# Cancelable IO

**Alexandru Nedelcu**

@alexelcu | alexn.org | oriel.io

# CATS-EFFECT

- Typelevel Project
  - Submitted Apr 20, 2017
  - Graduated Mar 14, 2018
- Integrated into:
  - FS2, Monix, Http4s
  - Eff, Doobie, ...

# CATS-EFFECT

- Typelevel Project
  - Submitted Apr 20, 2017
  - Graduated Mar 14, 2018
- Integrated into:
  - FS2, Monix, Http4s
  - Eff, Doobie, ...

# CATS-EFFECT

- 1.0.0-RC
- 1.0.0-RC2 (soon)

# CATS-EFFECT HISTORY

- Beginning 2017:
  - Monix had a **Task**
  - FS2 had a **Task**
  - Scalaz 7 had its own **Task**
- Libraries like Http4s and Doobie had to pick one

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

# New Project: Typelevel Schrodinger #66

Edit | New issue

Closed alexandru opened this issue on Mar 29, 2017 · 62 comments

alexandru commented on Mar 29, 2017 • edited

Member

I have a proposal for a new project, called *Typelevel Schrodinger*.

What this project aims to do is to provide a common interface between various `Task` and `IO` like data-types and would basically be what Reactive Streams is for streaming, the purpose being to allow interoperability between various libraries.

I started an initial draft here, but would be cool if we moved this to the Typelevel organization and collaborate on it:

https://github.com/alexandru/schrodinger

## Summary / Current Proposal (Updated: Apr 7, 2017 23:01)

1. we start project `schrodinger-core` which:

- provides `Evaluable`, `Deferrable`, `Eventual` (or `Effect`) and `Async` type-classes
- is meant as middleware, for interoperability purposes (e.g. conversions)
- this is meant for library authors, not users
- it should be as stable and as light as possible (i.e. no dependencies) and once we release `1.0.0` I'd like it to remain set in stone as to not create problems for the downstream

**Assignees**

No one assigned

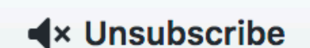
**Labels**

None yet

**Milestone**

No milestone

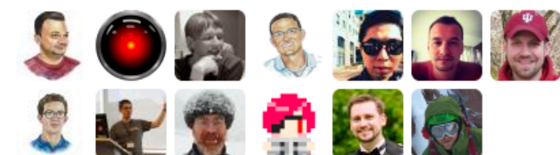
**Notifications**



Unsubscribe

You're receiving notifications because you modified the open/close state.

**13 participants**



# Submitting cats-effect for typelevel incubator membership #72

New issue

Closed djspiewak opened this issue on Apr 20, 2017 · 10 comments

djspiewak commented on Apr 20, 2017 • edited

Member

This is the accompanying typelevel/general issue for typelevel/cats#1617, and a fuller description of the project is there. I'd like to submit cats-effect as a Typelevel project. As this is the accompanying issue, I think it would be fair to treat an ultimate 👎 by the cats contributors as a similar vote on this issue.

1. The only developers are myself, **@mpilquist**, **@rossabaker** and **@tpolecat**, and this was our ultimate goal from the inception of the project
2. There is no mention of the typelevel CoC anywhere on the project at present, but that is easily fixed. Obviously no objections
3. Well, it promotes pure FP with cats in the face of harrowing side-effects, and it does use types. Some of those types are `Unit` though, so… 2 out of 3 I guess?
4. Readme is present. Scaladoc is very present. Tut docs are not present at all (ironically). Coming soon™

At present, the groupId is set to `org.typelevel`, simply because I don't think it makes sense to use my personal groupId if the package is `cats.effect`. I have secretly pushed artifacts to my bintray (mostly so Mike could get a build running), but ultimately I would rather this project sit within the fully-realized Sonatype release process on the main groupId. It is my intent to automate this in Travis, along with the artifact signing, when/if the project is approved.

👍 8

**Assignees**

No one assigned

**Labels**

Incubator approved

Typelevel membership

**Milestone**

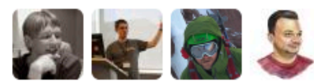
No milestone

**Notifications**

Unsubscribe

You're receiving notifications because you were mentioned.

**4 participants**



# EFFECTS

# EFFECTS

```
semaphore.acquire(10) +
semaphore.acquire(10)
```

```
val value = semaphore.acquire(10)
value + value
```

# EFFECTS

```
atomic.incrementAndGet(10) +
atomic.incrementAndGet(10)
```

```
val value = atomic.incrementAndGet(10)
value + value
```

# EFFECTS

```scala
val value: IO[Int] =
  atomic.incrementAndGet(10)

for {
  r1 <- value
  r2 <- value
} yield r1 + r2
```

# EFFECTS

```
def readLine(in: BufferedReader): IO[String] =
  IO(in.readLine())


def readLine(file: File): IO[String] = {
  val in = IO(new BufferedReader(new FileReader(file)))

  in.bracket(readLine)(in => IO(in.close()))
}
```

# EFFECTS

```scala
def forked[A](thunk: => A)
  (implicit ec: ExecutionContext): IO[A] =

  IO.async { cb =>
    ec.execute(() => cb(
      try Right(thunk)
      catch { case NonFatal(e) => Left(e) }
    ))
  }
```

# EFFECTS

```
def forked[A](thunk: => A)
  (implicit timer: Timer[IO]): IO[A] =

  timer.shift.flatMap(_ => IO(thunk))
```

# WHAT IS IO?

A pure abstraction representing the intention to perform a side effect

# WHAT IS IO?

```
type IO[+A] = () ⇒ Future[A]
```
*

* Not actually true ;-)

# ORIGINAL PHILOSOPHY

- Handling of **Effect Capture**
  - Atomic evaluation
  - No concurrency, no race conditions, no cancelation
- Type classes meant for abstracting over effects
  - Avoids a Scalaz 7 Task situation

# CANCELATION INCEPTION

- **Monix Task** has been cancelable since 2016

# CANCELATION INCEPTION

- **Monix Task** has been cancelable since 2016

- John De Goes announces Scalaz 8's new IO circa Aug 2017

# COMPETITION MODE: ON

Warning

Are you sure you want to cancel the IO?

No, cancel

Yes, continue

CREDITS: [@impurepics](@impurepics)

# BACK TO THE FUTURE[A]

# CANCELATION

```scala
def delay[A](delay: FiniteDuration)(f: ⇒ Future[A])
  (implicit sc: Scheduler): Future[A] = {

  val p = Promise[A]()
  sc.scheduleOnce(delay)(() ⇒ p.completeWith(f))
  p.future
}

def timeout[A](f: Future[A], after: FiniteDuration)
  (implicit sc: Scheduler): Future[A] = {

  val err = delay(after)(Future.failed(new TimeoutException))
  Future.firstCompletedOf(List(f, err))
}
```

# CANCELATION

# CANCELATION

```scala
def timeout[A](f: Future[A], after: FiniteDuration)
  (implicit sc: Scheduler): Future[A] = {

  val p = Promise[A]()
  val token = SingleAssignCancelable()

  token := sc.scheduleOnce(after) {
    p.tryFailure(new TimeoutException)
  }

  p.tryCompleteWith(f)
  p.future.onComplete(_ ⇒ token.cancel())
  p.future
}
```

# THE NAIVE WAY

```scala
def sleep(delay: FiniteDuration, sc: ScheduledExecutorService) =

  IO.async[Unit] { cb ⇒
    val r = new Runnable { def run() = cb(Right(())) }
    sc.schedule(r, delay.length, delay.unit)
  }
```

# THE REALISTIC WAY

```scala
def sleep(
  after: FiniteDuration,
  sc: ScheduledExecutorService): IO[(IO[Unit], IO[Unit])] = IO {

  val complete = Promise[Unit]()
  val r = new Runnable { def run() = complete.success(()) }
  val token = sc.schedule(r, after.length, after.unit)

  val io = IO.async { cb ⇒
    complete.future.onComplete(r ⇒ cb(r.toEither))
  }

  val cancel = IO { token.cancel(false); () }
  (io, cancel)
}
```

# THE REALISTIC WAY

```scala
case class Fiber[A](join: IO[A], cancel: IO[Unit])

def sleep(
  after: FiniteDuration,
  sc: ScheduledExecutorService): IO[Fiber[Unit]] =

  IO {
    // ...
    Fiber(io, cancel)
  }
```

# THE IDEAL

```scala
def sleep(
  delay: FiniteDuration,
  sc: ScheduledExecutorService): IO[Unit] = {

  IO.cancelable { cb ⇒
    // Scheduling of execution
    val r = new Runnable { def run() = cb(Right(())) }
    val token = sc.schedule(r, delay.length, delay.unit)
    // Cancellation
    IO(token.cancel(false))
  }
}
```

# THE IDEAL

```scala
def timeout[A](io: IO[A], after: FiniteDuration)
  (implicit timer: Timer[IO]): IO[A] = {

  val fallback =
    timer.sleep(after).flatMap { _ ⇒
      IO.raiseError[A](new TimeoutException(s"$after"))
    }

  IO.race(io, fallback).map(_.fold(a ⇒ a, b ⇒ b))
}
```

# THE REALIZATION

# THE REALIZATION

```scala
val task: IO[Unit] = sleep(10.seconds, scheduler)

val forked: IO[Fiber[Unit]] = task.start

// or in other words

val forked: IO[(IO[Unit], IO[Unit])] = task.start
```

# THE REALIZATION

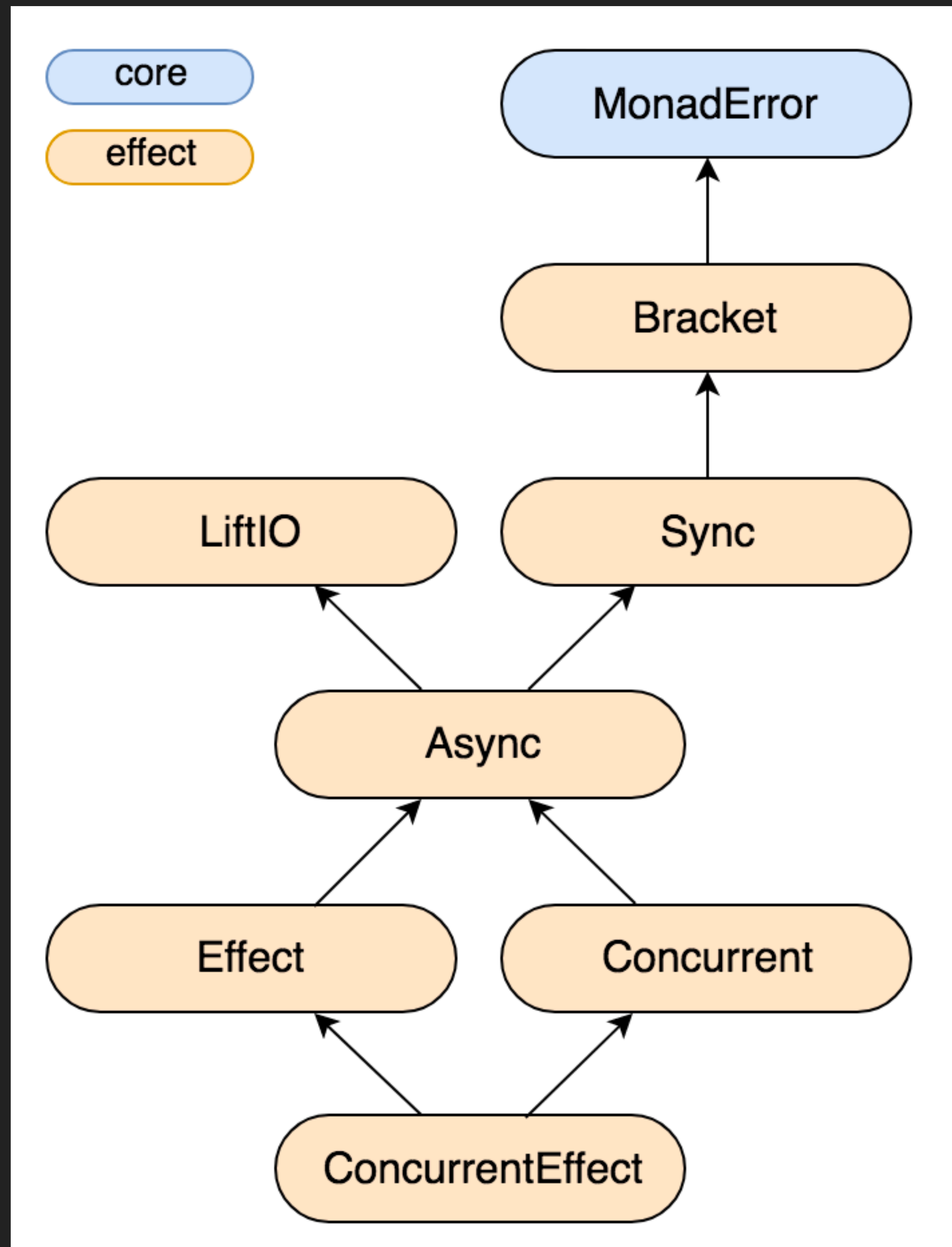
```
IO[A] ⇒ IO[(IO[A], IO[Unit])]
```

> Cancelability is nothing more than the ability to carry the cancelation token around, to use it in race conditions

Myself 😜

# THE API

# TYPECLASSES



# TYPECLASSES: BRACKET

```
trait Bracket[F[_], E] extends MonadError[F, E] {

  def bracketCase[A, B](acquire: F[A])
    (use: A ⇒ F[B])
    (release: (A, ExitCase[E]) ⇒ F[Unit]): F[B]

}
```

# TYPECLASSES: BRACKET

```scala
def readFile(file: File): IO[String] =
  IO(scala.io.Source.fromFile("file.txt")).bracket { in ⇒
    // Usage part
    IO(in.mkString)
  } { in ⇒
    // Release
    IO(in.close())
  }
```

# TYPECLASSES: CONCURRENT

```
trait Concurrent[F[_]] extends Async[F] {

  def cancelable[A](k: (Either[Throwable, A] ⇒ Unit) ⇒ IO[Unit]): F[A]

  def uncancelable[A](fa: F[A]): F[A]

  def onCancelRaiseError[A](fa: F[A], e: Throwable): F[A]

  def start[A](fa: F[A]): F[Fiber[F, A]]

  def racePair[A,B](fa: F[A], fb: F[B]):
    F[Either[(A, Fiber[F, B]), (Fiber[F, A], B)]]
}
```

# TYPECLASSES: CONCURRENT

```
def bracket[A, B](acquire: IO[A])(use: A ⇒ IO[B])
  (release: (A, ExitCase[Throwable]) ⇒ IO[Unit]): IO[B] = {

  acquire.flatMap { a ⇒
    use(a).onCancelRaiseError(new CancellationException).attempt.flatMap {
      case Right(a) ⇒
        // Success
        release(a, Completed).uncancelable *> IO.pure(a)
      case Left(_:CancellationException) ⇒
        // Cancelation
        release(a, Canceled(None)).uncancelable *> IO.never
      case Left(e) ⇒
        // Error
        release(a, Error(e)).uncancelable *> IO.raiseError(e)
    }
  }
}
```

# USE CASES

# USE-CASE: TIMER

```scala
trait Timer[F[_]] {
  def clockRealTime(unit: TimeUnit): F[Long]
  def clockMonotonic(unit: TimeUnit): F[Long]
  def sleep(duration: FiniteDuration): F[Unit]
  def shift: F[Unit]
}
```

# USE-CASE: TIMEOUTS

```
def never: IO[Nothing] = IO.async(_ ⇒ ())

never.timeout(10.seconds)
```

# USE-CASE: INTERVALS

```scala
package monix.tail

// ...
object Iterant {

  def intervalAtFixedRate[F[_]](period: FiniteDuration)
    (implicit F: Async[F], timer: Timer[F]): Iterant[F, Long] = ???
}

Iterant[IO].intervalAtFixedRate(10.seconds)
  .mapEval(_ ⇒ task)
```

# USE-CASE: CANCELABLE LOOPS

```scala
def fib(n: Int, a: Long, b: Long): IO[Long] =
  IO.suspend {
    if (n > 0) {
      val next = fib(n - 1, b, a + b)
      // Handles cancellation
      if (n % 128 == 0) IO.cancelBoundary *> next
      else next
    } else {
      IO.pure(a)
    }
  }
```

# USE-CASE: LOCKS

```scala
import cats.effect.concurrent.MVar

final class MLock(mvar: MVar[IO, Unit]) {
  def acquire: IO[Unit] =
    mvar.take

  def release: IO[Unit] =
    mvar.put(())

  def withPermit[A](fa: IO[A]): IO[A] =
    acquire.bracket(_ => fa)(_ => release)
}

object MLock {
  def apply(): IO[MLock] =
    MVar[IO].empty[Unit].map(ref => new MLock(ref))
}
```

# USE-CASE: LOCKS

```
lock.withPermit(IO(somethingExpensive))
  .timeout(10.seconds)
```

# USE-CASE: SEMAPHORE

```
import cats.effect.concurrent.Semaphore

for {
  semaphore ← Semaphore[IO](1)
  // ....
  task1 = semaphore.withLock(IO(somethingExpensive1))
  task2 = semaphore.withLock(IO(somethingExpensive2))
  // ...
  r ← IO.race(task1, task2) {
  }
} yield r
```

# USE-CASE: APP INTERRUPT

```
object Main extends IOApp {
  import ExitCode.Success

  def run(args: List[String]): IO[ExitCode] =
    IO.unit.bracket { _ ⇒
      for {
        _ ← IO(println("Started!"))
        _ ← IO.never
      } yield Success
    } { _ ⇒
      IO(println("Canceled!"))
    }
}
```

# DESIGN CHOICES

# DESIGN CHOICES

- Keeps the simplicity ideals of the project alive
  - **IO** is not and will not be as sophisticated as Monix's **Task**
  - **IO** is explicit by design
    - IO.shift
    - IO.cancelBoundary

# DESIGN CHOICES

- Keeps the simplicity ideals of the project alive
  - **IO** is not and will not be as sophisticated as Monix's **Task**
  - **IO** is explicit by design
    - IO.shift
    - IO.cancelBoundary

# DESIGN CHOICES

- Separation of concerns
  - Sync vs Async, Async vs Concurrent
- No auto-cancelation
  - Simplifies everything
  - Auto-cancelation infects the entire type-class hierarchy (e.g. a Monad restriction is no longer just a Monad restriction)

# DESIGN CHOICES

- Separation of concerns
  - Sync vs Async, Async vs Concurrent
- No auto-cancelation
  - Simplifies everything
  - Auto-cancelation infects the entire type-class hierarchy (e.g. a Monad restriction is no longer just a Monad restriction)

# CATS-EFFECT

- 1.0.0-RC2 (soon)
  - Ref
  - Deferred
  - Semaphore
  - MVar
  - IOApp

# QUESTIONS?

typelevel.org/cats-effect

@typelevel/cats-effect

@alexelcu