Free Monads and Free Applicatives

Markus Hauck, codecentric AG [@markus1189]

What this talk is about

- Write dsl using free monad / free applicative
- Explore the specific pros and cons
- Show how to optimize with free applicatives
- Combine free monads and free applicatives

Before We Start

- Requirements:
  - functional programming library of your choice
  - we will use *cats*, could have used *scalaz* as well
- Plan: write a Dsl to query GitHub
  1. using free monads
  2. using free applicatives
  3. using both

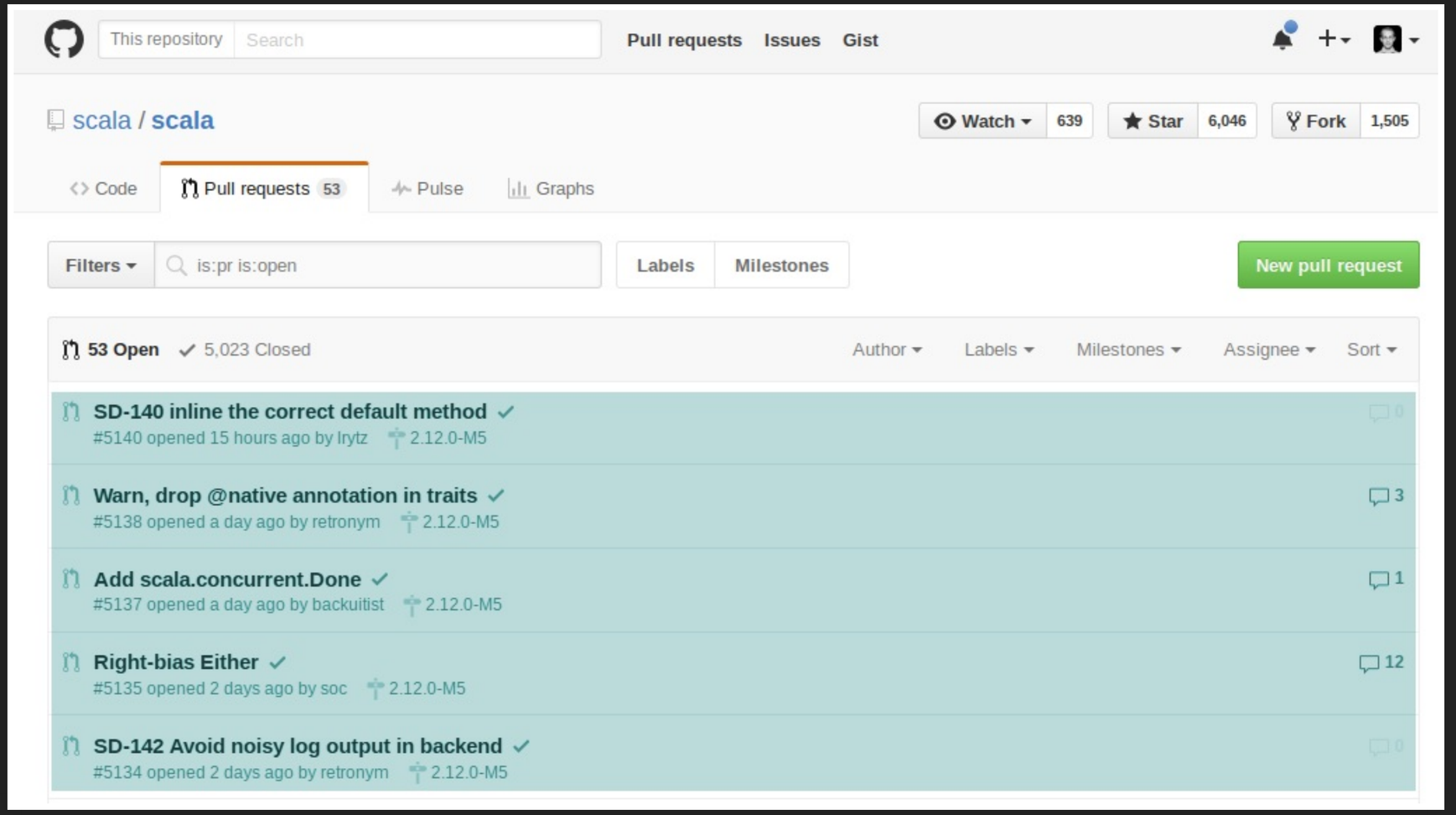
Before We Start - kind projector

- compiler plugin: kind projector
- allows nicer syntax for type lambdas
- so if you wonder about weird syntax like this:

```
λ[α => Int]  // a => 42
Option[?]    // Some(_)
```

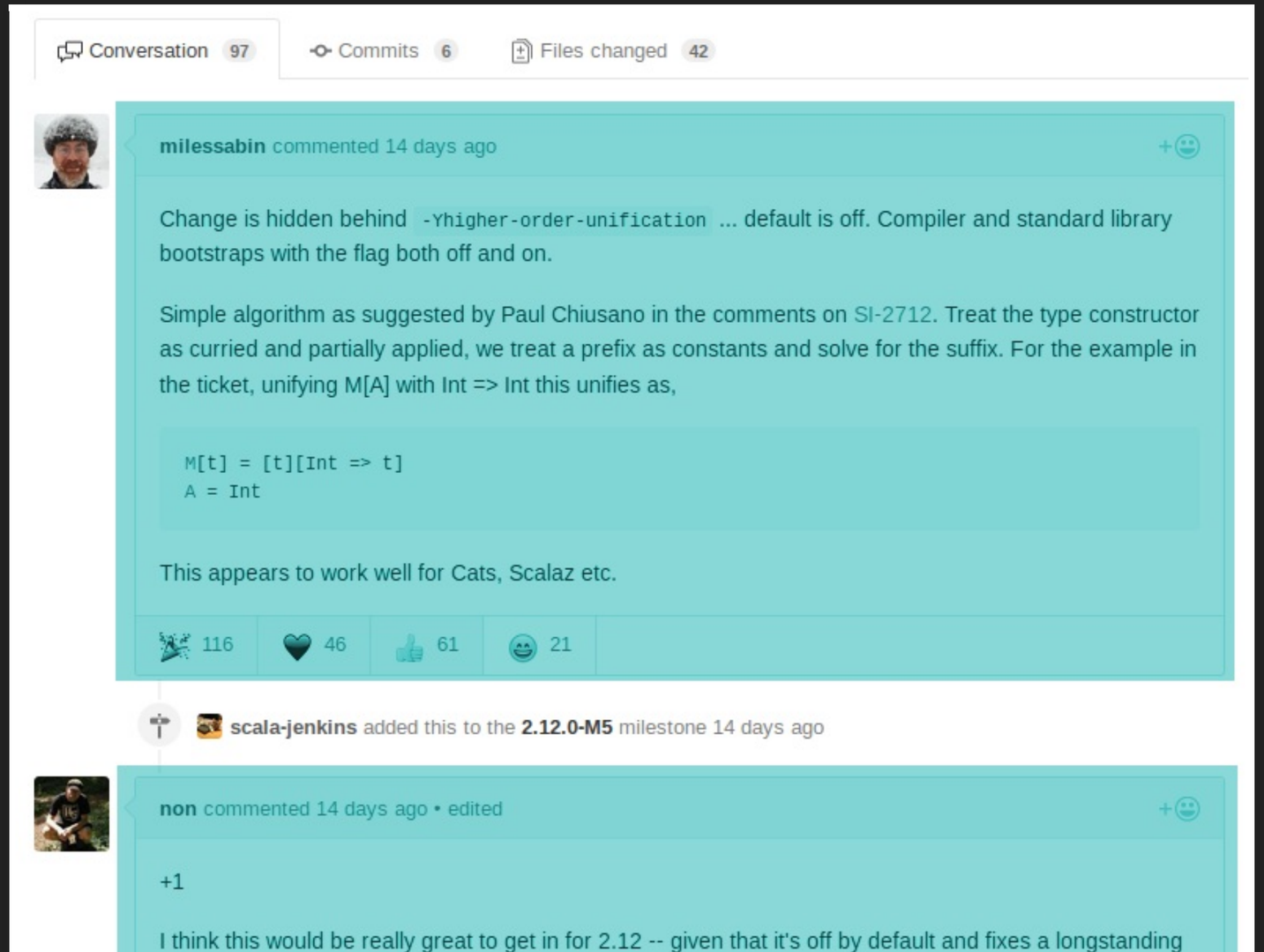
GitHub API - Issues

Given an owner and a repo, list issues



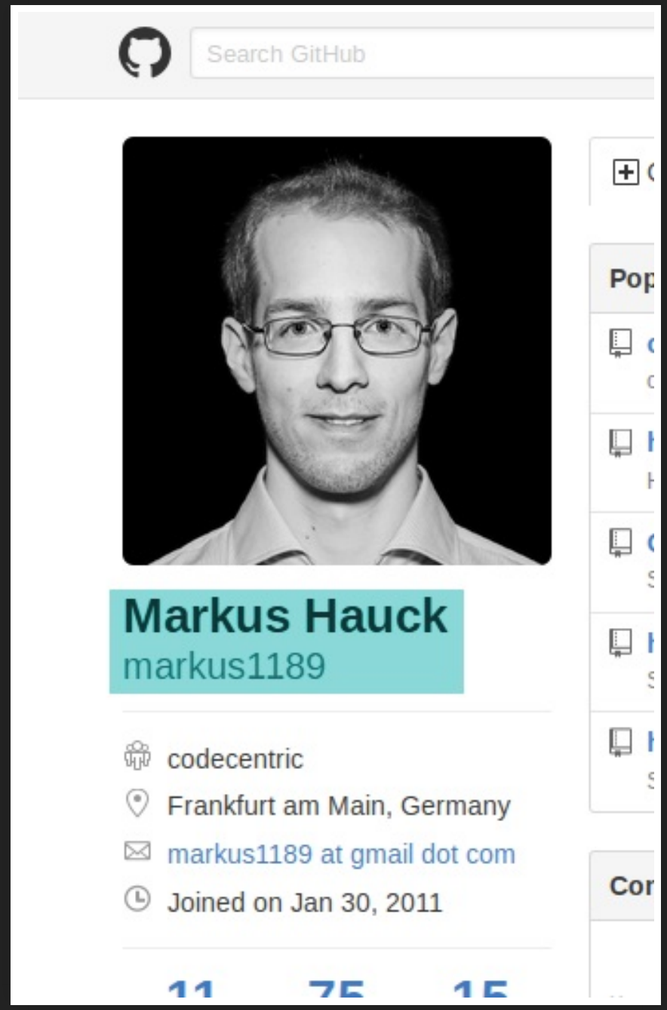
GitHub API - Comments

Given an issue, find all comments



GitHub API - Full Names

Finally, retrieve the full name (if given) from a login



Why free monads/applicatives at all?

```
def listIssues(owner: Owner,
               repo: Repo
              ): Future[List[Issue]] =

def getComments(owner: Owner,
                repo: Repo,
                issue: Issue
               ): Future[List[Comment]] = ???

def getUser(login: UserLogin
           ): Future[User] = ???
```

- limited to one interpretation
- hard to test without hitting network
- no control from dsl author perspective

The Start: Free Monads

- free monads allow all of this
  - not covered: different interpretation targets, e.g. network/pure
- free monads offer simple way to construct your Dsl:
  1. write your *instructions* as an ADT
  2. use *Free* to lift them into the free monad
  3. write an interpreter

1. Write your *instructions* as an ADT

```
sealed trait GitHub[A]

case class ListIssues(owner: Owner, repo: Repo)
  extends GitHub[List[Issue]]

case class GetComments(owner: Owner, repo: Repo, issue: Issue)
  extends GitHub[List[Comment]]

case class GetUser(login: UserLogin)
  extends GitHub[User]
```

2. Use *Free* to lift instructions

```
type GitHubMonadic[A] = Free[GitHub, A]

def listIssues(owner: Owner,
               repo: Repo
              ): GitHubMonadic[List[Issue]] =
  Free.liftF(ListIssues(owner,repo))

def getComments(owner: Owner,
                repo: Repo,
                issue: Issue
               ): GitHubMonadic[List[Comment]] =
  Free.liftF(GetComments(owner, repo, issue))

def getUser(login: UserLogin
            ): GitHubMonadic[User] =
  Free.liftF(GetUser(login))
```

3. write an interpreter

For interpretation *Free* has a method called *foldMap*:

```
sealed abstract class Free[S[_], A] {
  final def foldMap[M[_]:Monad](f: S ~> M): M[A] = ???
}
```

- twiddly arrow = natural transformation *GitHub* to target type constructor

```
type Target[A]
val interpret: GitHub ~> Target = ???
```

3. write an interpreter

```
def step(client:Client) : GitHub ~> Future =
  new (GitHub ~> Future) {

    def apply[A](fa: GitHub[A]): Future[A] = fa match {

      case ffa @ GetComments(_, _, _) =>
        client.fetch(Endpoint(ffa)).map(parseComment)

      case ffa @ GetUser(_) =>
        client.fetch(Endpoint(ffa)).map(parseUser)

      case ffa @ ListIssues(_,_) =>
        client.fetch(Endpoint(ffa)).map(parseIssue)
    }
  }
```

(4.) Write programs!

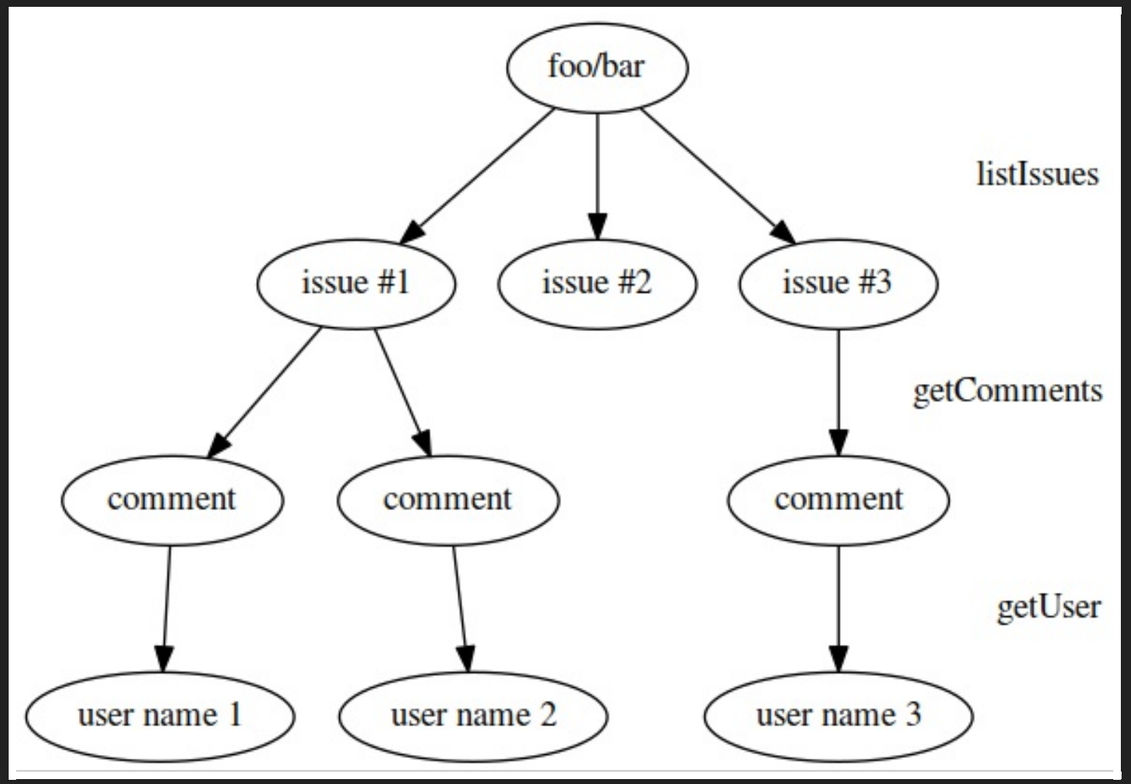
```
// 1) retrieve issues
// 2) retrieve comments from each issue
// 3) retrieve full name from each comments login name
def allUsers(owner: Owner, repo: Repo):
  GitHubMonadic[List[(Issue,List[(Comment,User)])]] = for {

  issues <- listIssues(owner,repo)

  issueComments <- issues.traverseU(issue =>
    getComments(owner,repo,issue).map((issue,_)))

  users <- issueComments.traverseU { case (issue,comments) =>
    comments.traverseU(comment =>
      getUser(comment.user).map((comment,_))).map((issue,_))
  }
} yield users
```

Almost done

- predefined functions based on monads:
  - ifM, whileM, groupM, ...
- DSL looks good, but what about:
  - parallelism?
  - efficiency? (API is rate limited!)
- let's (try to) see how our program executes

```
[info] GetUser(UserLogin(performantdata))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(janekdb))
[info] GetUser(UserLogin(FelAl))
[info] GetUser(UserLogin(FelAl))
[info] GetUser(UserLogin(Ichoran))
[info] GetUser(UserLogin(FelAl))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(FelAl))
[info] GetUser(UserLogin(janekdb))
[info] GetUser(UserLogin(FelAl))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(soc))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(soc))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(soc))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(soc))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(retronym))
[info] GetUser(UserLogin(lrytz))
[info] GetUser(UserLogin(sjrd))
```

Sequential execution all the way



Recap of Free Monads

- quite popular and many tutorials
- very expressive
- forced *sequential* execution :(
- *no* static analysis beyond first instruction :(

Free Applicatives

- problem: monads are too powerful
- applicatives < monads
- let's try free applicatives then!
  1. write your *instructions* as an ADT (✓)
  2. use *FreeApplicative* to lift them into the free applicative functor
  3. write an interpreter (✓)

Free Applicatives - Smart Constructors

```
type GitHubApplicative[A] = FreeApplicative[GitHub, A]

def listIssues(owner: Owner,
               repo: Repo
              ): GitHubApplicative[List[Issue]] =
  FreeApplicative.lift(ListIssues(owner,repo))

def getComments(owner: Owner,
                repo: Repo,
                issue: Issue
               ): GitHubApplicative[List[Comment]] =
  FreeApplicative.lift(GetComments(owner, repo, issue))

def getUser(login: UserLogin
            ): GitHubApplicative[User] =
  FreeApplicative.lift(GetUser(login))
```

Free Applicatives - Write Programs

```
// get issues from codecentric/foo and codecentric/bar
val issuesFooBar: GitHubApplicative[List[Issue]] =
    (listIssues(Owner("codecentric"),Repo("foo"))
      |@|
      listIssues(Owner("codecentric"),Repo("bar"))
    ).map(_++_)

// get full names from a list of logins
val logins: GitHubApplicative[List[User]] =
  List(UserLogin("markus1189"), UserLogin("..."), ???).
    traverseU(login => getUser(login))
```

- use applicative builder, traverse, ..., to combine results

Free Applicatives - Execution

- programs written using *GitHubApplicative* can be executed in parallel
- for *Future*, the *Applicative* instance does this already
- traversing a list of 1000 logins happens in parallel (limited only by concrete implementation of the http client)

Free Applicatives - Execution

```
List("scala","scala-dev","slip","scala-lang").
  traverseU(repo =>
    listIssues(Owner("scala"),Repo(repo))).
      map(_.flatten)
```

```
[info] Loading global plugins from /home/markus/.sbt/0.13/plugins
[info] Loading project definition from /home/markus/repos/topics/flatmap-oslo/project
[info] Set current project to flatmap-oslo (in build file:/home/markus/repos/topics/flatmap-oslo/)
[info] Running de.codecentric.github.ApplicativeDsl
```

Free Applicatives - Static Analysis

- parallel execution sure is a nice feature
- free applicatives allow static analysis of programs
- when fetching full names of users, why perform multiple requests for same login?
- idea: eliminate duplicate requests (rate limiting)

Free Applicatives - Eliminate Duplicates

- method on *FreeApplicative* called *analyze*:

```
sealed abstract class FreeApplicative[F[_], A] {
  def analyze[M:Monoid](
    f: F ~> λ[α => M]): M = ???
}
```

- translate one of our instructions into type that forms a monoid

(type level lambda syntax: kind projector)

Free Applicatives - Eliminate Duplicates

```
val logins: GitHub ~> λ[α=>Set[UserLogin]] = {
  new (GitHub ~> λ[α=>Set[UserLogin]]) {
    def apply[A](fa: GitHub[A]): Set[UserLogin] = fa match {
      case GetUser(u) => Set(u)
      case _ => Set.empty
    }
  }
}

def extractLogins(p: GitHubApplicative[_]): Set[UserLogin] =
  p.analyze(logins)
```

- we collect logins into a *Set[UserLogin]*

Free Applicatives - Improved Interpreter

```
def precompute[A,F[_]:Applicative](
  p: GitHubApplicative[A],
  interp: GitHub ~> F
): F[Map[UserLogin,User]] = {
  val userLogins = extractLogins(p).toList

  val fetched: F[List[User]] =
    userLogins.traverseU(getUser).foldMap(interp)

  Functor[F].map(fetched)(userLogins.zip(_).toMap)
}
```

Free Applicatives - Improved Interpreter

```
def optimizeNat[F[_]:Applicative](
  mapping: Map[UserLogin,User],
  interp: GitHub ~> F
): GitHub ~> F = new (GitHub ~> F) {
  def apply[A](fa: GitHub[A]): F[A] = fa match {
    case ffa@GetUser(login) =>
      mapping.get(login) match {
        case Some(user) => Applicative[F].pure(user)
        case None => interp(ffa)
      }
    case _ => interp(fa)
  }
}
```

Free Applicatives - Improved Interpreter

- now we can define a better interpreter:

```
// val interpret: GitHub ~> Future
def interpretOpt[A](p: GitHubApplicative[A]): Future[A] = {
  val mapping: Future[Map[UserLogin,User]] = precompute(p,interpre

  mapping.flatMap { m =>
    val betterNat = optimizeNat(m,interpret)
    p.foldMap(betterNat)
  }
}
```

Free Applicatives - Improved Interpreter

- we saw one possible optimization, also:
  - group single comment requests for each owner/repo, reduce calls
  - instead of executing all in parallel, limit to max factor
  - and more
- so, that's why Free Applicatives are cool!

Done?

Free Applicatives - The Catch

```
listIssues:  (Owner,Repo)       => GitHubApplicative[List[Issue]]
getComments: (Owner,Repo,Issue) => GitHubApplicative[List[Comment]]

val is: GitHubApplicative[List[Issue]] =
  listIssues(Owner("typelevel"),Repo("cats"))

val cs = GitHubApplicative[GitHubApplicative[List[Comment]]]
  is.map(_.map(getComments(Owner("typelevel"),Repo("cats"),_)))
```

- applicatives only allow lifting of *pure* functions
- we need monadic bind/join even for our small program!

Free Monads - Composition

- Free monads work over functors
- functors can be combined in many ways, e.g.:
  - nesting of two functors is a functor
  - product of two functors is a functor
  - **coproduct** of two functors is a functor
  - ...

```
final case class Coproduct[F[_],G[_],A](run: F[A] Xor G[A]) {
  ???
}
```

Free Monads - Composition

- instructions: either **GitHub** or **GitHubApplicative**

```
type GitHubBoth[A] =
  Free[Coproduct[GitHub,GitHubApplicative,?],A]

// lifting for listIssues, getComments, ... => boring

def embed[A](p: GitHubApplicative[A]): GitHubBoth[A] =
  Free.liftF[Coproduct[GitHub,GitHubApplicative,?],A](
    Coproduct.right(p))
```

Free Monads - TODO

- let's have a look at our TODOs
  1. write your *instructions* as an ADT (✓)
  2. use *Free* to lift instructions (✓)
  3. write an interpreter (almost)

Free Monads - Interpretation

- have all pieces already, just some glue needed

```
type GitHubBoth[A] =
  Free[Coproduct[GitHub,GitHubApplicative,?],A]

val interp: Coproduct[GitHub,GitHubApplicative,?] ~> Future
```

we have

```
GitHub ~> Future                  // this is just `interpret`
GitHubApplicative ~> Future       // e.g. `interpretOpt`
def interpretOpt[A](p: GitHubApplicative[A]): Future[A]
```

Free Monads - Interpretation

- **NaturalTransformation** defines a helpful method

```
trait NaturalTransformation[F[_],G[_]] {
  def or[H[_]](h: H ~> G): Coproduct[F, H, ?] ~> G = ???
}

type GitHubCo[A] = Coproduct[GitHub,GitHubApplicative,?]

val interpretMix: GitHubCo ~> Future =
  interpret.or[GitHubApplicative](interpretOptNat)

def run(p: GitHubBoth[A]): Future[A] = p.foldMap(interpretMix)
```

Revisiting our Program - Previously

```
def allUsers(owner: Owner, repo: Repo):
  GitHubMonadic[List[(Issue,List[(Comment,User)])]] = for {

  issues <- listIssues(owner,repo)

  issueComments <-
    issues.traverseU(issue =>
      getComments(owner,repo,issue).map((issue,_)))

  users <-
    issueComments.traverseU { case (issue,comments) =>
      comments.traverseU(comment =>
        getUser(comment.user).map((comment,_))).map((issue,_))
  }
} yield users
```

Revisiting our Program - Now

```
def allUsersM(owner: Owner, repo: Repo):
  GitHubBoth[List[(Issue,List[(Comment,User)])]] = for {

  issues <- listIssuesM(owner,repo)                              // M

  issueComments <- embed {                                       // M
    issues.traverseU(issue =>                                    // A
      getComments(owner,repo,issue).map((issue,_)))              // A
    }
  users <- embed {                                               // M
    issueComments.traverseU { case (i,comments) =>               // A
      comments.traverseU(comment =>                              // A
        getUser(comment.user).map((comment,_))).map((i,_))       // A
      }                                                          // M
    }
} yield users
```

Moment of truth

- let's see if all of our work pays off
- what do we expect:
  - parallel execution where possible
  - no duplicate requests for logins

```
[info] Loading global plugins from /home/markus/.sbt/0.13/plugins
[info] Loading project definition from /home/markus/repos/topics/flatmap-oslo/project
[info] Set current project to flatmap-oslo (in build file:/home/markus/repos/topics/flatmap-oslo/)
[info] Running de.codecentric.github.MixedDsl
```

Revisiting our Program - Execution

- convenient: mostly reuse existing parts
- result: an interpreter that
  - executes each "stage" in parallel
  - avoids duplicate requests for user names

Summary

- small DSL querying web API
- compare free monads and free applicatives
- parallel execution & static analysis
- regain monadic expressiveness
- combine everything into an interpreter

Thank you!

codecentric