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

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 constrained to belong to a *certain subset* of possible types

- We could then use the guaranteed properties of these type arguments
- 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}:

functions:	from value	from type
to value	`def f(x: Int): Int`	`def point[A]: A ⇒ List[A]`
to type	*dependent types*	`type Data[A] = Either[Int, A]`

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

designation:	from value (PF)	from type (PTTF; PTVF)
example:	`{ case Some(x) ⇒ ... }`	GADTs; `implicitly[T]`
when misapplied:	exception at run time	error at compile time

- Type classes are a systematic way of managing your PTFs
  - It is safe to apply a PTF to type `T` if `T` "belongs to a certain type class"

Example of uses of PFs: The caveats

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

```
val s: Seq[Int] = Seq( Left(1), Right(true), Left(2) )
  .filter(_.isLeft) // result here is still of type Seq[Either[...]]
  .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: GADTs

PTTFs: Partial Type-to-Type Functions

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

```
sealed trait MyTC[Z] // "sealed" – 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[Z]` for any type `Z` ?...
  - no, we can only ever create values of `MyTC[Int]` or `MyTC[String]`
- So `MyTC[Z]` is a PTTF defined only for `Z = Int` and `Z = String`
- This type constraint is checked and enforced *at compile time*!
- When to use GADTs:
  - for domain modeling (e.g. queries with a fixed set of result types)
  - for DSLs that have typed expressions
- Instead of GADTs, a PTTF can be a trait with implementation code

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

Managing PTFs by hand II: Traits with inheritance

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

- A trait with methods and a few implementations:

```
trait HasPlus[Z] {
  def plus(z1: Z, z2: Z): Z
}
implicit object CaseInt extends HasPlus[Int] {
  def plus(z1: Int, z2: Int): Int = z1 + z2
}
implicit object CaseString extends HasPlus[String] {
  def plus(z1: String, z2: String): String = z1 + z2
}
```

- *Similar* to having defined `plus[Z]` only for `Int` and `String`
- Limitations:
  - We can only access `plus()` via a TE value of type `HasPlus[Z]`
  - All PTVFs must be declared up front in the trait
    - We cannot add other PTVFs later

Managing PTFs by hand III: "Type Evidence" arguments

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

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

1. Create a PTTF defined only for the relevant types `T`, e.g. `IsGood[T]`
2. Add an extra argument of type `IsGood[T]` (**evidence**) to `func[T]`
3. Create some values of type `IsGood[T]` as needed, for relevant types `T`

What we gained:

- it is now impossible to call `func` with an unsupported type `T`
- trying to do so will fail *at compile time*, because TE won't type-check
- If `IsGood[T]` is not `sealed`, more types `T` can be added via user code

The cost:

- all calls to `func` now need TE values as extra argument(s)
- we need to keep passing the TE values around the code
- the TE values need to be created for each supported type `T`

How we mitigate this problem in Scala: use `implicit` values

- TE arguments are needed only at declaration site of `func`
- Once defined as `implicit`, TE values are passed around automatically
- New TE values can be often built up automatically (and recursively!)

Scala's mechanism of "`implicit` values"

Implicit values are:

- declared as `implicit val x: SomeType = ...`
- automatically passed into functions that declare extra arguments as

```
def f(args...)(implicit x: SomeType) = ...
```

- searched in local scope, imports, companion objects, parent classes
- standard library has `def implicitly[T](implicit t: T): T = t`

Special syntax for declaring implicit TE arguments in a PTVF:

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

This is equivalent to

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

We still need to:

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

Type classes I: The 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]`, defining the type domain, *together with*
  2. some code (usually, library imports) that creates some TE values, *and*
  3. one or more PTVFs that use this PTTF for TE arguments

  - for many important use cases, the PTVFs must 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
    - (usually, as a library import)
- A function `func[T]` "requires the type class `MyTypeClass` for `T`" if one of `func`'s arguments is a value of 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 in Scala 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 must be defined as `implicit` and made available via imports or in the *companion objects* for the specific types `T`
- some functions with an implicit argument of type `MyTypeClass[T]`
  - laws for these functions may need to be enforced by 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 **pointed** if there exists a function `point: T`
  - There is a special, somehow "naturally" selected value of that type
    - Examples: `0: Int`, `"": String`, `identity: A ⇒ A`
- A type `T` is a **semigroup** if it has an associative binary operation

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

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

```
def zero[T]: T
def append[T](x: T, y: T): T
```

- such that the usual algebraic laws hold:
  - `append` is associative
  - `append(zero, x) == append(x, zero) == x`

See examples of implementing the `Monoid` type class in various ways

- by using a case class as a PTTF
- by combining `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`
  - or, equivalently, `fmap`
  - that satisfies the functor laws
- 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[A, B]` for `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]`
    - here `Functor[F]` is a *higher-order* PTTF

See test code for implementation and functor laws checking

Worked examples

1. Define a PTVF `def bitsize[T]: Int` such that `bitsize[Int]` returns 32 and `bitsize[Long]` returns 64, and undefined on other types
2. Define a monoid instance for the type $1 + (\text{String} \Rightarrow \text{String})$
3. If A and B are monoids, define monoid instance for $A \times B$
4. 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 ⇒ A): C[A] ⇒ 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

1. Define a PTVF `def isLong[T]: Boolean` that returns `true` for `Long` and `Double`, returns `false` for `Int`, `Short`, and `Float`, otherwise undefined
2. Define a monoid instance for the type $\text{String} \times (1 + \text{Int})$
3. If A is a monoid and R any type, define monoid instance for $R \Rightarrow A$
4. If `S` is a semigroup then `Option[S]` is a monoid
5. Define a functor instance for `type F[T] = Future[Seq[T]]`
6. Define a Cats' `Bifunctor` instance for $B^{X,Y} \equiv (\text{Int} \Rightarrow X) + Y \times Y$
7. 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 $P^A \equiv A \Rightarrow (\text{Int} \Rightarrow A)$
8. Define a functor instance for recursive type $Q^A \equiv \text{String} + A \times Q^A$
9. * If F^A and G^A are functors, define functor instance for $F^A \times G^A$
10. * Define a functor instance for $F^A \Rightarrow G^A$ where F^A is a contrafunctor and G^A is a functor