scala world 22 Sep 2015

Principles for approachable, modular, functional libraries

Erik Osheim (@d6**)**

scala world 22 Sep 2015

Heuristics for approachable, modular, functional libraries

Erik Osheim (@d6**)**

who am i?

- **typelevel** member λ
- maintain **spire**, **cats**, and several other scala libraries
- interested in expressiveness **and** performance ☯
- hack scala code at **meetup**

code at **http://github.com/non**

what is this talk about?

We've been writing FP in Scala for awhile. We have:

- *Functional Programming in Scala* ("the red book")
- Businesses powered by FP
- Libraries to support FP
- Many many blog posts
- Smart people happy to talk about FP on Gitter/IRC/etc.
- Lots of gists, examples, and proofs-of-concept

what is this talk about?

Usually our talks are designed for outreach.

- **Education** - introducing new FP concepts
- **Evangelism** - advocacy, debate, high-level analysis
- **Encouragement** - success stories, positive reinforcement

These are all important!

what is this talk about?

But there are difficulties.

- "map is not a member of F[A]"
- `StackOverflowException`
- <snarky comment about the "cinnabon" operator>
- `232.188: [Full GC`
- SI-2712

In response, our libraries continue to evolve and adapt.

what is this talk about?

heuristic, /ˌhjʊ(ə)ˈrɪstɪk/

"In psychology, heuristics are simple, efficient rules, learned or hard-coded by evolutionary processes..." [1]

[1] https://en.wikipedia.org/wiki/Heuristic

Our libraries and the Scala language are co-evolving.

what is this talk about?

This talk is focused on *encoding* FP concepts in Scala.

Cats is used as a case study.

But these ideas apply to your libraries as well.

This is a hard problem.

cats

CATS

cats

CATS

Category-theoretic **A**lgorithms and **T**ools for **S**cala?

cats

CATS

Category-theoretic **A**lgorithms and **T**ools for **S**cala?

(No.)

cats

`Lightweight, modular, and extensible library for functional programming.`

- Created 28 Jan 2015
- 45 contributors
- 174 issues
- 363 pull requests
- 1174 commits

AUTHORS.md

Alissa Pajer
Alistair Johnson
Amir Mohammad Saied
Andrew Jones
Arya Irani
Benjamin Thuillier
Bobby
Brendan McAdams
Cody Allen
Colt Frederickson
Dale Wijnand
Dave Rostron
David Allsopp
Derek Wickern
Erik Osheim
Eugene Burmako
Eugene Yokota
Feynman Liang
Frank S. Thomas
Jisoo Park
Josh Marcus
Julien Richard-Foy
Julien Truffaut
Kenji Yoshida
Luis Angel Vicente Sanchez
Marc Siegel
Michael Pilquist
Mike Curry
Owen Parry
Pascal Voitot
Philip Wills
Rintcius
Rob Norris
Romain Ruetschi
Ross A. Baker
Sinisa Louc
Stephen Judkins
Stew O'Connor
Travis Brown
Wedens
Yosef Fertel
Zach Abbott

goals

1. Functional
2. Safe
3. Fast
4. Documented
5. Modular
6. Idiomatic
7. Pragmatic
8. Collaborative
9. Welcoming

goals

Too many goals?

☁

Really, there is only one:

"Remove barriers to doing functional programming in Scala."

☀

That's it.

background

my own journey

My claim is that we are still learning how to do this.

(Or at last I am.)

I started learning Scala in August 2011.

My experience comes primarily from working on Spire.

spire

- **3 Nov 2011** - first Spire commit
- **15 Jan 2012** - add benchmarks
- **25 Jul 2012** - name-based ops-macros (i.e. machinist)
- **22 Jan 2013** - use discipline, publish laws
- **19 Feb 2015** - add scoverage plugin
- **20 May 2015** - add scalastyle plugin
- **24 Jul 2015** - scala.js support

spire

Summary:

- A journey of almost 4 years.
- Some practices adopted very early (e.g. benchmarking)
- Others much later (e.g. code coverage)
- Major changes within the last few months.
- Many more changes on the horizon.

spire

Things we learned:

- Seemingly-insignificant costs will add up.
- If you care about performance, benchmark.
- Property-based testing is amazing; export your laws!
- If you care about testing, measure code coverage.
- `.equals`, `any2stringadd`, etc. are evil.
- `@specialized` is fragile, so design for it up-front.

spire

Setbacks/mistakes:

- Monolithic project
- Purpose of types/type classes not always apparent
- Lack of high-quality documentation
- Somewhat difficult to get involved
- Small (but dedicated) team

spire

Future plans:

- Modularize via **algebra**, **spire-extras**
- Improve internal documentation
- Tutorials and more examples
- Outreach and blogging
- Better JS support

barriers

barriers

Cats aims to remove barriers preventing people from doing functional programming in Scala.

⚠

What are those barriers?

technical barriers

What are they?

1. Type classes not first class in Scala
2. Standard Library exists
3. Hotspot not designed for optimizing FP code
4. Huge language surface area, many possible encodings
5. Impedance mismatches with ML, Haskell, etc.
6. **Complexity**

technical barriers

How to overcome them?

1. Build-your-own-runtime
2. Canonical encodings and aggressive subsetting
3. Opinionated style guidelines/style-checking
4. Benchmarking and testing
5. Macros, compiler plugins, and dark magic
6. **Willingness to break new ground**

barriers at work

What are they?

1. Perceptions of difficulty
2. Performance concerns
3. Stability concerns
4. Unfamiliar types, syntax, etc.
5. "Scarcity" of experience functional programmers
6. **Fear of the Unknown**

barriers at work

How to overcome them?

1. Tutorials and examples (which **work**!)
2. Documentation, tests, benchmarks, and profiling
3. Stand firm on important principles
4. Compromise on incidental details
5. Accept responsibility for education
6. **Provide high-quality libraries**

social barriers

What are they?

1. Perception of cliques
2. Imposter syndrome and delayed feelings of mastery
3. Powerful-but-confusing libraries
4. Wealth of alternatives
5. Concerns about harassment/bad behavior
6. **"This is not for me"**

social barriers

How to overcome them?

1. Model good technical and social practices
2. Reach out and welcome newcomers
3. Acknowledge the limits of our own knowledge
4. Provide opportunities for new work
5. Accept responsibility for education and codes of conduct
6. **Foster a stable, long-term, supportive community**

barriers tl;dr

We should:

- Be willing to break new ground, and
- Provide high-quality libraries, to
- Foster a stable, long-term, supportive community

goals

1. Functional λ
2. Safe ♖
3. Fast ⛧
4. Documented ⚿
5. Modular ⚙
6. Idiomatic ⮁
7. Pragmatic ⚖
8. Collaborative ⚛
9. Welcoming ☀

functional λ

1. Use type classes for open interfaces
2. Encourage/require laws for type classes/data types
3. Encourage/require referential transparency (i.e. purity)
4. First-class laziness support
5. Encourage composition
6. Support abstraction, minimize duplication (DRY)
7. Watch other libraries, languages, and researchers

safe ♖

1. Stack-safe by default
2. Express partiality with values not exceptions
3. Minimize allocations where possible
4. Property-based testing
5. Aim for 100% test coverage
6. Benchmarking and "real world" tests

fast ⛧

1. Avoid ineffecient default implementations
2. Do benchmarking and profiling to measure behavior
3. Perform manual/mechanical optimizations if possible
4. Design "low-level" abstractions with performance in mind
5. Make common syntax/usage efficient

documented 🔑

1. Require Type-checked tutorials and documentation
2. Require ScalaDoc on methods/classes
3. Create documents for users, contributors, maintainers.
4. Err on the side of more prose rather than less.
5. Link to the literature where possible.
6. Major design discussions should occur on Github/Gitter.

modular ⚙

1. Platform independent by default (`cats.jvm`/`cats.js`)
2. Non-monolithic, limited scope
3. Extensible, extensibility hooks
4. Publish laws and test-hooks for third-parties
5. Move small, useful utilities to standalone projects
6. Use / support other libraries

idiomatic

1. Explicitly mark/differentiate type classes
2. Minimize boilerplate where possible
3. Support built-in Scala types by default
4. Discuss and enforce common coding conventions
5. Consider and standardize on language extensions

pragmatic ⚖

1. Projects may "dial-in" their strictness (e.g. `alleycats`)
2. Allow specific, thoughtful exceptions (e.g. `Double`)
3. Aggressively listen to and respond to "pain points"
4. Willing to specialize the critical path
5. All features should have concrete motivating examples

collaborative

1. Give many people authority (11 maintainers)
2. Provide quick feedback on issues and PRs
3. Require 2 sign-offs on PR merging
4. Encourage discussion on issues/PRs
5. Leave opportunities for newcomers when possible

welcoming ☀

1. Be clear on expectations around conduct
2. Try to model productive and helpful behavior
3. Work to aim documentation towards newcomers
4. Reach out to the wider programming community

unsolved problems

unsolved problems

1. Project governance model
2. Type class inheritance inhibits modularity
3. Importing/masking implicits across multiple projects
4. Unknown unknowns

unsolved problems

"Project governance model"

- Ratifying new maintainers
- Beyond consensus decision-making
- Other formal processes

unsolved problems

"Type class inheritance inhibits modularity"

- Hard to add "earlier/looser" type classes
- Interacts poorly with limited scope/non-monoliths

unsolved problems

"Importing/masking implicits across multiple projects"

- Implicit scope is fraught
- Bulk imports are nice (but break modularity)
- Fine-grained imports are too fiddly

unsolved problems

In addition to the things I know are unsolved problems, there are likely problems I don't even know about:

- What do you think?
- What are we overlooking?

evolution

evolutionary projects

- **kind-projector** - type lambda syntax support
- **simulacrum** - type class annotations/codegen
- **machinist** - faster implicit ops classes
- **catalysts** - tools for platform independence (js/jvm)
- **export-hook** - pluggable type class derivation
- **imp** - quickly summon implicit instances

related projects

related projects

One goal of modularity is leaving room for other projects.

```
    (Flotilla of boats) > (one gigantic ship)
```

related projects

These projects interoperate with Cats:

* Algebra	generic abstract algebra
* Alleycats	outlaw type classes / instances
* Kittens	derived type class instances
* Dogs	data structures for pure FP
* Circe	pure functional JSON (argonaut port)
* (Monocle)	optics and lenses
* ((Spire))	fast, precise, generic numerics
* (((FS2)))	next generation of scalaz-stream

(Parens are for possible future support.)

recommendations

So you want to create an FP library in Scala?

Great!

☎

As you have seen, I have **Opinions™**.

recommendations

Here's a list of concrete recommendations.

Most of these have been used in Cats.

Many of them are works-in-progress.

Good luck!

recommendations

Use `SBT` with (at least) the following subprojects:

- **core** - the stuff your users need
- **laws** - laws/tests they need to test their code
- **tests** - run your tests, use your laws
- **docs** - compile and publish docs
- **benchmark** - run performance tests

(**core** and **laws** would be published as `.jar` files.)

recommendations

Support Javascript.

There is a ton of energy around **scala-js** [1].

The potential to grow our community this way is *HUGE*!

☺

(Thanks to Alistair Johnson for opening my eyes.)

[1] http://www.scala-js.org/

recommendations

Platform independence can actually help our design:

- Avoid baking-in ugly JVM cruft
- Avoid blocking APIs
- Avoid pattern-matching on JVM types
- FP in Scala is a great model for JS

Easier to get this right "up front" rather than retrofitting things into the design (although retrofitting can be done).

recommendations

You should do property-based testing.

- **laws** should depend on a library for property-based testing
- **laws** should export laws (properties) to be tested
- **laws** should export generators for interesting data types

(**laws** is seprate to minimize **core** dependencies.)

recommendations

You can use **scalacheck** [1] to do property-based testing.

There's also **scalaprops** [2], an excellent new library.

⛓

[1] http://github.com/rickynils/scalacheck
[2] https://github.com/scalaprops/scalaprops

recommendations

You can use **discipline** [1] to define your laws.

The previously-mentioned **scalaprops** [2] can also do this.

Both libraries help you to avoid unnecessary test duplication.

[1] https://github.com/typelevel/discipline
[2] https://github.com/scalaprops/scalaprops

recommendations

When possible, defer equality checks in your laws.

This restriction makes them more powerful.

♗

e.g. We can try to falsify that two `(A => A)` instances are equal.

recommendations

```
case class IsEq[A](x: A, y: A)

trait FunctorLaws[F[_]] {
  implicit override def F: Functor[F]

  def id[A](fa: F[A]): IsEq[F[A]] =
    fa.map(identity) <-> fa

  def comp[A, B, C](fa: F[A], f: A => B, g: B => C): IsEq[F[C]] =
    fa.map(f).map(g) <-> fa.map(f andThen g)
  }
}
```

recommendations

Measure test coverage.

You can set up **sbt-scoverage** [1] to do this.

☐ ☑ ☒

You can also look into setting up `codecov.io` [2].

[1] https://github.com/scoverage/sbt-scoverage
[2] https://codecov.io/github/non/cats?branch=master

recommendations

You should create beautiful, type-checked tutorials.

- **docs** aggregates all your projects
- **docs** should type-check your documenation
- **docs** might publish your docs too!

recommendations

You should use **tut** [1] to type-check Markdown files.

- Prevents broken docs post-refactor
- Ensures all the imports are included (!)
- Provide narrative, long-form documentation
- Real world examples

☥

[1] https://github.com/tpolecat/tut

recommendations

You should also created unified docs across your subprojects.

You can use **sbt-unidoc** [1] to do this.

☝️

[1] https://github.com/sbt/sbt-unidoc

recommendations

You should consider setting up publishing from SBT.

♧

If you use Github pages, you can use both:

- **sbt-ghpages** - https://github.com/sbt/sbt-ghpages
- **sbt-site** - https://github.com/sbt/sbt-site

recommendations

You should benchmark your code.

- Get a ballpark idea of performance
- Ensure that design satisfies requirements
- Track relative effect of changes

recommendations

You should use **sbt-jmh** [1] to do this.

You could also use **scalameter** [2], **thyme** [3], **caliper** [4], etc.

(You almost certainly should not hand-code benchmarks.)

♨

[1] https://github.com/ktoso/sbt-jmh
[2] https://scalameter.github.io/
[3] <unpublished>
[4] <semi-unpublished>

recommendations

You should use **scalastyle** [1] to help enforce consistency.

In order to effectively create convention (and subset the Scala language) we need consistency.

(See also **wart-remover**, **abide**, etc.)

✂

[1] http://www.scalastyle.org/

(In retrospect I should have created an sbt-plugin.)

recommendations

You should create a code of conduct for your project.

Cats (and many other projects) use the Typelevel CoC [1].

☤

[1] http://typelevel.org/conduct.html

recommendations

Try to limit the scope of your project.

Encourage others to create their own projects/modules, and work to help them interoperate.

✍

(This is a great response to large feature requests.)

recommendations

Use a sign-off process, even if it starts as a rubber stamp.

Shows potential contributors what to expect.

✌️

(It also helps establish good habits amongst maintainers.)

recommendations

Try to respond to questions, issues, PRs, in a timely manner.

(This can be hard.)

- In exchange for time/encouragement up front
- Author of a tiny bug fix may become prolific contributor
- (Or even a co-maintainer)
- This will ultimately save you tons of time

recommendations

Try to set a friendly, encouraging tone for discussion.

- Everyone is trying to learn more
- People will make mistakes (including you!)
- Corrections should be clear but non-hostile
- Not a place to compete for *"alpha nerd"* dominance

Internet communication is hard.

Be conservative in what you say, but liberal in what you accept.

recommendations

Obviously these are just recommendations.

We all have limited time/energy.

There will be trade-offs and no one is perfect.

But I think these are all worth striving for.

appendices

simulacrum

Type classes aren't a first class concept in Scala.

We recognize type class **encodings**.

1. Requires practice
2. Encodings differ
3. *Type classes, members, instances, Ops, oh my!*
4. Difficult for newcomers
5. No bread crumbs

simulacrum

Use an annotation to signal a trait is a type class.

You can write:

```
import simulacrum._
@typeclass trait Semigroup[A] {
  @op("|+|") def append(x: A, y: A): A
}
```

...which expands into...

simulacrum

```
trait Semigroup[A] {
  def append(x: A, y: A): A
}

object Semigroup {
  def apply[A](implicit instance: Semigroup[A]): Semigroup[A] = instance

  trait Ops[A] {
    def typeClassInstance: Semigroup[A]
    def self: A
    def |+|(y: A): A = typeClassInstance.append(self, y)
  }

  trait ToSemigroupOps {
    implicit def toSemigroupOps[A](target: A)(implicit tc: Semigroup[A]): Ops[A] = new Ops[A] {
      val self = target
      val typeClassInstance = tc
    }
  }

  trait AllOps[A] extends Ops[A] {
    def typeClassInstance: Semigroup[A]
  }

  object ops {
    implicit def toAllSemigroupOps[A](target: A)(implicit tc: Semigroup[A]): AllOps[A] = new AllOps[A] {
      val self = target
      val typeClassInstance = tc
    }
  }
}
```

kind-projector

Extend Scala to support a "type lambda" syntax.

You can write:

```
Functor[Either[E, ?]]
```

instead of:

```
Functor[({type L[x] = Either[E, x]})#L]
```

implicit ops classes

For "fast" implicit operators, overhead is significant.

```
def fast[A](x: A)(implicit ev: Foo[A]): A =
  ev.foo(x)

implicit class FooOps[A](lhs: A) extends AnyVal {
  def foo(implicit ev: Foo[A]): A = ev.foo(lhs)
}

def slow[A: Foo](x: A): A = x.foo
```

Use **machinist**.

implicitly is slow

```
def fast[A](implicit ev: Monoid[A]): A =
  ev.empty

def slower[A: Monoid]: A =
  implicitly[Monoid[A]].empty

def alsoSlower[A: Monoid]: A =
  Monoid[A].empty // without imp
```

Use **imp** to speed things up.

by-name params are not ideal

- Type system usually doesn't distinguish `A` and `(=> A)`.
- Methods cannot return `(=> A)`.
- `(=> A)` is silently evaluated to `A` when "needed"
- `(=> A)` is not memoized
- `(=> A)` never releases references it captures
- `(=> A)` unconditionally allocates a `Function0[A]`

Use `Eval[A]`.

by-name params are not ideal

Exception: using `(=> A)` for syntax is fine.

```
import cats._

def cond[A](b: => Boolean, t: => A, f: => A): Eval[A] =
  Later(b).flatMap { test =>
    if (test) Later(t) else Later(f)
  }
```

The `Later(_)` constructor wraps `(=> A)`, adds memoization, frees the thunk when possible, etc.

conclusions

conclusions

- None of this is easy.
- We're still learning the "best" way to do FP in Scala.
- (But we have some good heuristics.)
- Social and professional barriers to FP in Scala are real.
- We're all in this together.

Thank you.

the end