Chapter 5: Type-level functions and type classes

Sergei Winitzki

Academy by the Bay

January 14, 2018

Motivation for type classes I: Restricting type arguments

We would like a generic `sum` implementation 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 are "summable" in some sense

We would like to define `fmap` for functors, using the available `map`

- 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 required 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 the concept of *partial functions* – applied to types

Motivation for type classes II: Partial functions on types

- 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	`def f(x: Int): Int`	`def point[A]: A ⇒ List[A]`
to type	*dependent type*	`type MyData[A] = Either[Int, A]`

- Evaluation: value-to-value – run time, type-to-value – compile time
  - if we use type casts, type-to-value can become run-time (*yuck!*)

partial function:	value-level (PVVF)	from type (PTVF)
example:	`{ case Some(x) ⇒ x-1 }`	`implicitly[T]`
when misapplied:	exception at run time	error at compile time

- Type classes provide a systematic way of managing PTVFs
  - we can apply a PTVF 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[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 in certain places, such as within `.collect()`
  - make functions total: either add code, or use *more restrictive types*
  - e.g. types such as "non-empty list", "positive number", `Some[T]`, etc.

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

- Can we restrict the *type parameter(s)* of a PTVF to a subset of types?

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]`
  - see example code
- MyTC^A is a PTTF because its values exist only for $A \in \{\text{Int}; \text{String}\}$
  - The **type domain** $\{\text{Int}; \text{String}\}$ is defined *at compile time*
  - Note: MyTC^A cannot be a functor since it is not defined for all types A
- When to use GADTs:
  - 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

Managing PTFs by hand II: Using OO method overriding

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

- A trait with `def` methods that are overridden:

```
sealed trait HasPlus[A] {
  def plus(a1: A, a2: A): A
}
final case class CaseInt() extends HasPlus[Int] {
  override def plus(a1: Int, a2: Int): Int = a1 + a2
}
final case class CaseString() extends HasPlus[String] {
  override def plus(a1: String, a2: String): String = a1 + a2
}
```

- *Similar* to having defined `plus[A]` for $A \in \{\text{Int}; \text{String}\}$
- Limitations:
  - We can only call `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 `func[A](...)` only for certain types `A`:

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

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](args)` will now become `func[A](args, ev)`
- one TE value `ev` needs to be created for *each* supported type `A`
- we now need to keep passing all these TE values around the code

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 from previous ones

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 ≥ 2 `implicit` values of the same type is a compile-time error!

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

```
def func[A: TC1, B: TC2](args...) = ...
```

- This is entirely equivalent to this longer code:

```
def func[A, B](args...)(implicit evA: TC1[A], evB: TC2[B])= ...
```

- standard library has `def implicitly[A](implicit x: A): A = x`

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-constructed 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
  - this is how we know that `func[T]` is a PTVF

Type classes II

Implementation in Scala

A type class is typically implemented as:

1. a trait with a type parameter, e.g. `trait MyTC[T]`
2. code that creates values of type `MyTC[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`
3. some functions with implicit argument(s) of type `MyTC[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

A TE value should carry all information the PTVFs need about the type `T`

- usually, the trait `MyTC[T]` contains all the PTVFs as `def` methods
- 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
- additional PTVFs (with unchanged PTTF) can be added later
  - no need to modify the code of `MyTC[T]` if the type domain is unchanged
- can combine with other PTTFs/PTVFs defined later

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
  - $\forall x : \mathtt{combine}(\mathtt{empty}, x) = \mathtt{combine}(x, \mathtt{empty}) = x$
- 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 PTTF (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

Overview: Types and kinds

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

- the **domain** of a VVF is the set of admissible argument *values*
  - a "value domain" (subset of values) is called a **type**
  - the VVF is applied safely only to argument values of the right **type**

    ```
    def f(x: Option[Int]) = ...; f(y);
    ```
- the **type domain** of a PTVF is the set of admissible argument *types*
  - a "type domain" (subset of types) is called a **kind**
  - the PTVF is applied safely only to type arguments of the right **kind**

    ```
    def func[T: MyTypeClass](args...) = ...; f[A](args);
    ```
- In both cases, the function call's safety is guaranteed *at compile time*

Kinds are the "type system for types"

- a type class `MyTypeClass` defines a new kind (as a subset of types)
  - suggested **kind** notation: $(* : \text{MyTypeClass})$
- another existing kind is the **type function** kind (notation: $* \to *$)
  - in `F[T]`, the `F` and the `T` are types of different **kinds** ($* \to *$ and $*$ resp.)
  - define `type Ap[F[_], T] = F[T]`, then wrong kinds will fail in `Ap[A, B]`
    - ★ suggested **kind** notation: $\text{Ap} : (* \to *, *) \to *$ ("higher-kinded type")
  - 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 `MyTCSyntax[T]`
- declare an *implicit conversion* function from `T` to `MyTCSyntax[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 ⇒ 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. Show: 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 $P^A \equiv A \Rightarrow (\text{Int} \times A)$
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 (use Cats' `Contravariant` type class for F^A) and G^A is a functor

Further directions

- What problems can we solve now?
  - Define arbitrary PTTFs and use them to define type classes (PTVFs)
  - Define them together or separately, combine them at will
  - Use the Cats library to define instances for standard type classes
  - Derive type class instances automatically from previous ones
  - Reason about higher-order type functions, types, and kinds as necessary
- What problems cannot be solved with these tools?
  - Automatically derive type class instances for polynomial data types
    - see The guide to "shapeless", chapter 3
  - Derive a recursive type generically from an arbitrary type function
    - Given a type function `F[_]`, define a recursive type `R` via `R = F[R]`
    - This `R` will be a function of `F`; denote that type function by `Y[F[_]]`
    - This `Y` must be defined by a type equation like this,

```
type Y[F[_]] = F[Y[F]] // does not compile ("cyclic type")
```

  - Automatically derive type class instances for such recursive types
    - That requires type-level recursion (type-level fixpoints), see matryoshka
- This and other advanced topics are found in this blog post from 2010