Gears: Asynchronous Programming in Direct Style Scala

Nguyen Pham, LAMP, EPFL

2024-03-22

About me

- My name: **Nguyen Pham** (In Vietnamese: Phạm Cao Nguyên)
  - Pronounce me! Wi (as in *win*) - en (as in *enter*)
- Second year PhD student in LAMP, EPFL
- Previous work:
  - Delimited Continuation, Scala Native
  - In industry: (Async) Rust, Go, Node.js, a bit of Haskell
- Currently: focused on Gears!

What is Gears?

Previously, in Scalar 2023…

Direct-Style Futures

With suspend(*), we can implement lightweight and universal await construct that can be called anywhere.

This can express simple, direct-style futures.

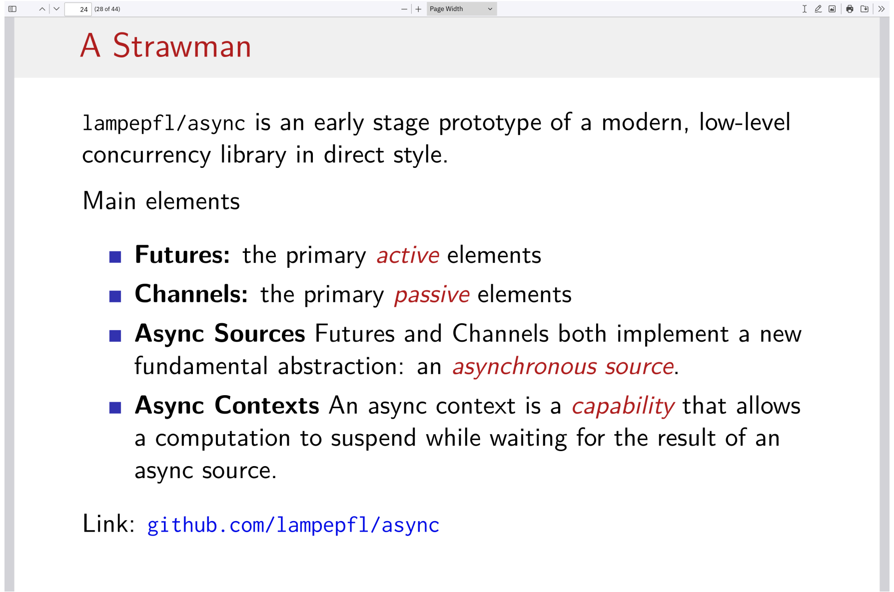
```
val sum = Future:
  val f1 = Future(c1.read)
  val f2 = Future(c2.read)
  f1.value + f2.value
```

Structured concurrency: Local futures f1 and f2 complete before sum completes. This might mean that one of them is cancelled if the other returns with a failure.

(*) Loom-like fibers would work as well.

(Martin Odersky - Direct Style Scala)

Previously, in Scalar 2023…



(Martin Odersky - Direct Style Scala)

Async is now Gears

- Before: *async - a strawman for asynchronous programming in Direct Style*
- Now: *Gears - an experimental library for asynchronous programming in Direct Style Scala*
- Releases: v0.1, v0.2-RC1!
- Supports Scala JVM (with Loom) and Scala Native (0.5 and above)
- API documentation: https://lampepfl.github.io/gears/api
  "Concurrency with Gears": https://lampepfl.github.io/gears

Async is now Gears

- Before: *async - a strawman for asynchronous programming in Direct Style*
- Now: *Gears - an experimental library for asynchronous programming in Direct Style Scala*
- Releases: v0.1, v0.2-RC1!
- Supports Scala JVM (with Loom) and Scala Native (0.5 and above)
- API documentation: https://lampepfl.github.io/gears/api
  "Concurrency with Gears": https://lampepfl.github.io/gears

Async is now Gears

- Before: *async - a strawman for asynchronous programming in Direct Style*
- Now: *Gears - an experimental library for asynchronous programming in Direct Style Scala*
- Releases: v0.1, v0.2-RC1!
- Supports Scala JVM (with Loom) and Scala Native (0.5 and above)
- API documentation: https://lampepfl.github.io/gears/api
  "Concurrency with Gears": https://lampepfl.github.io/gears

Async is now Gears

- Before: *async - a strawman for asynchronous programming in Direct Style*
- Now: *Gears - an experimental library for asynchronous programming in Direct Style Scala*
- Releases: v0.1, v0.2-RC1!
- Supports Scala JVM (with Loom) and Scala Native (0.5 and above)
- API documentation: https://lampepfl.github.io/gears/api
  "Concurrency with Gears": https://lampepfl.github.io/gears

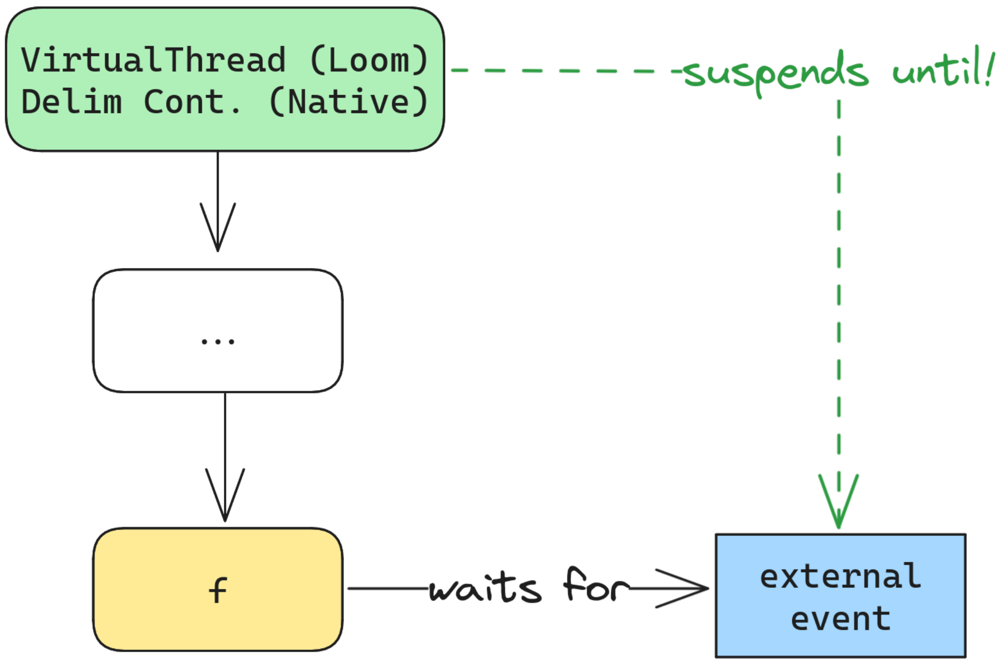
Async is now Gears

- Before: *async - a strawman for asynchronous programming in Direct Style*
- Now: *Gears - an experimental library for asynchronous programming in Direct Style Scala*
- Releases: v0.1, v0.2-RC1!
- Supports Scala JVM (with Loom) and Scala Native (0.5 and above)
- API documentation: https://lampepfl.github.io/gears/api
  "Concurrency with Gears": https://lampepfl.github.io/gears

The Primary Concepts

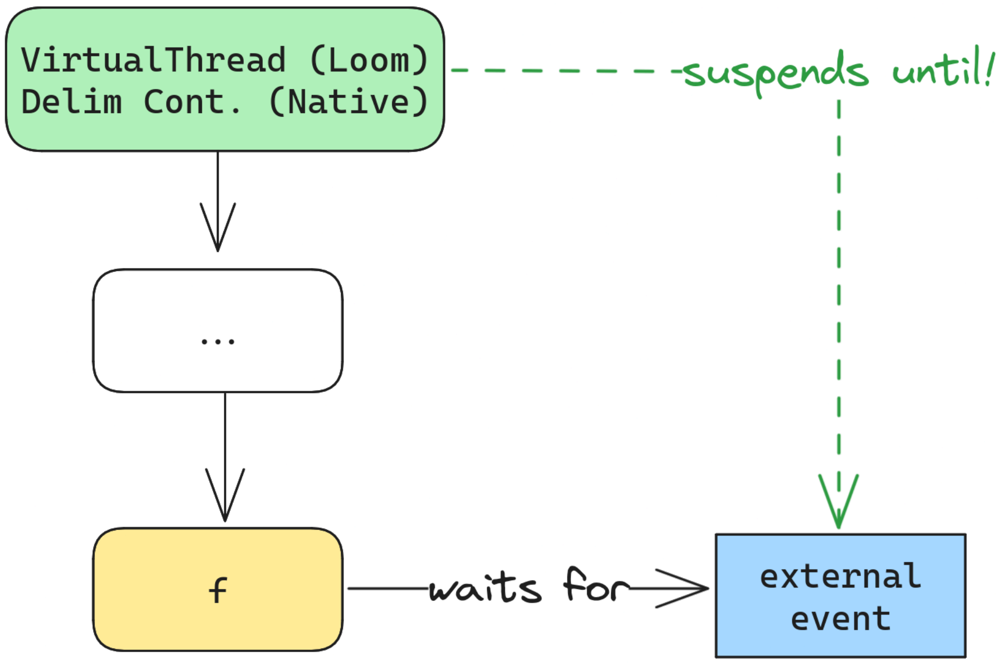
The ability to wait…

- Virtual threads gives you the ability to do blocking operations cheaply - by suspending themselves away from the physical thread.
  - Delimited continuations & a scheduler give you the same thing.
- Requires you to be part of it, from the root of the call stack…
- We can model it as a capability!



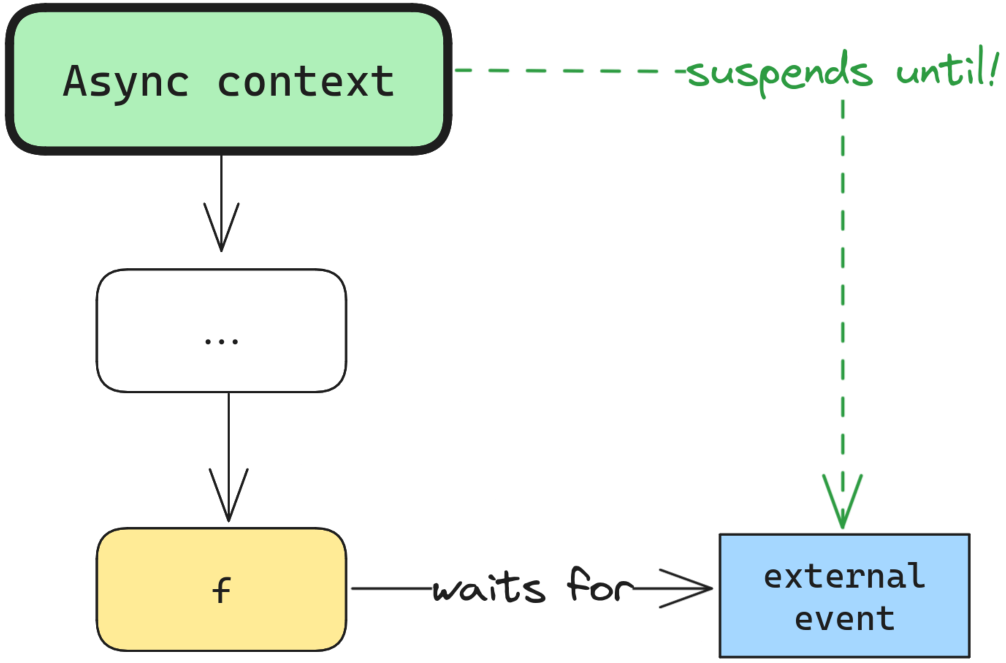
The ability to wait…

- Virtual threads gives you the ability to do blocking operations cheaply - by suspending themselves away from the physical thread.
  - Delimited continuations & a scheduler give you the same thing.
- Requires you to be part of it, from the root of the call stack…
- We can model it as a capability!



The ability to wait…

- Virtual threads gives you the ability to do blocking operations cheaply - by suspending themselves away from the physical thread.
  - Delimited continuations & a scheduler give you the same thing.
- Requires you to be part of it, from the root of the call stack…
- We can model it as a capability!



An Overview of Gears

- **Async Contexts**: A capability that allows cheap suspension of computations to wait for a future event. Gives `.await`.
- **Futures**: Simple, straightforward creation of concurrent computations.
- **Structured Concurrency**: Organize concurrent computations into an easily manageable tree-like structure.
- **Sources and Channels**: a toolbox for dealing with the complexity of external and inter-dependent unstructured concurrency.

An Overview of Gears

- **Async Contexts**: A capability that allows cheap suspension of computations to wait for a future event. Gives `.await`.
- **Futures**: Simple, straightforward creation of concurrent computations.
- **Structured Concurrency**: Organize concurrent computations into an easily manageable tree-like structure.
- **Sources and Channels**: a toolbox for dealing with the complexity of external and inter-dependent unstructured concurrency.

A simple example

```scala
def sumFiles(f1: File, f2: File)(using Async): Int =
  Async.group:
    val v1 = Future(f1.read())
    val v2 = f2.read()
    v1.await.parse() + v2.parse()
```

- v1 and v2 are concurrent
- .await suspends Async context until Future is ready, returns String
- It's all using Async all the way down*!

```scala
trait File:
  def read()(using Async): String
```

A simple example

```
def sumFiles(f1: File, f2: File)(using Async): Int =
  Async.group:
    val v1 = Future(f1.read())
    val v2 = f2.read()
    v1.await.parse() + v2.parse()
```

- v1 and v2 are concurrent
- .await suspends Async context until Future is ready, returns String
- It's all using Async all the way down*!

```
trait File:
  def read()(using Async): String
```

A simple example

```
def sumFiles(f1: File, f2: File)(using Async): Int =
  Async.group:
    val v1 = Future(f1.read())
    val v2 = f2.read()
    v1.await.parse() + v2.parse()
```

- `v1` and `v2` are concurrent
- `.await` suspends `Async` context until `Future` is ready, returns `String`
- It's all using `Async` all the way down*!

```
trait File:
  def read()(using Async): String
```

A simple example

```
def sumFiles(f1: File, f2: File)(using Async): Int =
  Async.group:
    val v1 = Future(f1.read())
    val v2 = f2.read()
    v1.await.parse() + v2.parse()
```

- `v1` and `v2` are concurrent
- `.await` suspends `Async` context until `Future` is ready, returns `String`
- It's all using `Async` all the way down*!

```
trait File:
  def read()(using Async): String
```

Async Contexts

We can look at the "`Async` context" both as a capability and a context:

- **As a capability**: allows the function/computation to be suspended!
  Signals possibility of cancellation, side-effect tracking, safety
- **As a context**: runtime information on *how* to perform suspension, attached scheduler (a.k.a `ExecutionContext`), a *structured scope*
- **But more importantly**: just a regular implicit parameter!

```
def fn()(using Async): String = ??? // returns a real string!
```

  No `Promise`, no `Future`, no monads!

Async Contexts

We can look at the "`Async` context" both as a capability and a context:

- **As a capability**: allows the function/computation to be suspended!
  Signals possibility of cancellation, side-effect tracking, safety
- **As a context**: runtime information on *how* to perform suspension, attached scheduler (a.k.a `ExecutionContext`), a *structured scope*
- **But more importantly**: just a regular implicit parameter!

```
def fn()(using Async): String = ??? // returns a real string!
```

No `Promise`, no `Future`, no monads!

Async Contexts

We can look at the "`Async` context" both as a capability and a context:

- **As a capability**: allows the function/computation to be suspended! Signals possibility of cancellation, side-effect tracking, safety
- **As a context**: runtime information on *how* to perform suspension, attached scheduler (a.k.a `ExecutionContext`), a *structured scope*
- **But more importantly**: just a regular implicit parameter!

```scala
def fn()(using Async): String = ??? // returns a real string!
```

No `Promise`, no `Future`, no monads!

Async Contexts

We can look at the "`Async` context" both as a capability and a context:

- **As a capability**: allows the function/computation to be suspended!
  Signals possibility of cancellation, side-effect tracking, safety
- **As a context**: runtime information on *how* to perform suspension, attached scheduler (a.k.a `ExecutionContext`), a *structured scope*
- **But more importantly**: just a regular implicit parameter!

  ```scala
  def fn()(using Async): String = ??? // returns a real string!
  ```

  No `Promise`, no `Future`, no monads!

Almost a blocking API

Sequentially calling “async functions” is as simple as

```
def f()(using Async): Int = ???
def g()(using Async): Int = ???

def h()(using Async) =
  f() + g()
```

`h` blocks until `f` returns, then blocks until `g` returns, possibly suspending within `f` or `g`.

Sequential actions stay the same

```scala
trait Item:
  def transform()(using Async): this.type
  def isValid(using Async): Boolean

def transformAll(items: Seq[Item])(using Async) =
  items
    .filter(_.isValid)  // Seq.filter
    .map(_.transform()) // Seq.map
```

Capturing `Async` is completely fine, if they don't persist.

Futures: Spawning Concurrent Computations

To spawn concurrent computations, you need `Async.Spawn`:

```
def spawn()(using Async): Int =
  Async.group: (spawn: Async.Spawn) ?=>
    val v1 = Future(using spawn)(async ?=> f()(using async))
    val v2 = Future(using spawn)(async ?=> g()(using async))
    val v3 = Future(using spawn): async ?=>
      sleep(1000.years)(using async)
    v1.await(using spawn) + v2.await(using spawn)
```

Once `Async.group` returns, v3 is cancelled. No futures running after `spawn`.

Futures: Spawning Concurrent Computations

To spawn concurrent computations, you need `Async.Spawn`:

```
def spawn()(using Async): Int =
  Async.group:
    val v1 = Future(f())
    val v2 = Future(g())
    val v3 = Future:
      sleep(1000.years)
    v1.await + v2.await
```

Once `Async.group` returns, `v3` is cancelled. No futures running after `spawn`.

Async Scopes

Futures are properly scoped to their context:

```
def run()(using async: Async) =
  Async.group: // creates child Async context
    val a = Future(...)
    val b = Future(...)
  // a & b cleaned up
```

Async Scopes

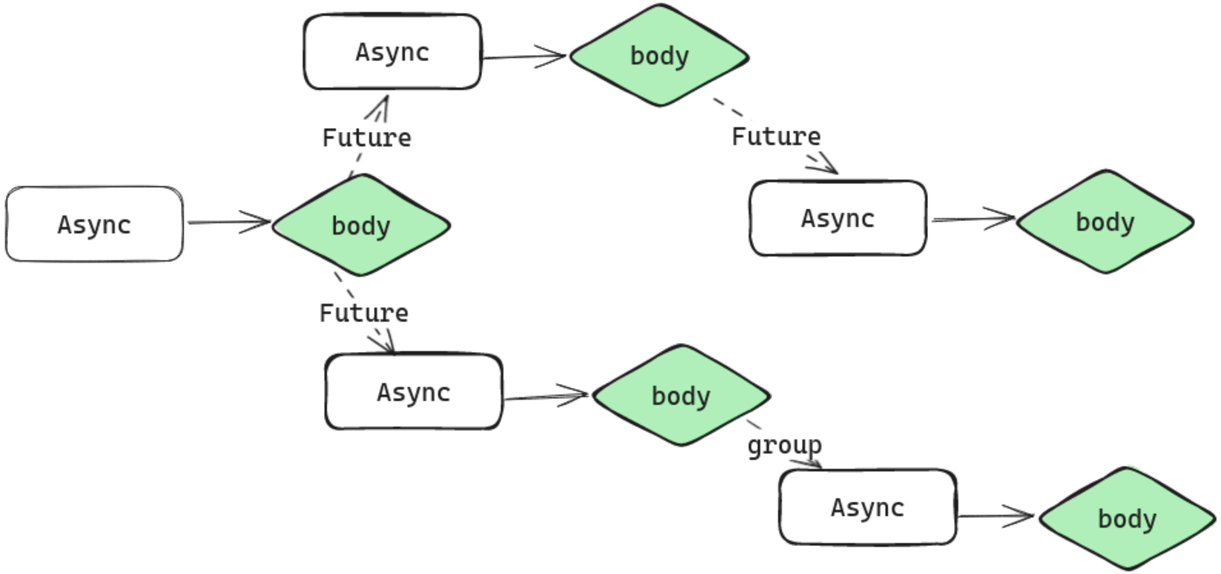
Futures are properly scoped to their context:

```
def run()(using async: Async) =
  Async.group: // creates child Async context
    val a = Future(...)
    val b = Future(...)
  // a & b cleaned up

  val vf = f()(using async)
  // all futures spawned by f() cleaned up
```

Structured Concurrency

```scala
def run()(using Async) =
  val f = Future:
    val f1 = Future(...)
    ...
  val g = Future:
    val g1 = Future(...)
    ...
  f.await + g.await
```



Async scopes with concurrent computations form a tree

Future Composition: In and Out

- `.await` throws if the `Future` does, `.awaitResult` returns `Try[T]`.
- `.or` and `.zip` simplifies racing and combining two futures.

```
val (v1, v2) = f1.zip(f2.or(f3)).await
```

- `Seq[Future[_]]` methods:
  - `.awaitAll`: essentially `.map(_.await)`, but throws early!
  - `.awaitFirst`: Get the first `Future` returning with success.
  - … and their `withCancel` counterparts: quickly cancel unneeded futures.

```
items.map(v => Future(v.transformAsync()))
     .awaitAll                  // in parallel, almost equivalent to...
  // .map(_.await)
```

Future Composition: In and Out

- `.await` throws if the `Future` does, `.awaitResult` returns `Try[T]`.
- `.or` and `.zip` simplifies racing and combining two futures.

```
val (v1, v2) = f1.zip(f2.or(f3)).await
```

- `Seq[Future[_]]` methods:
  - `.awaitAll`: essentially `.map(_.await)`, but throws early!
  - `.awaitFirst`: Get the first `Future` returning with success.
  - … and their `withCancel` counterparts: quickly cancel unneeded futures.

```
items.map(v => Future(v.transformAsync()))
     .awaitAll                      // in parallel, almost equivalent to...
  // .map(_.await)
```

Future Composition: In and Out

- `.await` throws if the `Future` does, `.awaitResult` returns `Try[T]`.
- `.or` and `.zip` simplifies racing and combining two futures.

```
val (v1, v2) = f1.zip(f2.or(f3)).await
```

- `Seq[Future[_]]` methods:
  - `.awaitAll`: essentially `.map(_.await)`, but throws early!
  - `.awaitFirst`: Get the first `Future` returning with success.
  - … and their `withCancel` counterparts: quickly cancel unneeded futures.

```
items.map(v => Future(v.transformAsync()))
     .awaitAll                    // in parallel, almost equivalent to...
  // .map(_.await)
```

Future Composition: Select

Go’s `select`, but for futures.

- No `Future` wrapping, no clunky syntax*,
- `.handle` takes a normal lambda and returns a real value.
- For side-effects: guarantees **exactly one** branch evaluated!

```
val f1 = Future(1)
val f2 = Future("one")
val v: Either[Int, String] = Async.select(
   f1.handle: i =>
     println(s"Int $i")
     Left(i),
   f2.handle: s =>
     println(s"String $s")
     Right(s),
)
```

Future Communication: Channels

- Simple `.read()(using Async)` and `.send(x: T)(using Async)` APIs
- Can combine with the power of `Async.select` if needed
- Comes in 3 variants: `SyncChannel`, `BufferedChannel`, `UnboundedChannel`

```
val in = SyncChannel[Work]()
val out = BufferedChannel[Result](size: 10)
val workers = (1 to 10).map: _ =>
  Future:
    in.read() match
      case Left(Closed) => ()
      case Right(work) => out.send(process(work))

def loop(i: Int): Unit =
  if i == 1000 then in.close()
  else Async.select(
    in.sendSource(Work(i)).handle(_ => loop(i+1)),
    out.readSource.handle: result =>
      println(s"Work result: $result")
      loop(i),
  )
```

Future Communication: Channels

- Simple `.read()(using Async)` and `.send(x: T)(using Async)` APIs
- Can combine with the power of `Async.select` if needed
- Comes in 3 variants: `SyncChannel`, `BufferedChannel`, `UnboundedChannel`

```scala
val in = SyncChannel[Work]()
val out = BufferedChannel[Result](size: 10)
val workers = (1 to 10).map: _ =>
  Future:
    in.read() match
      case Left(Closed) => ()
      case Right(work) => out.send(process(work))

def loop(i: Int): Unit =
  if i == 1000 then in.close()
  else Async.select(
    in.sendSource(Work(i)).handle(_ => loop(i+1)),
    out.readSource.handle: result =>
      println(s"Work result: $result")
      loop(i),
  )
```

Future Communication: Channels

- Simple `.read()(using Async)` and `.send(x: T)(using Async)` APIs
- Can combine with the power of `Async.select` if needed
- Comes in 3 variants: `SyncChannel`, `BufferedChannel`, `UnboundedChannel`

```
val in = SyncChannel[Work]()
val out = BufferedChannel[Result](size: 10)
val workers = (1 to 10).map: _ =>
  Future:
    in.read() match
      case Left(Closed) => ()
      case Right(work) => out.send(process(work))

def loop(i: Int): Unit =
  if i == 1000 then in.close()
  else Async.select(
    in.sendSource(Work(i)).handle(_ => loop(i+1)),
    out.readSource.handle: result =>
      println(s"Work result: $result")
      loop(i),
  )
```

Future Communication: Channels

- Simple `.read()(using Async)` and `.send(x: T)(using Async)` APIs
- Can combine with the power of `Async.select` if needed
- Comes in 3 variants: `SyncChannel`, `BufferedChannel`, `UnboundedChannel`

```
val in = SyncChannel[Work]()
val out = BufferedChannel[Result](size: 10)
val workers = (1 to 10).map: _ =>
  Future:
    in.read() match
      case Left(Closed) => ()
      case Right(work) => out.send(process(work))

def loop(i: Int): Unit =
  if i == 1000 then in.close()
  else Async.select(
    in.sendSource(Work(i)).handle(_ => loop(i+1)),
    out.readSource.handle: result =>
      println(s"Work result: $result")
      loop(i),
  )
```

Sources: Working with External, Unstructred Events

`Async.Source` is a common abstraction for all awaitable source of values.

- `Promise` and `Future.withResolver` creates bridges for callbacks
- Source allows a stream of values to arrive
- Existing tools work: `.await` and `Async.select`
- Conversion from `scala.concurrent.Future`: `.asGears`.

```
def withCallback(arg: Int)(callback: Try[String] => Unit)
    : Unit = ???
def withGears(arg: Int): Future[String] =
  Future.withResolver: resolver =>
    withCallback(arg)(resolver.complete))
```

Sources: Working with External, Unstructred Events

`Async.Source` is a common abstraction for all awaitable source of values.

- `Promise` and `Future.withResolver` creates bridges for callbacks
- `Source` allows a stream of values to arrive
- Existing tools work: `.await` and `Async.select`
- Conversion from `scala.concurrent.Future`: `.asGears`.

```
def withCallback(arg: Int)(callback: Try[String] => Unit)
    : Unit = ???
def withGears(arg: Int): Future[String] =
  Future.withResolver: resolver =>
    withCallback(arg)(resolver.complete))
val timer = new Timer.Tick(every: 500.millis)
val fut = Future(timer.run())
while true do
  timer.await
  println("Hi!") // prints every 500 millis
```

Sources: Working with External, Unstructred Events

`Async.Source` is a common abstraction for all awaitable source of values.

- `Promise` and `Future.withResolver` creates bridges for callbacks
- `Source` allows a stream of values to arrive
- Existing tools work: `.await` and `Async.select`
- Conversion from `scala.concurrent.Future`: `.asGears`.

```
def withCallback(arg: Int)(callback: Try[String] => Unit)
    : Unit = ???
def withGears(arg: Int): Future[String] =
  Future.withResolver: resolver =>
    withCallback(arg)(resolver.complete))
val timer = new Timer.Tick(every: 500.millis)
val fut = Future(timer.run())
while true do
  timer.await
  println("Hi!") // prints every 500 millis
```

Sources: Working with External, Unstructred Events

`Async.Source` is a common abstraction for all awaitable source of values.

- `Promise` and `Future.withResolver` creates bridges for callbacks
- `Source` allows a stream of values to arrive
- Existing tools work: `.await` and `Async.select`
- Conversion from `scala.concurrent.Future`: `.asGears`.

```
def withCallback(arg: Int)(callback: Try[String] => Unit)
    : Unit = ???
def withGears(arg: Int): Future[String] =
  Future.withResolver: resolver =>
    withCallback(arg)(resolver.complete))
val timer = new Timer.Tick(every: 500.millis)
val fut = Future(timer.run())
while true do
  timer.await
  println("Hi!") // prints every 500 millis
val stdFuture = new scala.concurrent.Future(...)
val gearsFuture = stdFuture.asGears
val value = gearsFuture.await
val stdFutureAgain = gearsFuture.asScala
```

Writing Gears code

Error Handling

Gears embraces `Try`, but direct style lets you write your own error handling easily.

- `Future` wraps exceptions in a `Try`, unwrapped by default.
- Cancellation are handled through catching `CancellationException`.
- Build your own error handling: `CanThrow`, `Result`, `boundary`/`break`: direct style makes it trivial.

Error Handling

Gears embraces `Try`, but direct style lets you write your own error handling easily.

- `Future` wraps exceptions in a `Try`, unwrapped by default.
- Cancellation are handled through catching `CancellationException`.
- Build your own error handling: `CanThrow`, `Result`, `boundary`/`break`: direct style makes it trivial.

```
val f = Future(...)
f.await // unwraps Try
f.awaitResult // returns Try[T]
```

Error Handling

Gears embraces `Try`, but direct style lets you write your own error handling easily.

- `Future` wraps exceptions in a `Try`, unwrapped by default.
- Cancellation are handled through catching `CancellationException`.
- Build your own error handling: `CanThrow`, `Result`, `boundary`/`break`: direct style makes it trivial.

```
val f = Future(...)
f.await // unwraps Try
f.awaitResult // returns Try[T]
```

```
Future:
  try
    sleep(10.minutes)
  catch
    case _: CancellationException =>
      println("Sleep cancelled")
```

Error Handling

Gears embraces `Try`, but direct style lets you write your own error handling easily.

- `Future` wraps exceptions in a `Try`, unwrapped by default.
- Cancellation are handled through catching `CancellationException`.
- Build your own error handling: `CanThrow`, `Result`, `boundary/break`: direct style makes it trivial.

```
val f = Future(...)
f.await // unwraps Try
f.awaitResult // returns Try[T]

def failible()(using Async): Result[Int]
val fut: Future[Result[Int]] = Future:
  Result:
    val f = failible().?
    f + 1
fut.await //: Result[Int]
```

Timeout and Retry

- `withTimeout` creates a scope that is cancelled after the timeout.
- `Retry` lets you run actions with retrying, delay, backoff, …
- All “blocking”: feel free to run them in `Future`. Actor pattern!

```
val body: String = withTimeout(10.millis):
  val f = requests.get("https://google.com")
  f.body
```

Timeout and Retry

- `withTimeout` creates a scope that is cancelled after the timeout.
- `Retry` lets you run actions with retrying, delay, backoff, …
- All "blocking": feel free to run them in Future. Actor pattern!

```
val body: String = withTimeout(10.millis):
  val f = requests.get("https://google.com")
  f.body


  Retry
    .untilSuccess
    .withMaximumFailures(5)
    .withDelay(
      Delay.exponentialBackoff(
        maximum = 1.minute,
        starting = 1.second,
        jitter = Jitter.full,
    )):
      val body = request.get("https://google.com")
      // ...
```

Timeout and Retry

- `withTimeout` creates a scope that is cancelled after the timeout.
- `Retry` lets you run actions with retrying, delay, backoff, …
- All "blocking": feel free to run them in `Future`. Actor pattern!

```
val body: String = withTimeout(10.millis):
  val f = requests.get("https://google.com")
  f.body

val worker = Future:
  Retry
    .untilSuccess
    .withMaximumFailures(5)
    .withDelay(
      Delay.exponentialBackoff(
        maximum = 1.minute,
        starting = 1.second,
        jitter = Jitter.full,
    )):
      val body = request.get("https://google.com")
      // ...
```

Entry into the async world

How do you get an `Async` context in the first place?

Entry into the async world

How do you get an `Async` context in the first place?

Components of `Async`:
- A suspension mechanism
- Capability to resume a computation
- Management of child scopes

Ingredients of `Async.blocking`:
- `SuspendSupport` a.k.a delimited continuation interface
- A Scheduler
- ~~`CompletionGroup`~~ created automatically

Entry into the async world

How do you get an `Async` context in the first place?

Components of `Async`:

- A suspension mechanism
- Capability to resume a computation
- Management of child scopes

Ingredients of `Async.blocking`:

- `SuspendSupport` a.k.a delimited continuation interface
- A Scheduler
- ~~`CompletionGroup`~~ created automatically

Entry into the async world

How do you get an `Async` context in the first place?

Components of `Async`:

- A suspension mechanism
- Capability to resume a computation
- Management of child scopes

Ingredients of `Async.blocking`:

- `SuspendSupport` a.k.a delimited continuation interface
- A Scheduler
- ~~`CompletionGroup`~~ created automatically

Entry into the async world

How do you get an `Async` context in the first place?

Components of `Async`:

- A suspension mechanism
- Capability to resume a computation
- Management of child scopes

Ingredients of `Async.blocking`:

- `SuspendSupport` a.k.a delimited continuation interface
- A Scheduler
- ~~`CompletionGroup`~~ created automatically

```
def blocking(body: Async ?=> T)(using AsyncSupport, AsyncSupport.Scheduler)
```

where default implementations of the interfaces are provided within Gears with `gears.async.default.given`. Custom implementations welcome!

Entry into the async world

How do you get an `Async` context in the first place?

Components of `Async`:

- A suspension mechanism
- Capability to resume a computation
- Management of child scopes

Ingredients of `Async.blocking`:

- `SuspendSupport` a.k.a delimited continuation interface
- A Scheduler
- ~~`CompletionGroup`~~ created automatically

```
def blocking(body: Async ?=> T)(using AsyncSupport, AsyncSupport.Scheduler)
```

where default implementations of the interfaces are provided within Gears with `gears.async.default.given`. Custom implementations welcome!

`Async.blocking` lets you "suspend" to wait for `Future`s. It does that… by blocking the thread.

Entry into the async world

How do you get an `Async` context in the first place?

Components of `Async`:

- A suspension mechanism
- Capability to resume a computation
- Management of child scopes

Ingredients of `Async.blocking`:

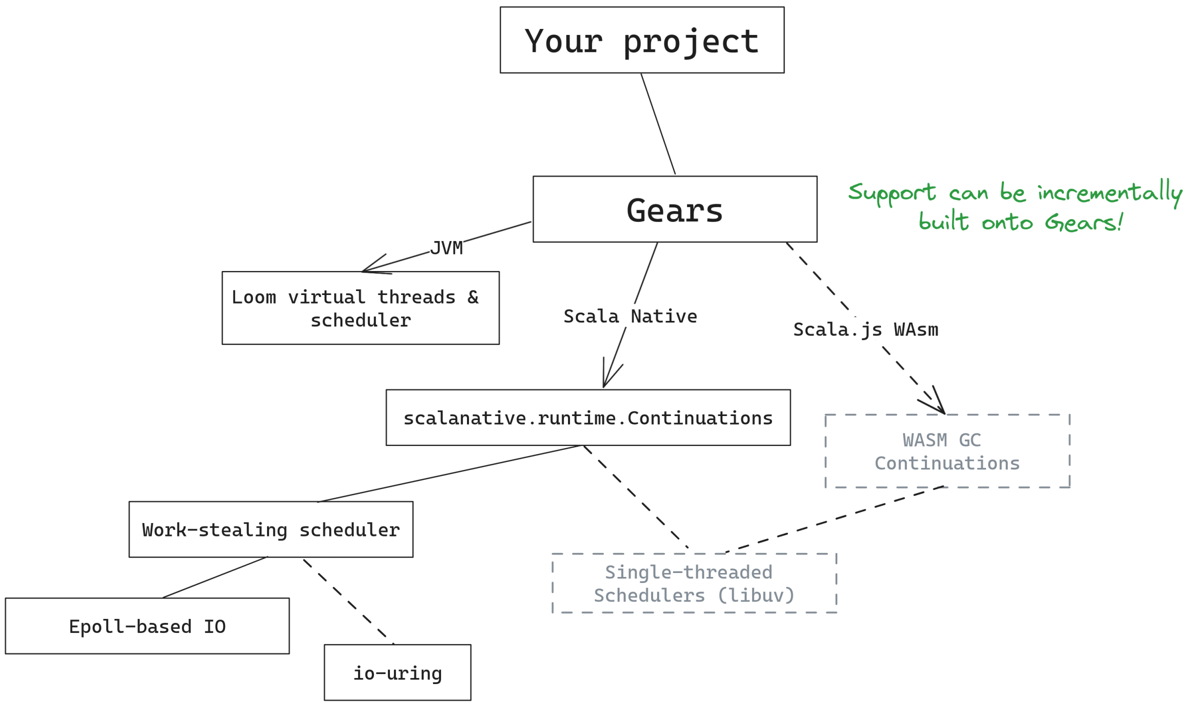
- `SuspendSupport` a.k.a delimited continuation interface
- A Scheduler
- ~~`CompletionGroup`~~ created automatically

```
def blocking(body: Async ?=> T)(using AsyncSupport, AsyncSupport.Scheduler)
```

where default implementations of the interfaces are provided within Gears with `gears.async.default.given`. Custom implementations welcome!

`Async.blocking` lets you "suspend" to wait for `Future`s. It does that… by blocking the thread.
Usage: part of `@main`, or during conversion from blocking code!

Target support: Now and beyond



What's next?

A new view of concurrency

- **Loom and Continuations** allow a direct-style `.await` API, making natural asynchronous code possible
- **Viewing `Async` as a capability** lets us use Scala's unique implicit parameter for a lean approach to managing concurrent code.
- Gears combines both and introduces **Structured Concurrency** as a guiding principle for writing concurrent programs.

A new view of concurrency

- **Loom and Continuations** allow a direct-style `.await` API, making natural asynchronous code possible
- **Viewing `Async` as a capability** lets us use Scala's unique implicit parameter for a lean approach to managing concurrent code.
- Gears combines both and introduces **Structured Concurrency** as a guiding principle for writing concurrent programs.

A new view of concurrency

- **Loom and Continuations** allow a direct-style `.await` API, making natural asynchronous code possible
- **Viewing `Async` as a capability** lets us use Scala's unique implicit parameter for a lean approach to managing concurrent code.
- Gears combines both and introduces **Structured Concurrency** as a guiding principle for writing concurrent programs.

Next steps for Gears

- Gears right now is just base framework!
- IO: the source of (most) suspends!
  - gears-io: a cross-platform interface for IO ops. Think fs2, but on gears.

```
trait Reader:
  def read(buf: Buffer)(using Async): Int
```

    Coming soon!

- A first "real-use" library: an HTTP client!
- To flesh out: customizing cancellation models, supervising futures

Next steps for Gears

- Gears right now is just base framework!
- IO: the source of (most) suspends!
  - `gears-io`: a cross-platform interface for IO ops. Think `fs2`, but on `gears`.

```
trait Reader:
  def read(buf: Buffer)(using Async): Int
```

Coming soon!

- A first "real-use" library: an HTTP client!
- To flesh out: customizing cancellation models, supervising futures

Next steps for Gears

- Gears right now is just base framework!
- IO: the source of (most) suspends!
  - `gears-io`: a cross-platform interface for IO ops. Think `fs2`, but on `gears`.

```
trait Reader:
  def read(buf: Buffer)(using Async): Int
```

Coming soon!

- A first "real-use" library: an HTTP client!
- To flesh out: customizing cancellation models, supervising futures

Next steps for Gears

- Gears right now is just base framework!
- IO: the source of (most) suspends!
  - `gears-io`: a cross-platform interface for IO ops. Think `fs2`, but on `gears`.

    ```
    trait Reader:
      def read(buf: Buffer)(using Async): Int
    ```

    Coming soon!
- A first "real-use" library: an HTTP client!
- To flesh out: customizing cancellation models, supervising futures

Thank you!

To learn more about Gears:

https://lampepfl.github.io/gears

Follow its development:

- GitHub: lampepfl/gears
- Me: `@natsukagami` (GitHub), `@nki@dtth.ch` (Mastodon)
- Lots of development documented on Gears Website!

Learn more about Direct Style Scala:

- Martin Odersky, "Direct Style Scala", Scalar 2023
- Adam Warski, Ox: Asynchronous Programming with Direct Style & Loom

Bonus Slides

Comparison to Ox

- Ox forgoes the concept of suspension.
  Loom Virtual threads means blocking == suspending.
  - Gears keeps this explicit. Allows explicit tracking of this capability, and allowing independent implementations from core Scala Native.
- Ox has user, daemon and unsupervised threads.
  Gears make a simplification: There are only `Futures` that:
  - Completes with `Failure` on exception
  - Don't cancel parent scope on failure
  - Are cancelled when scope ends
- Ox bakes in `Either` and `Try` support for error handling, Gears prefers `Try`.

What color is your function?

1. Every function has a color. **Yes, either you take `Async`, or you don't.**
2. The way you call a function depends on its color. **No!**
3. You can only call a red function from within a red function. **Yes***.
   - `Async.blocking` exists, but you have to be aware of its limits.
4. Red functions are more painful to call. **No…**
5. Some core library functions are red. **Not yet, but will be, and that's fine!**

Original Article:
https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/