

# Chapter 5: Type classes and their applications

Sergei Winitzki

Academy by the Bay

January 14, 2018

# Motivation for type classes I: Restricting type arguments

We need different `sum` implementations for `Seq[Int]`, `Seq[Double]`, etc.

- but we cannot generalize `sum` to arbitrary types `T` like this:

```
def sum[T](s: Seq[T]): T = ???
```

- this can work only for `T` that have a zero value and a `+` method

Suppose we want to define `fmap` in addition to `map` for functors

- but we cannot generalize `fmap` to arbitrary type constructors `F[_]`:

```
def fmap[F[_], A, B](f: A ⇒ B): F[A] ⇒ F[B] = ???
```

- this can work only for type constructors `F[_]` that are functors

We would like to define functions whose type arguments, such as `T` or `F[_]`, are assumed to belong to a *certain subset* of possible types

- We could then use the known properties of these type arguments
- We would also like to add new supported types as needed
  - ▶ This is similar to *partial functions* – but at type level

# Motivation for type classes II: Partial type-level functions

- Functions can be **total** or **partial**
  - ▶ Total function: has a result for all argument values
  - ▶ Partial function: has *no result* for *some* argument values
- Also, functions can be, in principle, {from/to} {values/types}:

function:	from value	from type
to value	<code>def f(x: Int): Int</code>	<code>def point[A]: A ⇒ List[A]</code>
to type	<i>dependent type</i>	<code>type Data[A] = Either[Int, A]</code>

- value-to-value = run time, type-to-\* = compile time
  - ▶ if we use JVM reflection, type-to-\* can become run-time (*yuck!*)

partial function:	from value (PF)	from type (PTTF; PTVF)
example:	<code>{ case Some(x) ⇒ x-1 }</code>	GADTs; <code>implicitly[T]</code>
when misapplied:	exception at run time	error at compile time

- **Type classes** are a systematic way of managing our PTFs
  - ▶ It is safe to apply a PTF to type **T** if **T** “belongs to a certain type class”

## Example of using value-level PFs: The caveats

- Filter a `Seq[Either[Int, Boolean]]`, then apply `map` with a PF:

```
val s: Seq[Either[Int, Boolean]] = Seq( Left(1), Right(true), Left(2) )  
  .filter(_._isLeft) // result here is still of type Seq[Either[Int, Boolean]]  
  .map { case Left(x) => x } // result is of type Seq[Int] but unsafe
```

- “We know” it is okay to apply this PF here...
  - ▶ but the types do not show this, – compile-time checking doesn't help
  - ▶ if refactored, the code may become wrong and break *at run time*

- The type-safe version uses `.collect` instead of `.filter().map()`:

```
val s: Seq[Int] = Seq( Left(1), Right(true), Left(2) )  
  .collect { case Left(x) => x } // result is safe, of type Seq[Int]
```

- PFs are only safe to use in certain places, such as `.collect()`
  - ▶ in all other cases, value-level functions should better be total
  - ▶ can use “refined” types such as “non-empty list”, “positive number” etc.

```
def f(xs: NonEmptyList[Int]) = {  
  val h = xs.head // safe and checked at compile time  
}
```

# Managing PTFs by hand I: Using GADTs

## PTTFs: Partial Type-to-Type Functions

- A type constructor that accepts only certain types as parameters:

```
sealed trait MyTC[A] // “sealed” GADT – user code can’t add cases
final case class Case1(d: Double) extends MyTC[Int]
final case class Case2() extends MyTC[String] // whatever
```

- It looks like we have defined `MyTC[A]` for any type `A` ?...
  - ▶ actually, we can only ever create values of `MyTC[Int]` or `MyTC[String]`
- Effectively, `MyTCA` is a PTTF defined only for  $A \in \{\text{Int}; \text{String}\}$ 
  - ▶ This **type domain** is enforced *at compile time*!
- When to use GADTs (they are *not* functors!):
  - ▶ for domain modeling (e.g. queries with a fixed set of result types)
  - ▶ for DSLs that represent typed expressions
- Alternatively, a PTTF can be a `trait` with some implementation code:

```
trait MyPTTF[A] {...} // not “sealed” – user code may extend
class C1(...) extends MyPTTF[Int] {...} // arbitrary code
```

# Managing PTFs by hand II: Using traits and OO inheritance

PTVFs: Partial Type-to-Value Functions – the object-oriented way

- A trait with methods and a few created values (as `object`):

```
trait HasPlus[A] {  
  def plus(a1: A, a2: A): Z  
}  
object CaseInt extends HasPlus[Int] {  
  def plus(a1: Int, a2: Int): Int = a1 + a2  
}  
object CaseString extends HasPlus[String] {  
  def plus(a1: String, a2: String): String = a1 + a2  
}
```

- *Similar* to having defined `plus[A]` only for  $A \in \{\text{Int}; \text{String}\}$
- Limitations:
  - ▶ We can only access `plus()` via a value of type `HasPlus[A]`
  - ▶ All PTVFs must be declared up front in the trait
    - ★ Not extensible – cannot add new PTVFs later
    - ★ Not compositional – cannot use this in other PTVFs defined later

# Managing PTFs by hand III: “Type Evidence” arguments

PTVFs: Partial Type-to-Value Functions – the general case

To define a function `def func[A](...)` only for certain types `A`:

- 1 create a PTF defined only for the relevant types `A`, e.g. `IsGood[A]`
- 2 add an *extra argument* of type `IsGood[A]` (**type evidence**) to `func[A]`
- 3 create some values of types `IsGood[A]` for relevant types `A` as needed

What we gained:

- it is now impossible to call `func[A]` with an unsupported type `A`
  - ▶ trying to do so will fail *at compile time* – TE values won't type-check
- new supported types can be added in user code if `IsGood` is not `sealed`

The cost:

- all calls to `func[A](...)` will now need TE values as extra argument(s)
- we need to keep passing the TE values around the code
- one TE value needs to be created for *each* supported type `A`

Mitigate these issues in Scala by using `implicit` values:

- TE arguments are explicit only at `func` declaration site
- once defined as `implicit`, TE values are passed around invisibly
- new `implicit` values can be built up automatically (and recursively!)

# Scala's mechanism of “implicit values”

Implicit values are:

- declared as `implicit val x: SomeType = ...`
  - ▶ also have `implicit def f[T](...) = ...` and `implicit class(...)`
- automatically passed into functions that declare extra arguments as  
`def f(args...)(implicit x: SomeType) = ...`
- searched in local scope, imports, companion objects, parent classes
  - ▶ having  $\geq 2$  `implicit` values of the same type is a compile-time error!
- standard library has `def implicitly[A](implicit x: A): A = x`

Special short syntax for declaring implicit TE arguments in a PTVF:

```
def func[A: MyTypeClass](args...) = ...
```

This is equivalent to

```
def func[A](args...)(implicit ev: MyTypeClass[A]) = ...
```

We still need to:

- declare `MyTypeClass[A]` as a PTTF elsewhere
- create TE values of various types and declare them as `implicit`



# Type classes I

## The general definition

A **type class** is a set of PTVFs that all have the same type domain

- In terms of specific code to be written, a type class is:
  - ① a PTTF, e.g. `MyTypeClass[T]` with some code that creates TE values,  
*and*
  - ② the desired PTVFs that use this PTTF to define their type domain
    - ▶ for many important use cases, the PTVFs must also satisfy certain laws
- A type `T` “**belongs to** the type class `MyTypeClass`” if a TE value exists
  - ▶ i.e. if *some* value of type `MyTypeClass[T]` can be found
- A function `func[T]` “requires the type class `MyTypeClass` for `T`” if one of `func`’s arguments is a value of PTTF type `MyTypeClass[T]`
  - ▶ that argument is the **type class instance** for the type parameter `T`
  - ▶ this **constrains** the type parameter `T` to **belong to** the type class

# Type classes II

## Implementation in Scala

A type class is typically implemented as:

- a trait with a type parameter, e.g. `trait MyTypeClass[T]`
- code that creates values of type `MyTypeClass[T]` for various `T`
  - ▶ these values are declared as `implicit` and made available via imports or in the companion objects for the specific types `T`
- some functions with implicit argument(s) of type `MyTypeClass[T]`
  - ▶ these functions are usually `def` methods in a trait, but don't have to be
  - ▶ laws for these functions may need to be enforced by property tests

Usually, all information about the type `T` is contained in the TE value

- the trait `MyTypeClass[T]` contains all relevant PTVFs as `def`'s
- in simpler cases, TE can be a data type (not a trait with `def` methods)
  - ▶ a trait with `def` methods is necessary for *higher-order* type functions

See example code

# Examples of type classes I

Some simple PTFs and their use cases

- A type `T` is a **semigroup** if it has an *associative* binary operation

```
def op(x: T, y: T): T
```

- ▶ a bare-bones operation, no inverse – just “can combine”

- A type `T` is **pointed** if there exists a function `point: T`

- ▶ This is a special, somehow “naturally” selected value of that type

★ Examples: `0: Int`; `"": String`; `identity[A]: A ⇒ A`

- A type `T` is a **monoid** if there exist functions

```
def empty[T]: T
```

```
def combine[T](x: T, y: T): T
```

such that the usual algebraic laws hold:

- ▶ `combine` is associative
- ▶  $\forall x : \text{combine}(\text{empty}, x) = \text{combine}(x, \text{empty}) = x$

- Monoids are an abstraction for any sort of data aggregation

See example code for implementing the `Monoid` type class:

- by using a case class as a PTF (instance from scratch)
- by assuming `Pointed` and `Semigroup` (“derived” instance)

# Examples of type classes II

## Higher-order PTFs

- A type constructor  $F^A$  is a functor if it has a `map` operation
  - ▶ or, equivalently, `fmap`
  - ▶ that satisfies the functor laws (identity law, composition law)
- We would like to write a generic function that tests the functor laws

```
def checkFunctorLaws[F[_], A, B, C](): Assertion = ???
```

- Need to get access to the function `map` defined for the given `F`
- We treat `map` as a PTVF whose type domain is all functors `F`:

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

- We constrain `F` to belong to the `Functor` type class
  - ▶ by adding `implicit ev: Functor[F]` as extra argument to `map`

★ note: `Functor` is a *higher-order* PTTF – its type argument is `F[_]`

See test code for implementation and functor laws

# Types and kinds

Compare value-to-value functions (VVF) vs. type-to-value functions:

- the **domain** of a VVF is the set of admissible argument values
  - ▶ a “value domain” is called a **type**
  - ▶ the VVF can be applied safely if its argument is of the right **type**  
`def f(x: Option[Int]) = ...`
- the **type domain** of a PTVF is the set of admissible argument types
  - ▶ a “type domain” is called a **kind**
  - ▶ the PTVF can be applied safely if its type argument is of the right **kind**  
`def func[T](args...)(implicit ev: MyTypeClass[T]) = ...`
- In both cases, the function call safety is guaranteed *at compile time*

Kinds are the “type system for types”

- a type class `MyTypeClass` defines a new kind (as a set of types)
  - ▶ suggested **kind** notation: `(* : MyTypeClass)`
- another available kind is the **type constructor** kind, e.g.: `F[_]`
  - ▶ in `F[T]`, the `F` and the `T` are types of different **kinds**
  - ▶ define `type Ap[F[_], T] = F[T]`, then wrong kinds will fail in `Ap[A, B]`
    - ★ suggested **kind** notation: `Ap : (* → *, *) → *`
  - ▶ See test code

# Scala's “implicit method” syntax for PTVFs

Two sorts of available syntax for Scala functions:

- ① as in ordinary math: `func(x, y)` or `func(x, y)(z)` etc.
- ② as “method”: `x.func(y)` or equivalently `x func y`
  - ▶ this is similar to `func(x)(y)` but is implemented differently

It is often convenient to use functions syntactically as methods

```
def +++[T: HasPlusPlusPlus](t: T, arg: ...) = ...  
val t: T = ...  
+++ (t, arg) // that's how we have to call this function  
// but instead we want to be able to write t +++ arg
```

Implementing the “implicit method syntax” for a PTVF `func`:

- declare `func` as a method on a new trait or class, say `MyTCMethods[T]`
- declare an *implicit conversion* function from `T` to `MyTCMethods[T]`
  - ▶ to make the code shorter, use an `implicit class`
  - ▶ see example code

What we gained:

- the PTVF appears as a method *only* on values of the relevant types
- the new syntax is defined automatically on *all* the relevant types `T`

# Worked examples

- 1 Define a PTVF `def bitsize[T] = ...` such that `bitsize[Int]` returns 32 and `bitsize[Long]` returns 64; otherwise `bitsize[T]` is undefined
- 2 Define a monoid instance for the type  $1 + (\text{String} \Rightarrow \text{String})$
- 3 Assuming that  $A$  and  $B$  are monoids, define monoid instance for  $A \times B$
- 4 Show: If  $A$  is a monoid and  $B$  is a semigroup then  $A + B$  is a monoid
- 5 Define a functor instance for `type F[T] = Seq[Try[T]]`
- 6 Define a Cats' `Bifunctor` instance for  $Q^{X,Y} \equiv X + X \times Y$
- 7 Define a `ContraFunctor` type class having `contrafmap`:

`def contrafmap[A, B](f: B  $\Rightarrow$  A): C[A]  $\Rightarrow$  C[B]`

Define a `ContraFunctor` instance for type constructor  $C^A \equiv A \Rightarrow \text{Int}$

- 8 Define functor instance for recursive type  $Q^A \equiv (\text{Int} \Rightarrow A) + \text{Int} + Q^A$
- 9 \* If  $F^A$  and  $G^A$  are functors, define functor instance for  $F^A + G^A$

# Exercises

- ❶ Define a PTVF `def isLong[T]: Boolean` that returns `true` for `Long` and `Double`; returns `false` for `Int`, `Short`, and `Float`; otherwise undefined
- ❷ Define a monoid instance for the type `String × (1 + Int)`
- ❸ If  $A$  is a monoid and  $R$  any type, define monoid instance for  $R \Rightarrow A$
- ❹ Show: If  $S$  is a semigroup then `Option[S]` is a monoid
- ❺ Define a functor instance for `type F[T] = Future[Seq[T]]`
- ❻ Define a Cats' `Bifunctor` instance for  $B^{X,Y} \equiv (\text{Int} \Rightarrow X) + Y \times Y$
- ❼ Define a `ProFunctor` type class having `dimap`:

```
def dimap[A, B](f: A ⇒ B, g: B ⇒ A): F[A] ⇒ F[B]
```

Define a `ProFunctor` instance for  $P^A \equiv A \Rightarrow (\text{Int} \times A)$

- ❽ Define a functor instance for recursive type  $Q^A \equiv \text{String} + A \times Q^A$
- ❾ \* If  $F^A$  and  $G^A$  are functors, define functor instance for  $F^A \times G^A$
- ❿ \* Define a functor instance for  $F^A \Rightarrow G^A$  where  $F^A$  is a contrafunctor (use Cats' `Contravariant` type class for  $F^A$ ) and  $G^A$  is a functor