]
        and value intMonoid2 of type => Monoid[Int]
        match expected type Monoid[Int]
```

Uniqueness

```
implicit val intMonoid2: Monoid[Int] = new Monoid[Int]{
  def combine(x: Int, y: Int): Int = (x + y).abs
  def empty: Int = 0
}

implicit val intMonoid3: Monoid[Int] = new Monoid[Int]{
  def combine(x: Int, y: Int): Int = x
  def empty: Int = 99
}
```

```
scala> fold(List(1,2,3,4))
       fold(List(1,2,3,4))
           ^
On line 2: error: ambiguous implicit values:
        both value intMonoid of type => Monoid[Int]
        and value intMonoid2 of type => Monoid[Int]
        match expected type Monoid[Int]
```

How to ensure uniqueness?

1. Parametricity

```
def map[F[_], A, B](fa: F[A])(f: A => B): F[B]
```

2. Laws

```
"Monoid: combine is associative" in {
  forAll((x: A, y: A, z: A) => ((x |+| y) |+| z) === (x |+| (y |+| z)))
}
```

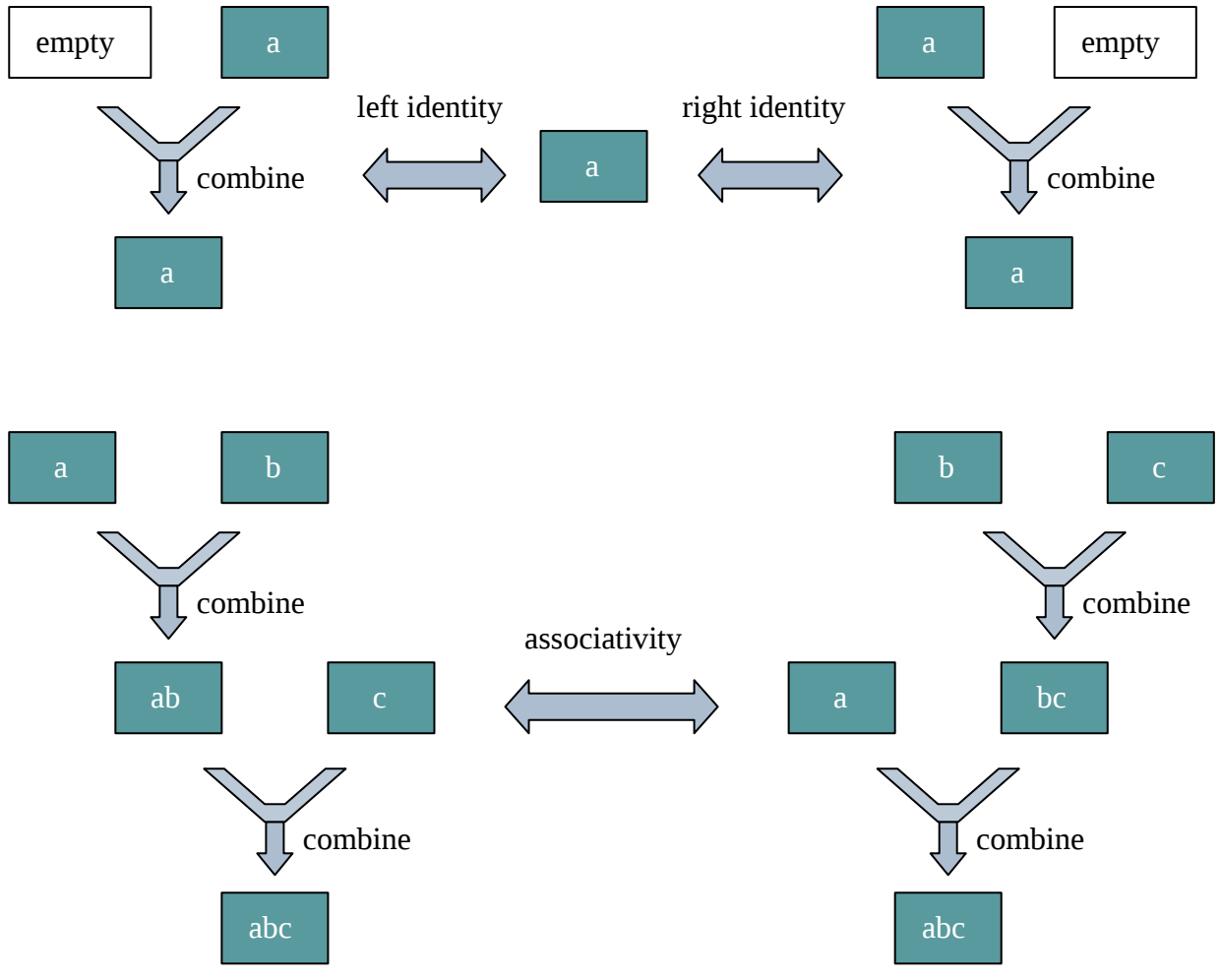
Monoid Laws

```
- Double.Monoid.associative *** FAILED ***
   GeneratorDrivenPropertyCheckFailedException was thrown during property evaluation.
    (Discipline.scala:14)
     Falsified after 8 successful property evaluations.
     Location: (Discipline.scala:14)
     Occurred when passed generated values (
       arg0 = -4.498344780314773E208,
       arg1 = -5.116678570398714E199,
       arg2 = -1.0056301257545645E214
     )
```

Failed after two weeks of CI build

Exercises 3

Monoid Laws


empty
a
combine
a
left identity
a
right identity
a
empty
combine
a
a
b
combine
ab
c
combine
abc
associativity
b
c
combine
a
bc
combine
abc
abc
value of type A
function
equivalence

Use case: Distributed Map-Reduce

```
class RDD[A]{

  def map[B](f: A => B): RDD[B]

  def reduce(f: (A, A) => A): A

}
```

Spark Accumulators (aka reduce)

Accumulators are variables that are only “added” to through an **associative** and **commutative** operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums.

Exercises 4

Where to define instances?

Where to define instances?

1. Companion object of the target type

```
case class Metrics(ordersCreated: Long, ordersDeleted: Long)

object Metrics {
  implicit val Monoid: Monoid[Metrics] = new Monoid[Metrics]{
    def combine(x: Metrics, y: Metrics): Metrics =
      Metrics(
        ordersCreated = x.ordersCreated + y.ordersCreated,
        ordersDeleted = x.ordersDeleted + y.ordersDeleted
      )

    def empty: Metrics =
      Metrics(0, 0)
  }
}
```

Where to define instances?

2. Companion object of typeclass

```
object Monoid {
  implicit val int: Monoid[Int] = new Monoid[Int] {
    def combine(x: Int, y: Int): Int = x + y
    def empty: Int = 0
  }

  implicit val string: Monoid[String] = new Monoid[String] {
    def combine(x: String, y: String): String = x + y
    def empty: String = ""
  }
}
```

Where to define instances?

3. Ad-hoc object (to avoid as much as possible!)

```
object MonoidInstance {
  implicit val booleanMonoid: Monoid[Boolean] = new Monoid[Boolean]{
    def combine(x: Boolean, y: Boolean): Boolean = x && y
    def empty: Boolean = true
  }
}
```

Where to define instances?

3. Ad-hoc object (to avoid as much as possible!)

```
object MonoidInstance {
  implicit val booleanMonoid: Monoid[Boolean] = new Monoid[Boolean]{
    def combine(x: Boolean, y: Boolean): Boolean = x && y
    def empty: Boolean = true
  }
}
```

```
scala> fold(List(true, false, false))
       fold(List(true, false, false))
            ^
On line 2: error: could not find implicit value for parameter ev: Monoid[Boolean]
```

Where to define instances?

3. Ad-hoc object (to avoid as much as possible!)

```
object MonoidInstance {
  implicit val booleanMonoid: Monoid[Boolean] = new Monoid[Boolean]{
    def combine(x: Boolean, y: Boolean): Boolean = x && y
    def empty: Boolean = true
  }
}
```

```
scala> fold(List(true, false, false))
       fold(List(true, false, false))
            ^
On line 2: error: could not find implicit value for parameter ev: Monoid[Boolean]
```

```
import MonoidInstance._
```

```
scala> fold(List(true, false, false))
res56: Boolean = false
```

Where to define instances?

3. Ad hoc object (to avoid as much as possible!)

```
object OtherMonoidInstance {
  implicit val booleanMonoid: Monoid[Boolean] = new Monoid[Boolean]{
    def combine(x: Boolean, y: Boolean): Boolean = x || y
    def empty: Boolean = false
  }
}
```

Where to define instances?

3. Ad hoc object (to avoid as much as possible!)

```
object OtherMonoidInstance {
  implicit val booleanMonoid: Monoid[Boolean] = new Monoid[Boolean]{
    def combine(x: Boolean, y: Boolean): Boolean = x || y
    def empty: Boolean = false
  }
}
```

```
import MonoidInstance._
```

```
scala> fold(List(true, false, false))
res57: Boolean = false
```

```
import OtherMonoidInstance._
```

```
scala> fold(List(true, false, false))
res58: Boolean = true
```

When should I use ad hoc object for typeclass instances

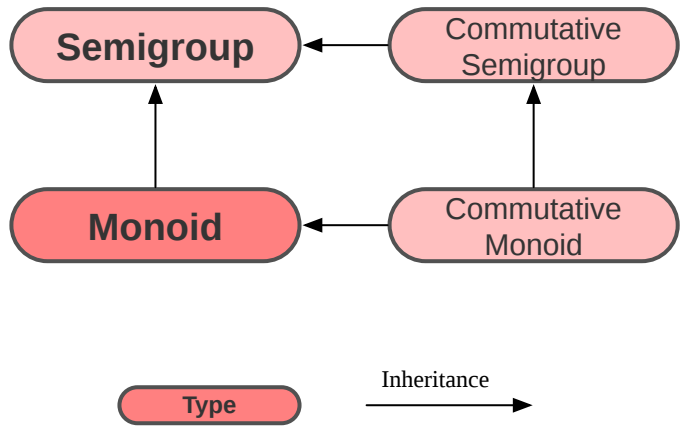
- In libraries when you have a typeclass hierarchy

```
trait Semigroup[A]

object Semigroup {
  implicit val int: Semigroup[Int] = ...
}

trait Monoid[A] extends Monoid[A]

object Monoid {
  implicit val int: Monoid[Int] = ...
}

trait CommutativeMonoid[A] extends Monoid[A]

object CommutativeMonoid {
  implicit val int: CommutativeMonoid[Int] = ...
}
```

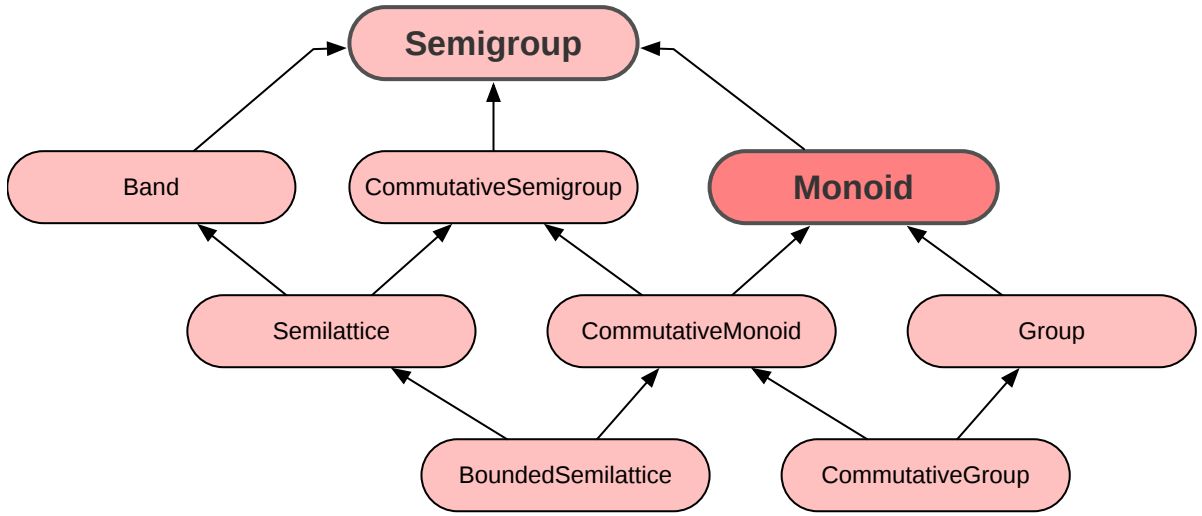
- If you control neither the typeclass or the type (Please make a PR!)

Exercises 5

Semigroups


Semigroup
Commutative Semigroup
Monoid
Commutative Monoid
Type
Inheritance

Semigroups in Cats



Kinds

Kinds

```
:k Int
Int :: Type
```

Kinds

```
:k Int
Int :: Type
```

```
:k String
String :: Type
```

Kinds

```
:k Int
Int :: Type
```

```
:k String
String :: Type
```

```
:k List
List :: Type -> Type
```

Kinds

```
:k Int
Int :: Type
```

```
:k String
String :: Type
```

```
:k List
List :: Type -> Type
```

```
:k List[Int]
List[Int] :: Type
```

Kinds

```
:k Int
Int :: Type
```

```
:k String
String :: Type
```

```
:k List
List :: Type -> Type
```

```
:k List[Int]
List[Int] :: Type
```

```
:k Either
Either :: Type -> Type -> Type
```

Kinds

```
:k Int
Int :: Type
```

```
:k String
String :: Type
```

```
:k List
List :: Type -> Type
```

```
:k List[Int]
List[Int] :: Type
```

```
:k Either
Either :: Type -> Type -> Type
```

```
:k Either[String, ?]
Either[String, ?] :: Type -> Type
```

Kinds

```
trait Semigroup[A]{
  def combine(x: A, y: A): A
}

implicit val intSemigroup: Semigroup[Int] = new Semigroup[Int]{
  def combine(x: Int, y: Int): Int = ???
}
```

Kinds

```
trait Semigroup[A]{
  def combine(x: A, y: A): A
}

implicit val intSemigroup: Semigroup[Int] = new Semigroup[Int]{
  def combine(x: Int, y: Int): Int = ???
}
```

```
scala> implicit val listSemigroup: Semigroup[List] = ???
       implicit val listSemigroup: Semigroup[List] = ???
                                               ^
On line 2: error: type List takes type parameters
```

```
implicit def listSemigroup[A]: Semigroup[List[A]] = new Semigroup[List[A]]{
  def combine(x: List[A], y: List[A]): List[A] = ???
}
```

Foldable

```
trait Foldable[F[_]] {
  def foldLeft[A, B](fa: F[A], z: B)(f: (B, A) => B): B
  def foldRight[A, B](fa: F[A], z: B)(f: (A, => B) => B): B
}
```

Foldable

```
trait Foldable[F[_]] {

  def foldLeft[A, B](fa: F[A], z: B)(f: (B, A) => B): B

  def foldRight[A, B](fa: F[A], z: B)(f: (A, => B) => B): B

}
```

```
implicit val listFoldable: Foldable[List] = new Foldable[List] {
  def foldLeft[A, B](fa: List[A], z: B)(f: (B, A) => B): B = ???
  def foldRight[A, B](fa: List[A], z: B)(f: (A, => B) => B): B = ???
}
```

Foldable

```
trait Foldable[F[_]] {

  def foldLeft[A, B](fa: F[A], z: B)(f: (B, A) => B): B

  def foldRight[A, B](fa: F[A], z: B)(f: (A, => B) => B): B

}
```

```
implicit val listFoldable: Foldable[List] = new Foldable[List] {
  def foldLeft[A, B](fa: List[A], z: B)(f: (B, A) => B): B = ???
  def foldRight[A, B](fa: List[A], z: B)(f: (A, => B) => B): B = ???
}
```

```
scala> implicit val intFoldable: Foldable[Int] = ???
       implicit val intFoldable: Foldable[Int] = ???
                                          ^
On line 2: error: Int takes no type parameters, expected: one
```

Exercises 6

Use Typeclass when

1. You have only one valid implementation for (almost) all types
2. You require uniqueness per type e.g. Eq, Hash, Ord

Use Interface / Record of functions when

1. You have many valid / interesting implementations per type
2. You need to package several functions together

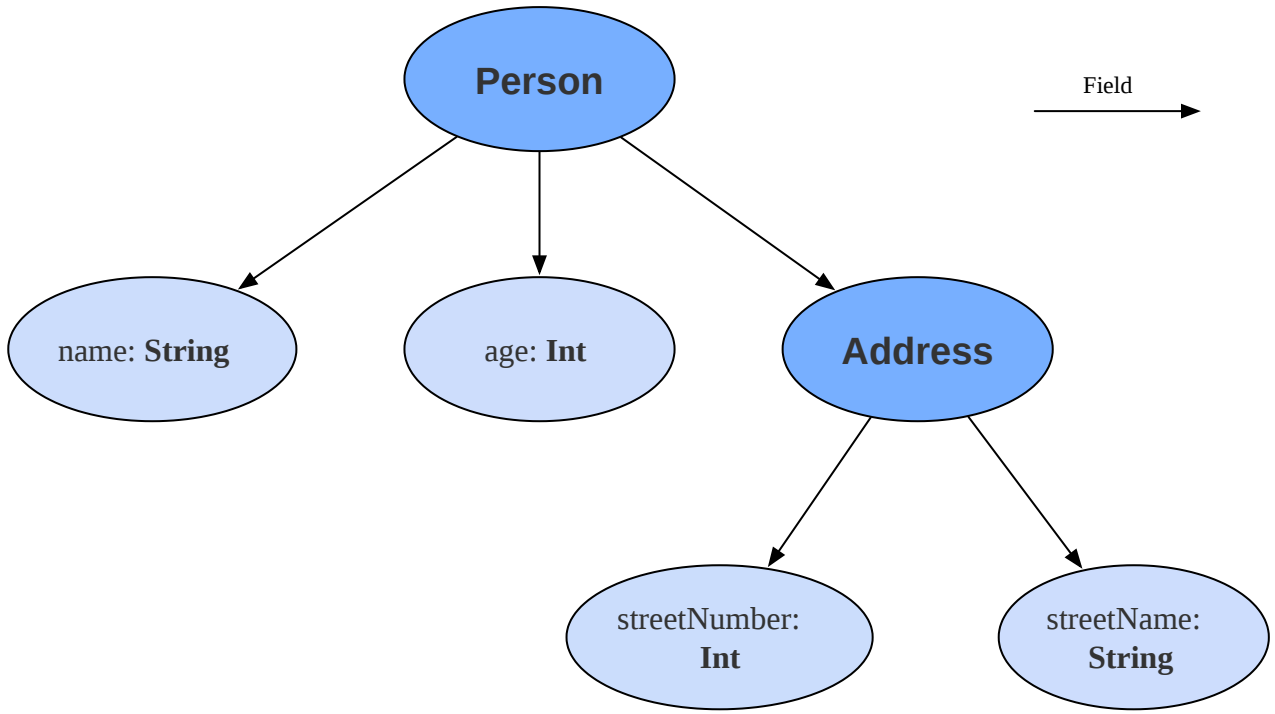
Use Overloading when

1. You have many valid / interesting implementations per type
2. Single function

Two reasons why typeclasses are overused

1. Typeclass Derivation

Typeclass Derivation: Product



Typeclass Derivation: Product

```
case class Person(name: String, age: Int)
```

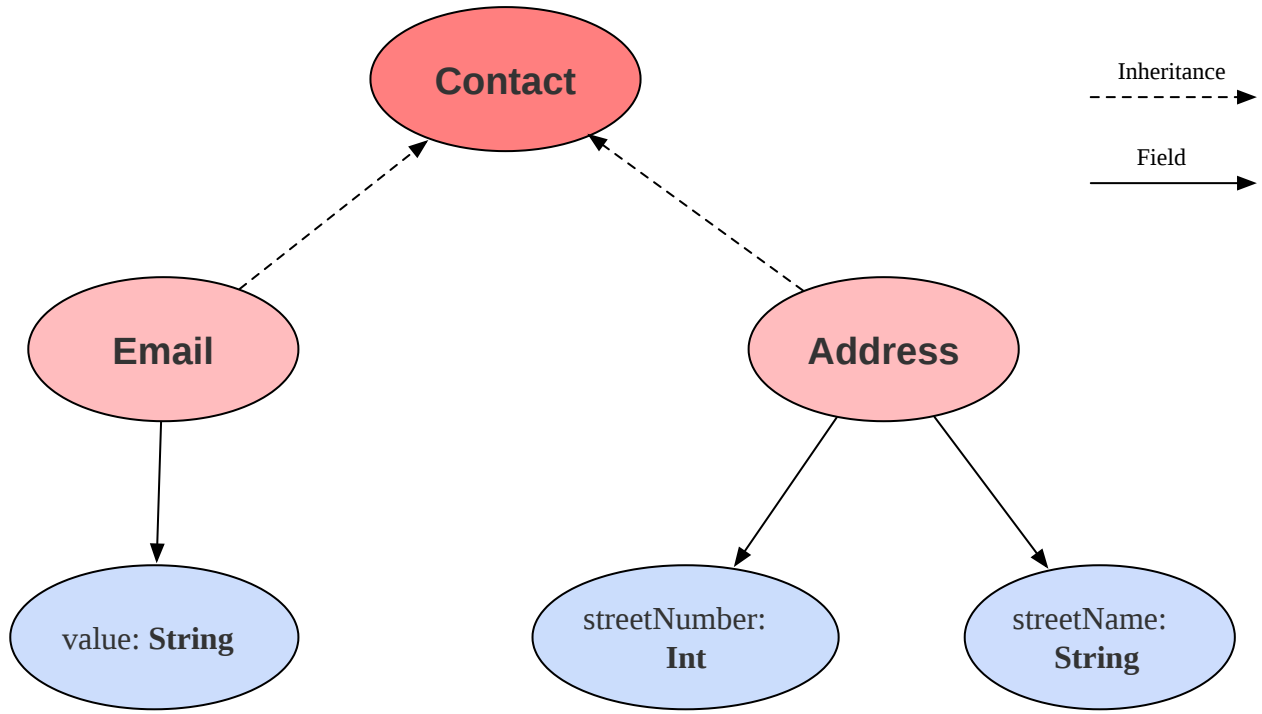
```
Person <==> (String, (Int, Unit))
```

```
implicit val intEq   : Eq[Int]    = (x: Int   , y: Int   ) => x == y
implicit val stringEq: Eq[String] = (x: String, y: String) => x == y
implicit val unitEq  : Eq[Unit]   = (x: Unit  , y: Unit  ) => true
```

```
implicit def tuple2[A: Eq, B: Eq]: Eq[(A, B)] = new Eq[(A, B)] {
  def eqv(x: (A, B), y: (A, B)): Boolean =
    x._1 === y._1 && x._2 === y._2
}
```

```
Eq[Person] => Eq[(String, (Int, Unit)]
           => tuple2(Eq[String], Eq[(Int, Unit)])
           => tuple2(Eq[String], tuple2[Int, Unit])
           => tuple2(Eq[String], tuple2(Eq[Int], Eq[Unit]))
```

Typeclass Derivation: Sum



Typeclass Derivation: Sum

```
sealed trait Contact
case class Email(value: String) extends Contact
case class Address(streetNumber: Int, StreetName: String) extends Contact
```

```
Contact <==> Either[Email, Either[Address, Nothing]]
```

```
implicit val nothingEq: Eq[Nothing] = (x: Nothing, y: Nothing) => true

implicit def either[A: Eq, B: Eq]: Eq[Either[A, B]] = new Eq[Either[A, B]] {
  def eqv(e1: Either[A, B], e2: Either[A, B]): Boolean =
    (e1, e2) match {
      case ( Left(x),  Left(y)) => x === y
      case (Right(b1), Right(b2)) => b1 === b2
      case _                      => false
    }
}
```

```
Eq[Contact] => Eq[Either[Email, Either[Address, Nothing]]]
            => either(Eq[Email], Eq[Either[Address, Nothing]])
            => ...
```

2. Better syntax

Better syntax

```
"foo" === "hello"
stringEq.eqv("foo", "hello")
```

```
List("hello", "world", "!").foldMap(_.size)
foldMap(List("hello", "world", "!"))(_.size)(intMonoid)
```

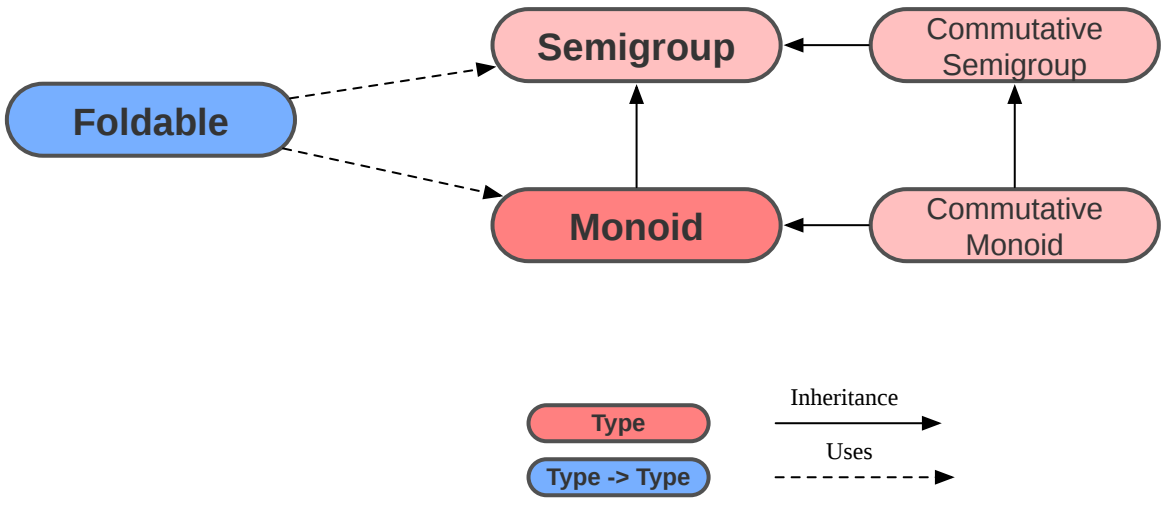
```
val json = """{ "name" : "John", "age"  : 34 }"""

json.as[Person]
Person.restJsonDecoder.parse(json)
```

Future

```
List(1,2,3,4,5).fold (@via Sum)     // 15
List(1,2,3,4,5).fold (@via Product) // 120
```

See @Iceland_jack work on #DerivingVia for Haskell

Review


Semigroup
Commutative Semigroup
Foldable
Monoid
Commutative Monoid
Type
Type -> Type
Inheritance
Uses

Module 7: Functors