scalaz

zee is for zed, some of the time

What is scalaz?

- scalaz is a library of typeclasses and datatypes for functional programming
  - Typeclasses are like post-hoc interfaces; they give extra functionality to other datatypes
  - In other words, they enrich (used to be called pimp) existing types with new behavior
- Current version is version 7, a major redesign from version 6 and prior
  - Version 6 and prior kinda sucked
- Downloadable at `http://github.com/scalaz/scalaz` branch `scalaz-seven`

What is scalaz? (cont.)

- Provides an ontology for existing widely used types, such as `List` or `Option`
- Provides new types which also fit into the ontology that have behavior not in the standard library
- Words of warning:
  - Very deep, and in many cases daunting
  - New design in version 7 helps with this quite a lot
  - Very powerful
  - Use with care

Why scalaz?

- Combining proven patterns yields safer code
- Using patterns rooted in category theory and encoded in the type system makes many classes of error a compile time error
- Can make thorny problems more tractable and more readable
  - Assuming one makes it up the learning curve, of course
  - Good example is MICROS check building in the PXC, which uses the `State` monad to good effect
- Deeper understanding of the rigorous underpinnings of common types (e.g. `List`, or imperative code) improves one's ability to program, and particularly to reason about program behavior.

Getting Started

- Download source code so you can build docs, since they are not conveniently available on the web:

```
    bash> git clone git://github.com/scalaz/
scalaz.git

    bash> cd scalaz; ./sbt

    sbt> update

    sbt> doc
```

- Spawn a REPL to play around:

```
    bash> cd scalaz; ./sbt

    sbt> project core

    sbt> console
```

What's in there?

- Three major sections:
  - Typeclasses and typeclass instances
    - E.g. `Semigroup`, `Apply`, `Monad`
  - Datatypes and typeclass instances
    - E.g. `Kleisli`, `Lens`, `State`
  - Syntax for typeclasses
    - E.g. `m >>= f`, `(m1 |@| m2 |@| m3) (f)`

Typeclasses?

- Extensions of some type which give additional behavior based on some primitive operations
- For example, given `bind` (`flatMap`) and `point`, the `Monad` typeclass gives you `applyN` (`apply2`, `apply3`, etc):

```
val m1 = Some(1); val m2 = Some(2); val m3 = Some(3)

apply3(m1, m2, m3) ( _ + _ + _ )

    == Some(6)

(for (a1 <- m1; a2 <- m2; a3 <- m3) yield a1 + a2 + a3)

    == Some(6)

m1 flatMap { a1 => m2 flatMap { a2 => m3 flatMap { a3 =>

    Some(a1+a2+a3)

} } } == Some(6)
```

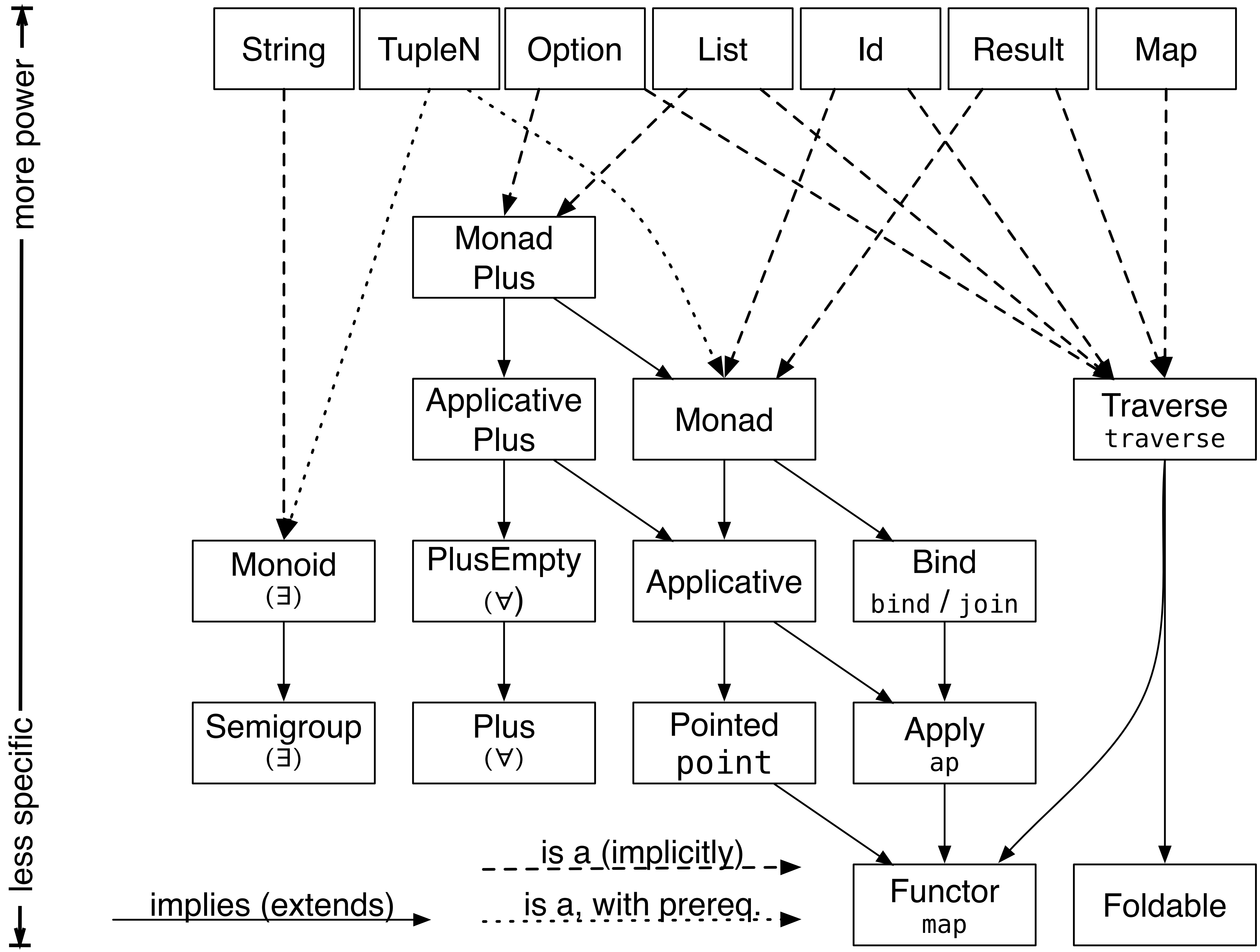
Typeclasses in scalaz

- Typeclasses are instantiated for particular types, not values, using implicits
  - For example, `scalaz.std.option.optionInstance` is an implicit value giving instances of `Traverse`, `MonadPlus`, etc. for `Option`.
- Typeclasses are encoded as a trait which may extend other traits to reify implication
  - For example, trait `MonadPlus` extends `Monad` which extends `Applicative`, etc.
- Typeclasses typically have an associated companion object which lets you easily obtain an instance of the typeclass implicitly
  - For example, `Apply[Option]` gives the `Apply` instance for `Option`. It's equivalent to `implicitly[Apply[Option]]`

Typeclasses in scalaz

- Typeclasses imply (extend) other typeclasses
  - For example, a `Monad` implies an `Applicative` which in turn implies `Apply`, which in turn implies `Functor`
- In this way, typeclasses form an abstraction tower, where stronger typeclasses (such as `Monad`) build on weaker ones (such as `Applicative`)
- Each more powerful typeclass has stronger requirements for the type being classified, so the more powerful typeclass you have the fewer types will be eligible members of it
  - For example, `MonadPlus` is strictly more powerful than `Monad`; `MonadPlus` implies `PlusEmpty` which requires that the type have some "empty" state. Because of this, `MonadPlus` is applicable to less types, because not all monadic types have an "empty" state such as `Result`. However, some do such as `Option`.

(Partial) Typeclass hierarchy in scalaz



Tour of the basic typeclasses

Onwards, to adventure!

Functor

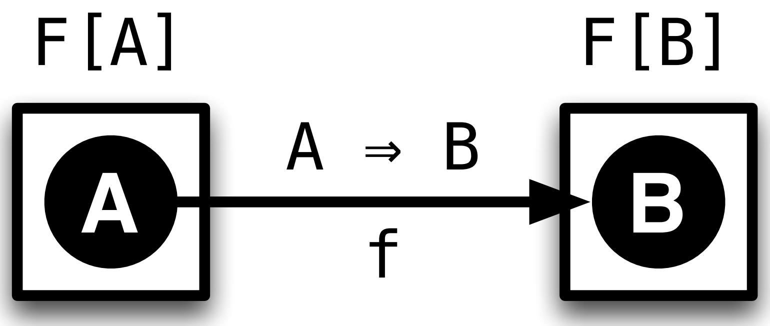
- "Things that contain things"
- `List`, `Option`, `Result`, `Id` are all functors

```
scala> import scalaz.std.list.listInstance
scala> Functor[List].map(List(1,2,3))(_ * 2)
res3: List[Int] = List(2, 4, 6)

scala> import scalaz.std.option.optionInstance
scala> Functor[Option].map(Some(123))(_ * 2)
res5: Option[Int] = Some(246)

scala> import scalaz.Id.Id
scala> Functor[Id].map(123)(_ * 2)
res8: scalaz.Id.Id[Int] = 246
```

```scala
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A ⇒ B): F[B]
}
```



- Perhaps the most basic structure, which models structures "things that contain things" with a very loose definition of contain.
- Since typeclasses are for types not values, the operations they contain are like singleton methods or static functions. So this `map` function would be used as in `Functor[List].map(list)(f)` opposed to the built-in `list.map(f)`
- The type being classified (F) is *universally quantified*, which is what the [_] indicates.
    - Universal quantification guarantees (requires) that the element type is polymorphic. For example, a `List` is a candidate `Functor` since it is polymorphic in its element type. Conversely `String` is not since it has restrictions on its element type (e.g. must always be `Char`)
- The only required operation is `map`, sometimes known as *lift* or `fmap`. This operation "lifts" a function `f` into the `Functor` `F`, transforming elements of type `A` from `F[A]` into elements of (possibly the same type) `B`, producing `F[B]`

Pointed <: Functor

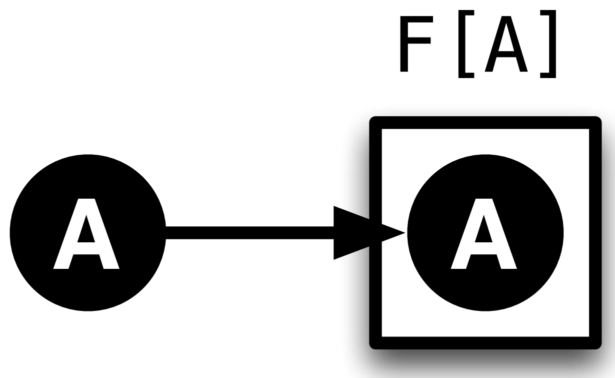
- Single function `point` "injects" a value into the functor
- `List`, `Option`, `Result`, `Id` are all pointed

```
scala> Pointed[List].point(123)
res9: List[Int] = List(123)

scala> Pointed[Option].point(123)
res10: Option[Int] = Some(123)

scala> Pointed[Id].point(123)
res11: scalaz.Id.Id[Int] = 123
```

```scala
trait Pointed[F[_]] extends Functor[F] {
    def point[A](a: => A): F[A]
}
```



- Slight (but critical) refinement on Functor that allows you to inject values into a functor.
- The only required operation is `point` (sometimes called `return`). This operation "lifts" a value `a` into the `Functor` `F`, producing `F[A]`
- There is also a corresponding `Copointed` which lets you extract values from conforming types

Apply <: Functor

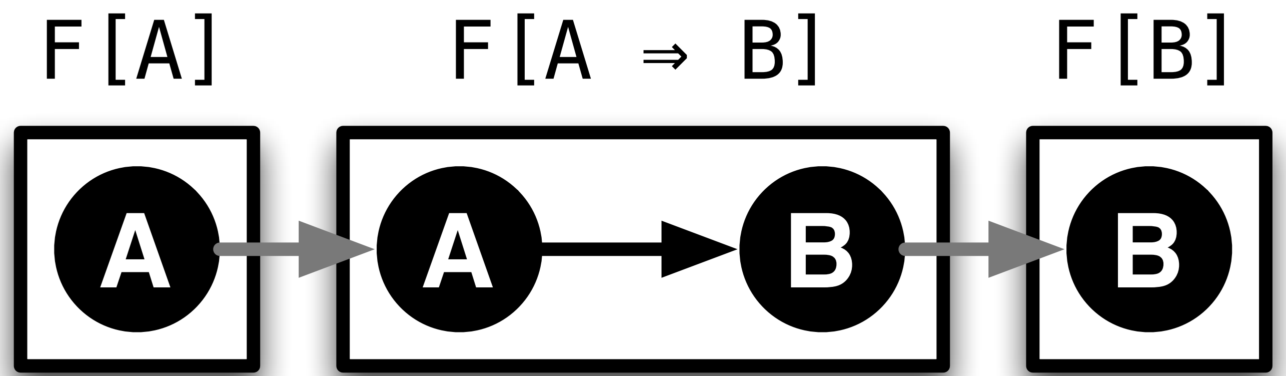
- Apply has `ap` which applies a value in a functor to a function in a functor
- `List`, `Option`, `Result`, `Id` all have instances of `Apply`

```
scala> Apply[List].ap(List(1,2,3))(
           List[Int => String](_ + "foo", _ + "bar")
       )
res13: List[String] = List(1foo, 2foo, 3foo, 1bar,
2bar, 3bar)

scala> Apply[Option].ap(Some(123))(
           Some[Int => String](_ + "foo")
       )
res15: Option[String] = Some(123foo)

scala> Apply[Id].ap(123)(_ + "foo")
res16: scalaz.Id.Id[String] = 123foo
```

```
trait Apply[F[_]] extends Functor[F] {
    def ap[A, B](fa: ⇒ F[A])(f: ⇒ F[A ⇒ B]): F[B]
}
```



- Allows application of function(s) embedded in the functor to values embedded in the functor, which lets you combine multiple functors using some function:

```
def f(s: String, t: String): String = s + t
val fc: String => String => String = (f _).curried

Apply[List].ap(List("c", "d"))(
    Apply[List].map(List("a", "b"))(fc)
) == List("ac", "ad", "bc", "bd")
```

- Or better yet, using a function on `Apply` for just this purpose:

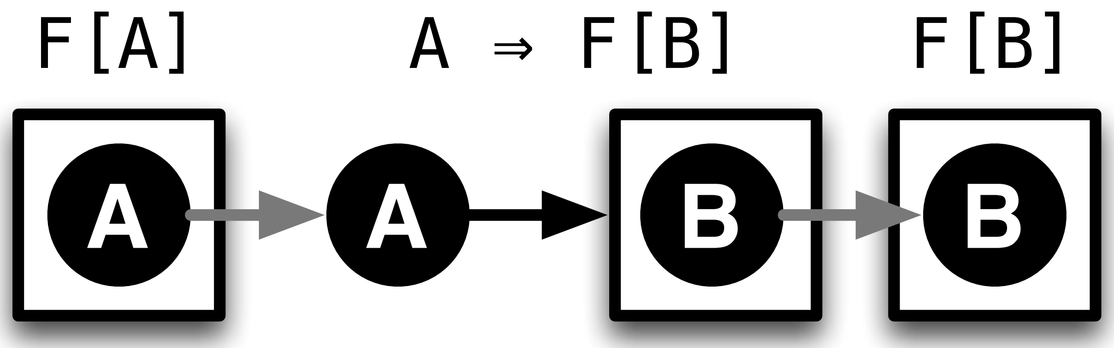
```
Apply[List].apply2(List("a", "b"), List("c", "d"))(f)
```

Bind <: Apply

- Bind has `bind` and `join` which allow for nested functor values to be collapsed (flattened)
- `join` is the essential operation which collapses nested values in theory, but `bind` is the one more used, and the one that's required to implement `Bind`. They are interchangeable using `map` implied by `Functor` (`join(x) == bind(x)(identity)`, `bind(x)(f) == join(map(x)(f))`)
- `List`, `Option`, `Result`, `Id` all have instances of `Bind`

```
Bind[List].bind(List(1,2,3))(a => List(a, a*2))
    == List(1, 2, 2, 4, 3, 6)
Bind[List].join(List(List(1,2),List(3,4)))
    == List(1, 2, 3, 4)
Bind[Option].bind(Some(1))(a => Some(a*2))
    == Some(2)
Bind[Option].join(Some(Some(1)))
    == Some(1)
Bind[Id].bind(1)(_ * 2)
    == 2
Bind[Id].join(1)
    == 1
```

```scala
trait Bind[F[_]] extends Apply[F] {
    def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```



- Allows collapsing of towers of embedded values, e.g. `F[F[A]] ⇒ F[A]` via `join` and the more commonly used `bind`, which implements sequencing
- If you've written any nontrivial Scala, you've used `bind` in it's Scala-named guise as `flatMap`
    - The word `bind` tends to have a "sequencing" feel to it, and it came from using Monads for effect sequencing in lazy evaluation
    - The word `flatMap` tends to have a "collection" feel to it, but they are equivalent
- `Bind` lets you change the "shape" of the functor with a function. For `map` (the `Functor` primitive) the shape of the structure remains the same- `List`s maintain their length, `Option`s stay a `Some` -but for `bind`/`flatMap` the `List` can grow or shrink, a `Some` can become `None`, etc.

Applicative & Monad

- `Applicative` combines `Pointed` and `Apply`. That is, it models structures which contain things (`Functor`), you can inject values into (`Pointed`), and you can apply functions within to combine multiple instances of the structure (`Apply`).
- `Monad` combines `Applicative` and `Bind`, to model structures which do all of the above but can also change their shape with `bind`/`flatMap`

Semigroup & Monoid

- Semigroup models a structure F where two values of F can be combined or added via the function append.
  - Examples are Lists (concatenation), Strings (concatenation), Ints (addition or multiplication), Options (first-wins or second-wins)
  - Does not require that information is preserved, hence why Option can be a Semigroup
  - Does require that appends is associative, but not commutative, which is why Ints can be Semigroups using addition or multiplication but not subtraction or division
- Monoid extends Semigroup with an identity value zero
  - For Lists, zero is Nil. For Strings, zero is “”. For Ints under addition, zero is 0. For ints under multiplication, zero is 1.
  - Monoids can be very useful in unusual and interesting ways beyond the usual cases, see “Monoids and Finger Trees” by Heinrich Apfelmus

Plus & PlusEmpty

- `Plus` is similar to `Semigroup`, e.g. describes some structure which can be combined using an associative operator. The difference is in the *quantification*.
  - `Semigroup` is *existentially quantified*, which translates into English as "there exists some type `F` which can be combined". Conversely, `Plus` is *universally quantified*, which translates into English as "for any type `A`, there exists some type `F[A]` which can be combined".
  - `Semigroup` (existential quantification) is encoded in Scala as:

    ```
    trait Semigroup[F] { def append(f1: F, f2: F): F }
    ```

  - `Plus` (universal quantification) is encoded in Scala as:

    ```
    trait Plus[F[_]] { def append[A](f1: F[A], f2: F[A]): F[A] }
    ```

  - `List` can be either a `Semigroup` or a `Plus` since it's polymorphic.
  - Conversely, `String` cannot have a `Plus` instance since it is monomorphic (elements are always `Char`)
- As `Plus` is to `Semigroup`, `PlusEmpty` is to `Monoid`. That is, `PlusEmpty` is a universally quantified version of `Monoid`.

  ```
  trait Monoid[F] extends Semigroup[F] { def zero: F }
  trait PlusEmpty[F[_]] extends Semigroup[F] { def empty[A]: F[A] }
  ```

ApplicativePlus & MonadPlus

- `ApplicativePlus` combines `Applicative` and `PlusEmpty`, describing structures which are `Applicative` (`Pointed`, `Apply`, `Functor`) and also monoidal and universally quantified (`PlusEmpty`).
  - Universal quantification was already guaranteed by `Functor`, by the way.
- `MonadPlus` is the same for `Monad`; it combines `ApplicativePlus` and `Monad`

Some Important Scala Features

it’s not safe to go alone. take this.

Implicits

- In Scala, a value (`val`) or function (`def`) can be marked `implicit`, e.g. `implicit val x: T = ...` or `implicit def f(a: T): U = ...`
- Implicit parameters can be added to functions. These parameters do not need to be given arguments when calling the function; if no argument is given then the compiler which searches for any implicit val or def in scope that matches the given type.
  - Implicit parameters are used extensively by scalaz for typeclasses.
- Implicit conversions are inserted by the compiler automatically where the value is of one type and a different type is expected *and* a function is implicitly available which takes the value's type and produces the required type.
- Implicit conversions are also inserted by the compiler automatically when you attempt to use a method on a type that doesn't have that method, but there is an implicit conversion available in scope which would make that method available.
  - Implicit conversions triggered by method calls are used extensively by scalaz for typeclass syntax.

Implicit examples

```scala
trait Foo[A] { def foo: String }

implicit val fooString =
    new Foo[String] { def foo = "fooString" }
implicit def fooify[A](in: A) =
    new Foo[A] { def foo = "fooified" }

def useFoo[A](a: A)(implicit foo: Foo[A]) =
    a.toString + foo.foo

// fooString implicitly passed by compiler
useFoo("abc") == "abcfooString"

// fooString explicitly passed
useFoo("abc")(fooString) == "abcfooString"

// fooify conversion inserted by compiler
"abc".foo == "fooified"

// manually inserted conversion
fooify("abc").foo == "fooified"
```

Type lambdas

- Scala doesn't have them, but you need them sometimes, particularly in scalaz where you need a `F[_]`, but have a `G[_, _]`. For example, `ResultG` is of kind `(•,•) ⇒ •` but the Functor typeclass expects `F[_]` which has kind `• ⇒ •`. To make a `Functor` instance for `ResultG[E, _]`, you'd need to "curry off" one of the two type arguments to yield `• ⇒ •`.
- Worked around the lack of language support using structural types, written as `({ type F[A] = G[..., A] })#F`
  - That is, define an anonymous structural type with one member `F` which has kind `• ⇒ •`. `F` is a type alias which applies some fixed type ("…" here) and the given `A`
  - Once that structural type is defined, project the `F` member out of it using `#F`. `#` is to types as `.` is to values.

Syntax

can I borrow a pound of sugar?

Syntax

- Syntax makes using typeclasses easier.
- Use syntax by first importing the typeclasses you'll need (regularly from scalaz.std.*type*.*type*Instance), then import the particular syntax you need from scalaz.syntax.typeclass

```
import scalaz.std.option.{optionInstance, some}
import scalaz.syntax.apply.^^

val m1 = some(1); val m2 = some(2); val m3 = some(3)

^^(m1, m2, m3) { _ + _ + _ }
    == Some(6)

(for (a <- m1; b <- m2; c <- m3) yield a + b + c)
    == Some(6)
```

Functor Syntax

- `^(fa)(f)` maps `f` over `fa`. It's equivalent to `Functor[F].map(fa)(f)`
- `fa >| b` replaces the value(s) in a functor with `b`. It's equivalent to `Functor[F].map(fa)(_ => b)`
- `f.lift` lifts `f` into a functor context. It's equivalent to `fa => Functor[F].map(fa)(f)`

Pointed Syntax

- `a.point` or `a.pure` injects `a` into a pointed. It's equivalent to `Pointed[F].point(a)`
- Boring Pointed has boring syntax

Apply Syntax

- `^(fa, fb) { _ + _ }` applies the pure function to pairs of values from `fa` and `fb`, according to the application rules of `fa` and `fb` (e.g. `List` gives cross product). It's equivalent to `Apply[F].apply2(fa, fb)(f)`
- `^^(fa, fb, fc) { f }`, `^^^(fa, fb, fc, fd) { f }` and so on also exist, corresponding to `Apply#applyN`
- `fa *> fb` combines fa and fb by discarding the values in fa. This is only useful when F embeds side effects such as success/failure. You can read it as "then", or "use the value to the right". The left-leaning variant fa <* fb is also provided.
- `fa <*> f` applies the function(s) in `f` to the value(s) in `fa`. It's equivalent to `Apply[F].ap(fa)(f)`
- `fa tuple fb` yields a tuple `F[(A, B)]` from `F[A]` and `F[B]`
- `(fa |@| fb |@| fc)(f)` is another way of writing ^^ or `Apply[F].apply2`.
- `(fa |@| fb |@| fc).tupled` is a way of constructing n-tuples from n functors. `(fa |@| fb).tupled` is equivalent to `fa tupled fb`

Bind Syntax

- `ma >>= f` binds `ma` through `f`, and is equivalent to `Bind[F].bind(ma)(f)` or `ma.flatMap(f)` but briefer. I find `>>=` reads better than `flatMap` in many cases for flow control.
- `ma >> mb` is another way of combining two `Bind`s while discarding the value in the left hand one, the other way being `Apply`'s `ma *> mb` which is available since `Bind` implies `Apply`. It's equivalent to `ma >>= { _ => mb }`
- `ma.join` is equivalent to `Bind[M].join(ma)`
- `ma.ifM(ifTrue, ifFalse)` is a way of switching consequences based on a `Boolean` inside of a `Bind`. It's equivalent to `ma >>= { b => if (b) ifTrue else ifFalse }`

Applicative Syntax

- `ma.unlessM(cond)` "executes" (evaluates and binds) ma unless the condition is `true`. It's equivalent to:

```
if (cond) Pointed[M].point(()) else ma >> ()
```

- `ma.whenM(cond)` is the converse, only binding when the condition is `true`

Monad Syntax

- `m.liftM` lifts `m` into some transformed monad `G` (monad transformers explained later). It's equivalent to `MonadTrans[G].liftM(m)`
- `m.replicateM(10)` binds together 10 copies of `m`.

Plus / Semigroup Syntax

- `pa <+> pb` concatenates `pa` and `pb`, which have instances of `Plus`. It's equivalent to `Plus[F].append(pa, pb)`
- `fa |+| fb` concatenates two monoids. It's equivalent to `Monoid[F[A]].append(fa, fb)`
- `mzero` is the zero value for a `Monoid`. It's equivalent to `Monoid[F[A]].zero`

MonadPlus Syntax

- ma.filter(p) yields ma if p yields true for the value(s) in ma, and mzero if the predicate fails. It's equivalent to:

```
ma >>= { a => if (p(a)) a.point else empty }
```

Tour of Monads and Monad Transformers

Bring in the Monads!

Monads

- The monads we'll talk about are datatypes, not typeclasses like we've been talking about. They also have instances of the `Monad`/ `MonadPlus` typeclass.
- Monads model composable control flow, such as keeping state (`State`), providing some readable context (`Reader`), computations that can fail (`Option`, or `Result`), doing logic programming (`List`), or doing asynchronous computation (`Future`).
- One mnemonic which might help understanding monads that embed control flow is to read `M[A]` as "an action in monad `M` which yields `A`"
  - For example `State[Int]` is an action in the `State` monad which yields an `Int`. It might read or write the state, or do nothing with the state and just perform some pure computation (`point` / `return`)

Option

- The Option monad builds computations that can fail (silently, so use Result when you can)
- Failure is represented as None, while success is represented by Some
- join for Some(Some(a)) yields Some(a), any other combination yields None
- bind / flatMap applies f only if given a Some, and f can fail by yielding None or succeed by yielding Some(b). binding None yields None and f is skipped since there’s no value to give to f

Option Example

```scala
import scalaz.std.option.optionInstance
import scalaz.syntax.bind.ToBindOps /* >>= */

def halve(in: Option[Int]): Option[Int] =
    in >>= { a =>
       if (a % 2 == 0) Some(a/2) else None
   }

halve(None) == None
halve(Some(1)) == None
halve(Some(2)) == Some(1)
```

Reader

- The `Reader[R, A]` monad threads some read-only context `R` that each step in the computation can read from.
- `Reader[A]` is isomorphic to `R => A` where `R` is the context to read.
  - So functions that you bind into a workflow of type `A => Reader[B]` are isomorphic to `A => R => B` which means each function needs some input value, the context value, and produces a result of type `B`
- Binding proceeds by applying the same context value, e.g. `ma >>= f` is equivalent to `r => f(ma(r))(r)`
  - Each binding creates a new function which accepts the context `r`, gives that context to the `ma` reader, then gives the result of that and the same context to `f`

Reader Example

```scala
import scalaz.Reader
// Reader typeclasses are in Kleisli because
// Reader[R, A] = Kleisli[Id, R, A]
import scalaz.Kleisli.{ask, kleisliIdMonadReader}
import scalaz.syntax.pointed.PointedIdV /* point */

type RM[+A] = Reader[Int, A]

val computation: RM[Int] =
    for {
        a <- 10.point[RM]
        b <- ask: RM[Int]
        c <- ask: RM[Int]
    } yield a + b + c

computation(4) == 18
computation(125) == 260
```

Writer

- Converse of `Reader[R, A]`, the `Writer[L, A]` monad composes computations that might emit some kind of "side channel" value. `Reader` is the opposite, where computations might read some kind of side channel, known as the context.
- For `Writer`, since a value can be emitted by multiple computations, the written values are collected in a `Monoid`. This `Monoid` is sometimes called the log.
- `Writer[L, A]` is isomorphic to `L => (L, A)`. That is, actions in the writer monad take in the previous state of the log and produce a new state of the log along with the result of the computation.
- `Writer[L, A]` is the same as `State[S, A]` (described next) except that `State` allows a computation to read the current state (log) but `Writer` does not. In addition, `Writer` mandates that writing to the state concatenates via the `Monoid` instead of replacing the state value.

Writer Example

```
import scalaz.Writer
import scalaz.WriterT.{tell, writerMonad}
import scalaz.std.list.listMonoid

type WM[+A] = Writer[List[String], A]

val computation: WM[Int] =
    for {
        _ <- tell(List("starting computation!"))
        val first = 10
        _ <- tell(List("first is " + first))
        val second = 5
        _ <- tell(List("second is " + second))
    } yield first + second

val (log, result) = computation.run
log == List(
    "starting computation!", "first is 10", "second is 5"
)
result == 15
```

State

- The `State[S, A]` monad is a little like a blend of the `Reader` and `Writer` monads. It threads some updatable state `S` through computations which can read or alter that state.
  - Note that because `State` does not entail mutation you can "rewind" a state monad to an earlier state by using a previous `State[S, A]` value
- `State[S, A]` is isomorphic to `S => (S, A)`. That is, each action in the State monad is a function that takes the current state and produces a new state along with some result.
- Binding proceeds fairly straightforwardly, `ma >>= f` is equivalent to `s => { val (s2, a) = ma(s); f(a)(s2) }`
  - That is, each binding produces a function which takes in the prior state (`s`), threads it through `ma` via application, and then applies `f` to the result (`a`) and the updated state (`s2`).

State Example

```scala
import scalaz.State
import scalaz.State.{get, put}
import scalaz.StateT.stateMonad

type SM[+A] = State[Int, A]

val computation: SM[String] =
    for {
        i <- get: SM[Int]
        val j = i + 10
        _ <- put(j)
    } yield "i was " + i + ", now " + j

val (state, result) = computation.run(5)
state == 15
result == "i was 5, now 15"

val (state, result) = computation.run(100)
state == 110
result == "i was 100, now 110"
```

Monad Transformers

- Monads are just fine for encapsulating behaviors, but what if you need to compose behaviors? For example, what if you want to thread a State but also allow for the computation to fail?
- Monad transformers are the solution. They allow you to construct a `Monad` out of pieces.
- By convention, monad transformers have a name ending with `T`, e.g. `ReaderT` (`Kleisli`), `WriterT`, `StateT` .
- They take an underlying Monad as the first type argument, e.g. `StateT[Option, S, A]` is a Monad which composes `State` and `Option`.
- Type level lambdas are used quite often for transformers. E.g. when nesting `State` inside some other monad, the outer monad will expect `M[_]` but `State` is `State[S, A]`, so one might need to use a type lambda to curry off `S`, e.g. `({ type F[A] = State[MyState, A] })#F`

Monad Transformer Example

```
import scalaz.{Id, Kleisli, State, StateT}
import Kleisli.kleisliMonadTrans
import StateT.stateMonad
import scalaz.syntax.monad.ToMonadOpsUnapply
```

```
final case class MyContext(readOnly: String)
final case class MyState(readWrite: String)

type SM[+A] = State[MyState, A]
type Computation[+A] =
    Kleisli[SM, MyContext, A]
val computationMonadTrans = kleisliMonadTrans[MyContext]
import computationMonadTrans.liftM

val computation: Computation[String] =
    for {
        context <- Kleisli.ask: Computation[MyContext]
        priorState <- liftM(State.get: SM[MyState])
        val newState = MyState(priorState.readWrite + " with more")
        _ <- liftM(State.put(newState): SM[Unit])
    } yield "context = " + context + " prior = " + priorState

val (finalState, result) =
    computation.run(MyContext("foo")).apply(MyState("initial"))

finalState == MyState("initial with more")
result = "context = MyContext(foo) prior = MyState(initial)"
```

Monad Transformers (cont.)

- Monad transformers "thread" an operation all the way down the stack, so all the behaviors are composed at each step. This means for example if you compose `Validation` and `Result` which both express a kind of failure case, then if either the `Validation` or `Result` behavior results in a failure the overall result is failure.
- Most non-transformer monads in scalaz are type aliases that compose their transformer equivalent with the `Id` monad. E.g. `Reader[R, A] = ReaderT[Id, R, A]`
- Monad transformers in general can be tricky to work with, so use your judgement. You can pre-blend together certain monad behaviors to make it easier to use.
  - To this end, most monad transformers have a corresponding typeclass particular to their behavior, so you can provide that typeclass for your composed monad and make it easier to use, rather than `liftM`ing all over. For example `StateT` has `MonadState`, `Kleisli` / `ReaderT` has `MonadReader`.