香港科技大學
THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 13)

COMP 3031, Fall 2025

Lionel Parreaux

Abstract Algebra and Type Classes

Doing Abstract Algebra with Type Classes

Type classes let one define concepts that are quite abstract, and that can be instantiated with many types. For instance:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T
```

This models the algebraic concept of a semigroup with an associative operator `combine`.

Doing Abstract Algebra with Type Classes

Type classes let one define concepts that are quite abstract, and that can be instantiated with many types. For instance:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T
```

This models the algebraic concept of a semigroup with an associative operator `combine`.

We can then define methods that work for all semigroups. For instance:

```
def reduce[T: SemiGroup](xs: List[T]): T =
  xs.reduceLeft(_.combine(_))
```

Type Class Hierarchies

Algebraic type classes often form natural hierarchies. For instance, a *monoid* is defined as a semigroup with a left- and right-unit element.

Here's its natural definition:

```
trait Monoid[T] extends SemiGroup[T]:
  def unit: T
```

Where `unit` is called the *neutral element* of `combine`.

Exercise

Generalize `reduce` to work on lists of `T` where `T` has a `Monoid` instance such that it also works for empty lists.

Exercise

Generalize `reduce` to work on lists of `T` where `T` has a `Monoid` instance such that it also works for empty lists.

```
def reduce[T](xs: List[T])(using m: Monoid[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
```

Exercise

Generalize `reduce` to work on lists of `T` where `T` has a `Monoid` instance such that it also works for empty lists.

```
def reduce[T](xs: List[T])(using m: Monoid[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
```

Question: Does this compute the same result as the previous definition on non-empty input lists? Can we prove it?

Using Context Bounds

In the previous example we had to pass an explicitly named type class instance `m: Monoid[T]` to `reduce`, so that we could refer to `m.unit`.

One could alternatively use a context bound and a `summon`.

```
def reduce[T: Monoid](xs: List[T]): T =
  xs.reduceLeft(summon[Monoid[T]].unit)(_.combine(_))
```

Streamlining Access

A simpler calling syntax can be obtained if we do some preparation in the `Monoid` trait itself.

```
trait Monoid[T] extends SemiGroup[T]:
  def unit: T
object Monoid:
  def apply[T](using m: Monoid[T]): Monoid[T] = m
```

This defines a global function `Monoid.apply[T]` that returns the `Monoid[T]` instance that is currently visible.

With this helper, `reduce` can be written like this:

```
def reduce[T: Monoid](xs: List[T]): T =
  xs.reduceLeft(Monoid[T].unit)(_.combine(_))
```

Multiple Type class Instances

It's possible to have several given instances for a typeclass/type pair. For instance, `Int` could be a `Monoid` in (at least) two ways:

- with `+` as `combine` and `0` as `unit`, or
- with `*` as `combine` and `1` as `unit`.

```
given sumMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int) : Int = x + y
  def unit: Int = 0

given prodMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int) : Int = x * y
  def unit: Int = 1
```

Exercise

Define the `sum` and `product` functions on `List[Int]` in terms of `reduce`.

Exercise

Define the `sum` and `product` functions on `List[Int]` in terms of `reduce`.

```
def sum(xs: List[Int]): Int = reduce(xs)(using sumMonoid)
def product(xs: List[Int]): Int = reduce(xs)(using prodMonoid)
```

What happens if you leave out the `using` arguments?

Exercise

Define the `sum` and `product` functions on `List[Int]` in terms of `reduce`.

```
def sum(xs: List[Int]): Int = reduce(xs)(using sumMonoid)
def product(xs: List[Int]): Int = reduce(xs)(using prodMonoid)
```

What happens if you leave out the `using` arguments?

An ambiguity error.

Type class Laws

Algebraic type classes are not just defined by their type signatures but also by the laws that hold for them.

For example, any given instance of `Monoid[T]` should satisfy the laws:

```
x.combine(y).combine(z)  ==  x.combine(y.combine(z))
        unit.combine(x)  ==  x
        x.combine(unit)  ==  x
```

where `x`, `y`, `z` are arbitrary values of type `T` and `unit = Monoid.unit[T]`.

The laws can be verified either by a formal or informal proof, or by testing them.

A good way to test that an instance is *lawful* is using randomized testing with a tool like ScalaCheck.

Higher-Kinded Type Classes

We saw that the concept of *monad* is implemented by types like `List` and `Option` in terms of `flatMap` and `unit`.

Question: Can `Monad` be a typeclass?

Higher-Kinded Type Classes

We saw that the concept of *monad* is implemented by types like `List` and `Option` in terms of `flatMap` and `unit`.

Question: Can `Monad` be a typeclass?

What is, then, the type of `unit`? And `flatMap`?

```
trait Monad[M]:
  def unit[T](x: T): ???
  extension [T](x: ???)
    def flatMap[U](f: T => ???): ???
```

Higher-Kinded Type Classes

We saw that the concept of *monad* is implemented by types like `List` and `Option` in terms of `flatMap` and `unit`.

Question: Can `Monad` be a typeclass?

What is, then, the type of `unit`? And `flatMap`?

```
trait Monad[M]:
  def unit[T](x: T): ???
  extension [T](x: ???)
    def flatMap[U](f: T => ???): ???
```

Answer: We need some new type concepts first.

Monad is a property not of a *type* but of a *type constructor*.

Indeed, `List` itself is a monad, not `List[T]` for some `T`.

Higher-Kinded Type Classes

We saw that the concept of *monad* is implemented by types like `List` and `Option` in terms of `flatMap` and `unit`.

Question: Can `Monad` be a typeclass?

What is, then, the type of `unit`? And `flatMap`?

```
trait Monad[M]:
  def unit[T](x: T): ???
  extension [T](x: ???)
    def flatMap[U](f: T => ???): ???
```

Answer: We need some new type concepts first.

Monad is a property not of a *type* but of a *type constructor*.

Indeed, `List` itself is a monad, not `List[T]` for some `T`.

So we need to a way to abstract over type constructors, the same way type parameters abstract over plain types.

Higher-Kinded Types

Abstracting over type constructors is done using *higher-kinded* types.

For instance:

```
def foo[F[_], X](f: X => F[X], x: X): F[X] = f(x)
```

Higher-Kinded Types

Abstracting over type constructors is done using *higher-kinded* types.

For instance:

```
def foo[F[_], X](f: X => F[X], x: X): F[X] = f(x)

foo[List, Int](_ :: Nil, 1)           // == List(1)
foo[Option, String](Some(_), "fuel") // == Some("fuel")
```

Here, `F[_]` is a type parameter that can be instantiated with type constructors, not plain types like `Int`.

Type Functions

To be exact, arbitrary *type functions* are also supported in Scala 3:

```
foo[[X] =>> (X, X), Int](x => (x, x), 1)  // == (1, 1)
foo[[X] =>> String, Int](x => "hi", 1)    // == "hi"
```

These are equivalent to:

```
type G[X] = (X, X)
foo[G, Int](x => (x, x), 1)  // == (1, 1)

type H[X] = String
foo[H, Int](x => "hi", 1)    // == "hi"
```

The Monad Type class

We now can forumalate a `Monad` type class as follows:

```
trait Monad[F[_]]:
  def unit[T](x: T): F[T]
  extension [T](x: F[T])
    def flatMap[U](f: T => F[U]): F[U]
    def map[U](f: T => U): F[U] = flatMap(f andThen unit)
```

The Monad Type class

We now can forumalate a `Monad` type class as follows:

```
trait Monad[F[_]]:
  def unit[T](x: T): F[T]
  extension [T](x: F[T])
    def flatMap[U](f: T => F[U]): F[U]
    def map[U](f: T => U): F[U] = flatMap(f andThen unit)
```

An implementation:

```
given ListMonad: Monad[List] with
  def unit[T](x: T): List[T] = x :: Nil
  extension [T](x: List[T])
      def flatMap[U](f: T => List[U]): List[U] = x.flatMap(f)
```

Should Monad be a Type class?

The advantage of monad being a type class is that we can define very abstract and generic operations that work for all monadic structures.

For example, we can define *sequence*, a function that permutes a list of some monadic type:

```
def sequence[F[_]: Monad, A](as: List[F[A]]): F[List[A]]
```

Should Monad be a Type class?

The advantage of monad being a type class is that we can define very abstract and generic operations that work for all monadic structures.

For example, we can define *sequence*, a function that permutes a list of some monadic type:

```
def sequence[F[_]: Monad, A](as: List[F[A]]): F[List[A]]
```

Example uses (assuming a `Monad[Option]` instance):

```
sequence(List(Some(1), Some(2), Some(3))) // == Some(List(1, 2, 3))
sequence(List(Some(1), None, Some(3)))    // == None
```

Should Monad be a Type class?

The advantage of monad being a type class is that we can define very abstract and generic operations that work for all monadic structures.

For example, we can define *sequence*, a function that permutes a list of some monadic type:

```
def sequence[F[_]: Monad, A](as: List[F[A]]): F[List[A]]
```

Example uses (assuming a `Monad[Option]` instance):

```
sequence(List(Some(1), Some(2), Some(3))) // == Some(List(1, 2, 3))
sequence(List(Some(1), None, Some(3)))    // == None
```

How many languages you know can express things like this?

Exercise

Exercise (recommended): implement sequence.

```
def sequence[F[_]: Monad, A](as: List[F[A]]): F[List[A]] =
```

Exercise

Exercise (recommended): implement `sequence`.

```
def sequence[F[_]: Monad, A](as: List[F[A]]): F[List[A]] =
  as match
    case Nil =>
```

Exercise

Exercise (recommended): implement sequence.

```
def sequence[F[_]: Monad, A](as: List[F[A]]): F[List[A]] =
  as match
    case Nil =>
      summon[Monad[F]].unit(Nil)
    case fa :: fas =>
```

Exercise

Exercise (recommended): implement sequence.

```
def sequence[F[_]: Monad, A](as: List[F[A]]): F[List[A]] =
  as match
    case Nil =>
      summon[Monad[F]].unit(Nil)
    case fa :: fas =>
      for a <- fa; as <- sequence(fas) yield a :: as
```

Done!

Other Useful Higher-Kinded Type Classes

The Cats library (`https://typelevel.org/cats`) contains many useful type classes to program generically in this “algebraic” style.

Other Useful Higher-Kinded Type Classes

The Cats library (`https://typelevel.org/cats`) contains many useful type classes to program generically in this "algebraic" style.

In fact, `Monad` inherits from a more basic type class called `Applicative`:

```
trait Applicative[F[_]] extends Functor[F]:
  def pure[A](a: A): F[A]
  extension [A](x: F[A])
    def ap[A, B](f: F[A => B]): F[B]
    def map[A, B](f: A => B): F[B] = ap(pure(f))(fa)
```

Other Useful Higher-Kinded Type Classes

The Cats library (`https://typelevel.org/cats`) contains many useful type classes to program generically in this "algebraic" style.

In fact, `Monad` inherits from a more basic type class called `Applicative`:

```
trait Applicative[F[_]] extends Functor[F]:
  def pure[A](a: A): F[A]
  extension [A](x: F[A])
    def ap[A, B](f: F[A => B]): F[B]
    def map[A, B](f: A => B): F[B] = ap(pure(f))(fa)
```

which is itself a special case of `Functor`:

```
trait Functor[F[_]]:
  extension [T](x: F[T])
    def map[U](f: T => U): F[U]
```

Other Useful Higher-Kinded Type Classes

Cats generalizes `sequence` further, as part of type class `Traverse`:

```
trait Traverse[F[_]] {
  // Abstract definition to be implemented:
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
  // Derived implementation:
  def sequence[G[_]: Applicative, A](fga: F[G[A]]): G[F[A]] = ... }
```

So one can operate on many combinations of types, such as:

```
sequence(Array(Right(0), Right(1), Left("oops")))
```

```
sequence(Vector(List(0, 1, 2), List(3, 4)))
```

Context Management

Context Passing vs Type Classes

Type classes are about *type instances of generic traits*. E.g.:

- What is the definition of `TC[A]` for the type class trait `TC` and the type argument `A`?

If we want to make `A` a type parameter, we need an implicit parameter to go with it.

On the other hand, there are also uses for abstracting over values of a simple type, asking

- What is the currently valid definition of type `T`?

Example: Execution contexts

To do computations in parallel, runtimes need *thread schedulers*.

There's usually a default scheduler, but it should be possible to override that choice in parts of the code.

How are references to schedulers propagated?

In Scala, they are embedded in values of types `ExecutionContext`. The default is:

```
given global: ExecutionContext = ForkJoinContext()
```

This defines the execution context `global` as an alias of an existing value (i.e. a freshly created `ForkJoinContext`)

The evaluation of `ForkJoinContext` is done lazily: the `ForkJoinContext` is created the first time `global` is used.

Propagating Execution Contexts

Execution contexts rarely change, but they should be changeable everywhere.

This is a poster-child for implicit parameters.

```
def processItems(...)(using ExecutionContext) = ...
```

Other Use Cases

Passing a piece of the context as an implicit parameter of a certain type is quite common.

For instance, we might want to propagate implicitly

- the current configuration,
- the available set of capabilities,
- the security level in effect,
- the layout scheme to render some data,
- The users that have access to some data.

Be Specific

Given instances should have specific types and/or be local in scope.

For example, this is a terrible idea:

```
given Int = 1
def f(x: Int)(using delta: Int) = x + delta
```

Never use a common type such as `Int` or `String` as the type of a globally visible given instance! It's too easy to mix them up.

Be Specific

Given instances should have specific types and/or be local in scope.

For example, this is a terrible idea:

```
given Int = 1
def f(x: Int)(using delta: Int) = x + delta
```

Never use a common type such as `Int` or `String` as the type of a globally visible given instance! It's too easy to mix them up.

An alternative is to use *opaque types*:

```
opaque type Delta = Int
def init(n: Int): Delta = 1
def f(x: Int)(using delta: Delta) = x + delta
```

Outside the current scope, `Delta` is unknown – cannot be mixed up.

Opaque Type Aliases: Definition

Consider this toy example

```scala
// Some domain models
case class Person(name: String, age: Int)

// A conference management system
object ConfManagement:
    opaque type Viewers = Set[String]

    // A way of creating a starting context value
    def createViewers(ps: Person*): Viewers = ps.map(_.name).toSet

    // Some operation that requires the context
    def someTask(arg: Int)(using Viewers): Int =
        ... summon[Viewers].contains(someName) ...
```

Opaque Type Aliases: Use

The equality `Viewers = Set[Person]` is known only within the scope where the alias is defined (in this case, within the `ConfManagement` object).

Everywhere else `Viewers` is treated as a separate, abstract type. So this is a type error:

```
ConfManagement.someTask(42)(using Set(”Oops”))
```

The correct usage is:

```
val p = Person(”Alice”, 42)
val q = Person(”Bob”, 27)
given ConfManagement.Viewers =
  ConfManagement.createViewers(p, q)

ConfManagement.someTask(42)
```

Exercise

Let's augment the previous `enum` for arithmetic expressions with a `Let` form:

```
enum Expr:
  case Number(num: Int)
  case Sum(x: Expr, y: Expr)
  case Prod(x: Expr, y: Expr)
  case Var(name: String)
  case Let(name: String, rhs: Expr, body: Expr)
import Expr._
```

Write an `eval` function for expressions of this type.

```
def eval(e: Expr): Int = ???
```

`Let(”x”, e1, e2)` should be evaluated like `{val x = e1; e2}`.
Assume every `Var(x)` occurs in the body `b` of an enclosing `Let(x, e, b)`.

Solution Hint

Use a map from variable names to their defined values as an implicit parameter.

The map is initially empty and is augmented in every `Let` node.

This suggests the following outline:

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = ???

  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =

  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)             =>
    case Sum(x, y)             =>
    case Prod(x, y)            =>
    case Var(name)             =>
    case Let(name, rhs, body) =>

  recur(e)(using Map())
```

Solution

```scala
def eval(e: Expr): Int =

  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)              => n
    case Sum(x, y)              =>
    case Prod(x, y)             =>
    case Var(name)              =>
    case Let(name, rhs, body) =>

  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =

  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)              => n
    case Sum(x, y)              => recur(x) + recur(y)
    case Prod(x, y)             =>
    case Var(name)              =>
    case Let(name, rhs, body) =>

  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =

  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)               => n
    case Sum(x, y)               => recur(x) + recur(y)
    case Prod(x, y)              => recur(x) * recur(y)
    case Var(name)               =>
    case Let(name, rhs, body) =>

  recur(e)(using Map())
```

Solution

```scala
def eval(e: Expr): Int =

  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)             => n
    case Sum(x, y)             => recur(x) + recur(y)
    case Prod(x, y)            => recur(x) * recur(y)
    case Var(name)             => env(name)
    case Let(name, rhs, body) =>

  recur(e)(using Map())
```

Solution

```scala
def eval(e: Expr): Int =

  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)              => n
    case Sum(x, y)              => recur(x) + recur(y)
    case Prod(x, y)             => recur(x) * recur(y)
    case Var(name)              => env(name)
    case Let(name, rhs, body) =>
      recur(body)(using env + (name -> recur(rhs)))

  recur(e)(using Map())
```