cats Typeclass Cheat Sheet

Adam Rosien (adam@rosien.net)

October 14, 2017

Installation

In your `build.sbt` file:

```
libraryDependencies += "org.typelevel" %% "cats-core" %
"1.0.0-MF"
```

Then in your `.scala` files:

```
import cats._
```

Defining Signatures

Each typeclass is defined by a particular function signature and a set of laws[1](invariants) that the typeclass must obey.

Typeclass	Signature				
Functor	`F[A]`	`=>`	`(A => B)`	`=>`	`F[B]`
Contravariant	`F[A]`	`=>`	`(B => A)`	`=>`	`F[B]`
Apply[2]	`F[A]`	`=>`	`F[A => B]`	`=>`	`F[B]`
FlatMap[3]	`F[A]`	`=>`	`(A => F[B])`	`=>`	`F[B]`
CoFlatMap	`F[A]`	`=>`	`(F[A] => B)`	`=>`	`F[B]`
Traverse[4]	`F[A]`	`=>`	`(A => G[B])`	`=>`	`G[F[B]]`
Foldable	`F[A]`	`=>`	`(B, (B, A) => B)`	`=>`	`B`
SemigroupK	`F[A]`	`=>`	`F[A]`	`=>`	`F[A]`
Cartesian	`F[A]`	`=>`	`F[B]`	`=>`	`F[(A, B)]`

[1] Typeclass laws are not listed here. See each typeclass' scaladoc link for more information.

[2] `Apply` has a (broader) subtype `Applicative`. See the expanded tables below.

[3] `FlatMap` has a (broader) subtype `Monad`.

[4] `Traverse` requires that the target type constructor `G` have an implicit `Applicative` instance available; that is, an implicit `Applicative[G]` must be in scope.

Informally, traversing a structure maps each value to some effect, which are combined into a single effect that produces a value having the original structure. For example, by transforming every `A` of a `List[A]` into a `Future[B]`, the traversal would return a `Future[List[B]]`.

Derived Functions

For each typeclass, its defining function is marked in **bold** and each derived function listed below it.

Typeclass			Signature			Function
Functor	`F[A]`	`=>`	`(A => B)`	`=>`	`F[B]`	**map**
		`=>`	`(A => B)`	`=>`	`F[(A, B)]`	fproduct
		`=>`	`B`	`=>`	`F[B]`	as
		`=>`	`B`	`=>`	`F[(B, A)]`	tupleLeft
		`=>`	`B`	`=>`	`F[(A, B)]`	tupleRight
		`=>`			`F[Unit]`	void
Contravariant	`F[A]`	`=>`	`(B => A)`	`=>`	`F[B]`	**contramap**
Apply[5]	`F[A]`	`=>`	`F[A => B]`	`=>`	`F[B]`	**ap**
		`=>`	`F[B] => ((A, B) => C)`	`=>`	`F[C]`	`map2`
Applicative	`F[A]`	`=>`	`F[A => B]`	`=>`	`F[B]`	**ap**
		`=>`	`Boolean`	`=>`	`F[Unit]`	unlessA
		`=>`	`Boolean`	`=>`	`F[Unit]`	whenA
		`=>`	`Int`	`=>`	`F[List[A]]`	replicateA
FlatMap	`F[A]`	`=>`	`(A => F[B])`	`=>`	`F[B]`	**flatMap**
		`=>`	`F[B]`	`=>`	`F[B]`	followedBy
		`=>`	`F[B]`	`=>`	`F[A]`	forEffect
		`=>`	`(A => F[B])`	`=>`	`F[(A, B)]`	mproduct
	`F[F[A]]`	`=>`		`=>`	`F[A]`	flatten
CoFlatMap	`F[A]`	`=>`	`(F[A]=> B)`	`=>`	`F[B]`	**coflatMap**
		`=>`		`=>`	`F[A[A]]`	coflatten

[5] Both the `Apply` and `Applicative` typeclasses implement the `ap` method; `Applicative` is a subtype of `Apply`, with an additional `pure` method to lift a value into the `Applicative`.

[6] If `B` has a `Monoid`

[7] If `A` has a `Monoid`

Issues and suggestions welcome at https://github.com/arosien/cats-cheatsheets

Typeclass			Signature			Function
Traverse	F[A]	=>	(A => G[B])	=>	G[F[B]]	**traverse**
		=>	((A, Int) => B)	=>	F[B]	mapWithIndex
		=>		=>	F[(A, Int)]	zipWithIndex
	F[G[A]]	=>			G[F[A]]	sequence
Foldable	F[A]	=>	B => ((B, A) => B)	=>	B	**foldLeft**
		=>	Eval[B] => ((A, Eval[B]) => Eval[B])	=>	Eval[B]	**foldRight**
		=>	(A => B)	=>	B	foldMap[6]
		=>			A	combineAll[7]
		=>	(A => Boolean)	=>	Option[A]	find
		=>	(A => Boolean)	=>	Boolean	exists
		=>	(A => Boolean)	=>	Boolean	forall
		=>			List[A]	toList
		=>			Boolean	isEmpty
		=>			Boolean	nonEmpty
		=>			Int	size
SemigroupK	F[A]	=>	F[A]	=>	F[A]	**combine**
Cartesian	F[A]	=>	F[B]	=>	F[(A, B)]	**product**