Embedding Generic Monadic Transformer into Scala.

{Can we return concurrent programming into mainstream? }

Ruslan Shevchenko, Kyiv, Ukraine.
<ruslan@shevchenko.kiev.ua>
proofspace.id

https://github.com/rssh/dotty-cps-async

Asynchronous control flow - Scala industrial usage.

```
for{
  r1 <- cacheService.get(id) match
    case Some(v) => IO.success(v)
    case None => talkToServer(id).map{v =>
      cacheService.update(id,v)
      v
    }
  r2 <- talkToServer(r1.data)
  result <- if (r2.isOk) then {
          writeToFile(resultData) >>=
          IO.println("done") >>=
          true
       } else {
          IO.println("abort") >>=
          false
       }
} yield result
```

Monadic DSL on top of some effect system or. Future

Asynchronous control flow - Scala industrial usage.

```
for{
  r1 <- cacheService.get(id) match
    case Some(v) => IO.success(v)
    case None => talkToServer(id).map{v =>
      cacheService.update(id,v)
      v
    }
  R2 <- talkToServer(r1.data)
  result <- if (r2.isOk) then {
          writeToFile(r1.data) >>=
          IO.println("done") >>=
          true
        } else {
          IO.println("abort") >>=
          false
        }
} yield result
```

Monadic DSL on top of some effect system or. Future

```
async[IO] {
  val r1 = cacheService.getOrUpdate(id,
                        await(talkToServer(id)))
  val r2 = talkToServer(r1.data)
  if (r2.isOk) then
    writeToFile(r1.data)
    IO.println("done")
    true
  else
    IO.println("abort")
    false
}
```

"Mainstream" Control-Flow

over some effect system or Future

API (simplified)

```
def async[F[_]](f: T):F[T]
def await[F[_]](f: F[T]):T
```

Well-known async/await interface.

+ - generic monad

  - ability to use higher-order functions without runtime support for continuations.

  - automatic colouring (awaits can-be omitted if can be restored from the context)

API (simplified)

```
def async[F[_]](f: T):F[T]
def await[F[_]](f: F[T]):T
```

```
transparent inline
    def async[F[_]](using am: CpsMonad[F])[T](inline f: am.Context ?=> T):F[T]
```

// in real signature here is temporary class, to allow currying of type parameters

```
@compileTimeOnly("await should be inside async block")
    def await[G[_],T,F[_]](f: G[T])(using CpsMonadConversion[G,F], CpsMonadContext[F]):
```

API (simplified)

```
transparent inline
   def async[F[_]](using am: CpsMonad[F])[T](inline f: am.Context ?=> T):F[T]
```



```
@compileTimeOnly("await should be inside async block")
def await[G[_],T,F[_]](f: G[T])(using CpsMonadConversion[G,F], CpsMonadContext[F]): T
```

Macro, evaluated during typing

API (simplified)

```
transparent inline
    def async[F[_]](using am: CpsMonad[F])[T](inline f: am.Context ?=> T):F[T]
```

```
@compileTimeOnly("await should be inside async block")
def await[G[_],T,F[_]](f: G[T])(using CpsMonadCoversion[G,F], CpsMonadContext[F]): T
```

F[_]. — Our monad : Future[T], IO[T], … etc

G[_]. — Monad which we await

API (simplified)

```
transparent inline
    def async[F[_]](using am: CpsMonad[F])[T](inline f: am.Context ?=> T):F[T]
```

```
@compileTimeOnly("await should be inside async block")
def await[G[_],T,F[_]](f: G[T])(using CpsMonadCoonversion[G,F], CpsMonadContext[F]): T
```

Using implementation of type class,
available in the current scope

API (simplified)

```
transparent inline
    def async[F[_]](using am: CpsMonad[F])[T](inline f: am.Context ?=> T):F[T]
```

```
trait FutureMonad extends CpsMonad[Future] {
   type Context = FutureContex
}

async[Future] {

   // provide API, aviable only inside async
   summon[FutureContext].executionContext.submit(..)

   ......

}
```


Context function

API (simplified)

```
transparent inline
    def async[F[_]](using am: CpsMonad[F])[T](inline f: am.Context ?=> T):F[T]
```

```
trait CpsMonad[F[_]] {

   type Context <: CpsMonadContext[F]

   def pure[T](t:T):F[T]
   def map[A,B](fa:F[A])(f: A=>B):F[B]
   def flatMap[A,B](fa:F[A])(f: A=>F[B]):F[B]

   def apply[T](op: Context => F[T]): F[T]

}
```

for. try/catch support

```
trait CpsTryMonad[F[_]] extends CpsMonad[F] {

  def error[A](e: Throwable): F[A]
  def flatMapTry[A,B](fa:F[A])(f: Try[A] => F[

}
```

Transformations: (monadification, based on cps-transform)

rest in continuation passing style

$$\frac{C_F[\{a; \overline{b}\}]}{F.flatMap(C_F[a])(\_ \Rightarrow C_F[b])}$$

Transformations: control-flow (monadification, based on cps-transform)

// rules for control-flow constructions are straightforward

$$\frac{C_F[\{a; b\}]}{F\,.flatMap(C_F[a])(\_ \Rightarrow C_F[b])}$$

$$\frac{C_F[t] : t \in Constant \vee t \in Identifier}{F\,.pure(t)}$$

$$\frac{C_F[val\ a = b;\ c]}{F\,.flatMap(C_F[a])(a' \Rightarrow C_F[b_{x/x'}])}$$

$$\frac{C_F[if\ a\ then\ b\ else\ c]}{F\,.flatMap(C_F[a])(a' \Rightarrow if\ (a')\ then\ C_F[b]\ else\ C_F[c])}$$

$$\frac{C_F[try\{a\}\{catch\ e \Rightarrow b\}\{finally\ c\}]}{}$$

```
F.flatMap(
 F.flatMapTry(C_F[a]){
   case Success(v) => F.pure(v)
   case Failure(e) => C_F[b]
 }
){x => F.map(C_F[c], x)}
```

$$\frac{C_F[throw\ ex]}{F\,.error(ex)}$$

Transformations: control-flow, optimisations

Two sequential synchronous fragments are merged into one:

$$\frac{F\,.\,flatMap(F\,.\,pure(a))(x \Rightarrow F\,.\,pure(b(x))}{F\,.\,pure(b(a))}$$

Transformation of each control-flow construction are specialised again sync/async components :

if a then b else c

$$C_F[a] \neq F\,.\,pure(a) \wedge (C_F[b] \neq F\,.\,pure(b) \vee C_F[c] \neq F\,.\,pure(c))$$

$$C_F[a] = F\,.\,pure(a) \wedge (C_F[b] \neq F\,.\,pure(b) \vee C_F[c] \neq F\,.\,pure(c))$$

$$C_F[a] = F\,.\,pure(a) \wedge C_F[b] = F\,.\,pure(b) \wedge C_F[c] = F\,.\,pure(c)$$

n(bounds) ~ n(awaits):

Performance == performance of concurrent framework.

Transformations: higher-order functions

```
class Cache[K,V] {

    def getOrUpdate(k:K, whenAbsent:  =>V): V

}
```

Call by name, synonym of ()=>V

```
async[Process]{
   val k = retrieveKey(request)
   cache.getOrUpdate(k, await(fetchValue(k)))
}
```

$type(C_F[(x : T)]) = F[T]$, is not function

$type(C_F[(x : A \Rightarrow B)]) = A \Rightarrow type(C_F[B])$

Usual answer: continuation support in runtime environment.

Problem: JVM Runtime (as Js and Native now) have no continuations support

Allow programmer to define async ‘shifted’ variant of function via typecalss.

Transformations: higher-order functions

```
class Cache[K,V] {

    def getOrUpdate(k:K, whenAbsent:  =>V): V

}
```

```
class CacheAsyncShift[K,V] extends
                                    AsyncShift[Cache[K,V]]{

   def getOrUpdate[F[_]](o:Cache[K,V],m:CpsMonad[F])
                        (k:K, whenAbsent: ()=> F[V]):F[V] =
     ....

}
```

```
async[Process]{
   val k = retrieveKey(request)
   cache.getOrUpdate(k, await(fetchValue(k)))
}
```

```
summon[AsyncCache[Cache[K,V]]].getOrUpdate[F](cache,monad)(k, ()=>fetchValue(k))
  // where monad = summon[CpsMonad[F]]
```

Transformations: higher-order functions

```
class Cache[K,V] {

    def getOrUpdate(k:K, whenAbsent:  =>V): V

    def getOrUpdateAsync[F[_]](m:CpsMonad[F])(k:K, whenAbsent: ()=>F[V]): F[V]

}
```

// author aware about dotty-cps-async

```
async[Process]{
   val k = retrieveKey(request)
   cache.getOrUpdate(k, await(fetchValue(k)))
}
```

```
cache.getOrUpdateAsync[F](monad)(k, ()=>fetchValue(k))
  // where monad = summon[CpsMonad[F]]
```

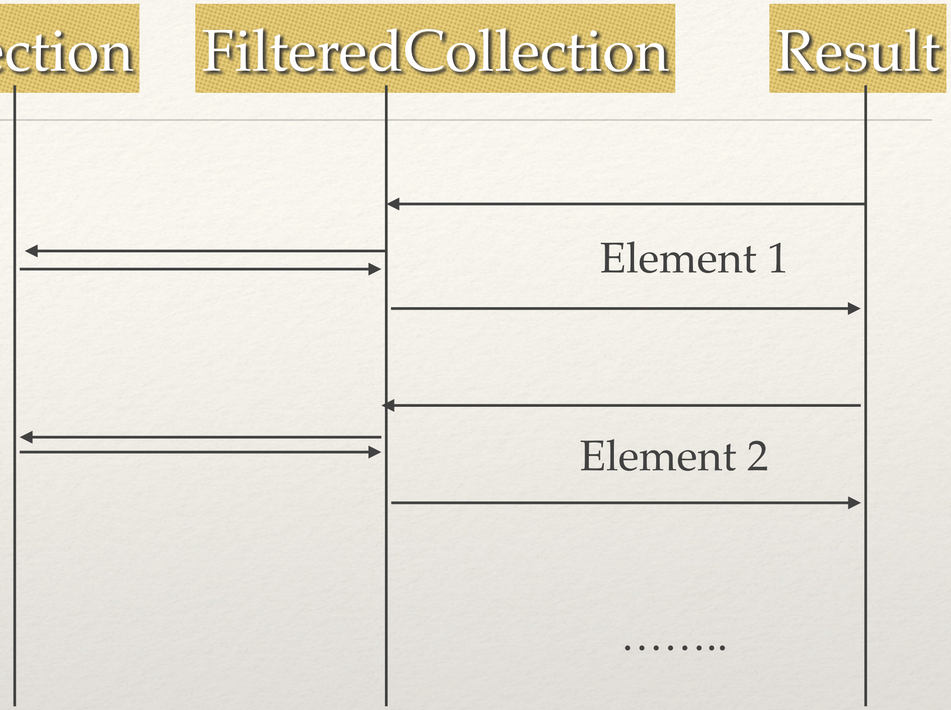
substitution classes for chain of higher-order methods

```
class Collection[A] {

  def map(f: A=>B): Collection[B]

  def withFilter(p: A=>Boolean): FilteredCollection[A]

}
```




```
for { url <- urls if await(score(url)) > limit) }
  yield await(fetchData(url))
```

```
urls.withFilter(url => await(score(url))>limit).map(await(fetchData(url)))
```

Developer expect that combination of map and filter will work at the same way, as in synchronous case.

substitution classes for chain of higher-order methods

```
class Collection[A] {

  def map(f: A=>B): Collection[B]

  def withFilter(p: A=>Boolean): FilteredCollection[A]

}
```

```
class AsyncShiftCollectin extends AsyncShift[Collection[A]]:

  def withFilter[F[_]](c:Collection[A],m:CpsMonad[F])
                          (p: A=>F[Boolean]): FilteredCollectionAsyncSubst[F,A]
```

```
class FilteredCollectionAsyncSubst[F[_],A](m:CpsMonad[F])
    extends CallChainAsyncShiftSubst[F, FilteredCollection[A], FilteredCollectionAsyncSubst[F,A]]:

  def _finishChain: FilteredCollection[A]

  def withFilter(p: A=> Boolean): this.type
  def withFilterAsync(p: A=>F[Boolean]): this.type
  def map[B](f: A => B): this.type
  def mapAsync[B](f: A=>F[B]): this.type
```

//. accumulate all ho-methods in chain, interpret via _finishChain

//categorical interpretation — left Kan extension for FilteredCollection

Transformations:

```
class Collection[A] {

  def map(f: A=>B): Collection[B]

  def withFilter(p: A=>Boolean): FilteredCollection[A]

}
```

```
class AsyncShiftCollectin extends AsyncShift[Collection[A]]:

  def withFilter[F[_]](c:Collection[A],m:CpsMonad[F])
                          (p: A=>F[Boolean]): FilteredCollectionAsyncSubst[F,A]
```

```
class FilteredCollectionAsyncSubst[F[_],A](m:CpsMonad[F])
    extends CallChainAsyncShiftSubst[F, FilteredCollection[A], FilteredCollectionAsyncSubst[F,A]]:

  def _finishChain: FilteredCollection[A]

  def withFilter(p: A=> Boolean): this.type
  def withFilterAsync(p: A=>F[Boolean]): this.type
  def map[B](f: A => B): this.type
  def mapAsync[B](f: A=>F[B]): this.type
```

//. accumulate all ho-methods in chain, interpret via _finishChain

//categorical interpretation — left Kan extension for FilteredCollection

substitution classes for chain of higher-order methods

```
class Collection[A] {

  def map(f: A=>B): Collection[B

  def withFilter(p: A=>Boolean):

}
```

```
class FilteredCollectionAsyncSubst[F[_],A](m:CpsMonad[F])
    extends CallChainAsyncShiftSubst[F, FilteredCollection[A], FilteredCollecti

  def _finishChain: FilteredCollection[A]

  def withFilter(p: A=> Boolean): this.type
  def withFilterAsync(p: A=>F[Boolean]): this.type
  def map[B](f: A => B): this.type
  def mapAsync[B](f: A=>F[B]): this.type
```

```
urls.withFilter(url => await(score(url))>limit).map(await(fetchData(url)))
```

```
summon[AsynsChift[Collection[A]].
          withFilterAsync(url => score(url).map(x => x>limit)).mapAsync(fetchData(url))
```

//. accumulate all ho-methods in chain, interpret via _finishChain

//categorical interpretation — left Kan extension for FilteredCollection

Automatic colouring.

```
async[IO] {
  val r1 = cacheService.getOrUpdate(id,
                    await(talkToServer(id)))
  val r2 = talkToServer(r1.data)
  if (await(r2).isOk) then
    await(writeToFile(r1.data))
    await(IO.println(“done”))
    true
  else
    await(IO.println(“abort”))
    false
}
```

```
async[IO] {
  val r1 = cacheService.getOrUpdate(id,
                    await(talkToServer(id)))
  val r2 = talkToServer(r1.data)
  if (r2.isOk) then
    writeToFile(r1.data)
    IO.println("done")
    true
  else
    IO.println("abort")
    false
}
```

$x : F[T]) \rightarrow await(x) : T$. implicit conversion:$F[T] \Rightarrow T$

discarded value $F[T] \rightarrow$ discarded value $await(F[T])$

Automatic colouring.

```
async[IO] {
  val r1 = cacheService.getOrUpdate(id,
                    await(talkToServer(id)))
  val r2 = talkToServer(r1.data)
  if (await(r2).isOk) then
    await(writeToFile(r1.data))
    await(IO.println(“done”))
    true
  else
    await(IO.println(“abort”))
    false
}
```

```
async[IO] {
  val r1 = cacheService.getOrUpdate(id,
                    await(talkToServer(id)))
  val r2 = talkToServer(r1.data)
  if (r2.isOk) then
    writeToFile(r1.data)
    IO.println("done")
    true
  else
    IO.println("abort")
    false
}
```

// disabled by default, enabled by import

$x : F[T]) \rightarrow await(x) : T$. implicit conversion:$F[T] \Rightarrow T$

discarded value $F[T] \rightarrow$ discarded value $await(F[T])$

Problem: this can be unsafe for some type of monads.

Automatic colouring.

$x : F[T]) \rightarrow await(x) : T$. implicit conversion:$F[T] \Rightarrow T$

discarded value $F[T] \rightarrow$ discarded value $await(F[T])$

```
def run(): F[Unit] = async[F] {
  val connection = openConnection()
  try
    while
      val command = readCommand(connection)
      val reply = handle(command)
      if (!reply.isMuted)
        connection.send(reply.toBytes)
      !command.isShutdown
    do ()
  finally
    connection.close()
}
```

This can be unsafe for some type of monads:

```
def run(): F[Unit] = async[F] {
  val connection = openConnection()
  try
    while
      val command = readCommand(await(connection))
      val reply = handle(await(command))
      if (!await(reply).isMuted)
        await(connection).send(await(reply).toBytes)
      !await(command).isShutdown
    do ()
  finally
    await(connection).close()
}
```

F[_]. Is cached, I.e. two usages refers to the same instance — Safe. (Example: Future)

F[_]. Is effect, I.e. two usage produce two effect — Unsafe. (Example: IO)

Automatic colouring.

$x : F[T]) \rightarrow await(x) : T$. implicit conversion:$F[T] \Rightarrow T$

discarded value $F[T] \rightarrow$ discarded value $await(F[T])$

```
def run(): F[Unit] = async[F] {
  val connection = openConnection()
  try
    while
      val command = readCommand(connection)
      val reply = handle(command)
      if (!reply.isMuted)
        connection.send(reply.toBytes)
      !command.isShutdown
    do ()
  finally
    connection.close()
}
```

This can be unsafe for some type of monads:

```
def run(): F[Unit] = async[F] {
  val connection = openConnection()
  try
    while
      val command = readCommand(await(connection))
      val reply = handle(await(command))
      if (!await(reply).isMuted)
        await(connection).send(await(reply).toBytes)
      !await(command).isShutdown
    do ()
  finally
    await(connection).close()
}
```

F[_]. Is cached, I.e. two usages refers to the same instance — Safe. (Example: Future)

F[_]. Is effect, I.e. two usage produce two effect — Unsafe. (Example: IO)

Automatic colouring.

$x : F[T]) \rightarrow await(x) : T$. implicit conversion:$F[T] \Rightarrow T$

discarded value $F[T] \rightarrow$ discarded value $await(F[T])$

F[_]. Is effect, I.e. two usage produce two effect — Unsafe. (Example: IO)

$memoize : F[T] \Rightarrow F[F[T]]$

```
def run(): F[Unit] = async[F] {
  val connection = await(memoize(openConnection()))
  try
    while
      val command = await(memoize(readCommand(await(connection))))
      val reply = await(memoize(handle(await(command))))
      if (!await(reply).isMuted)
        await(connection).send(await(reply).toBytes)
      !await(command).isShutdown
    do ()
  finally
    await(connection).close()
}
```

Automatic colouring.

$x : F[T]) \to await(x) : T$. implicit conversion:$F[T] \Rightarrow T$

discarded value $F[T] \to$ discarded value $await(F[T])$

F[_]. Is effect, I.e. two usage produce two effect — Unsafe. (Example: IO)

$memoize : F[T] \Rightarrow F[F[T]]$ Unsafe, when we mix synchronous and asynchronous usage.

Preliminary code analysis two prevent such mixing:

$v : F[T] var, U(v)$ – all usages of v

$u \in U(V)$

$sync(u) \iff u \sqsubset \text{await(v)}$

$async(u) \iff \neg sync(u) \wedge \neg(u \sqsubset (x = v))$

$discarded(u)$

$u_2 \sqsubset (v_1 = v_2) \Rightarrow U(v_1) \subset U(v_2)$

Safety condition:

$$\neg \exists u_1, u_2 \in U(v) : sync(u_1) \wedge async(u_2)$$

Patterns of practical usage:

https://github.com/rssh/dotty-cps-async

- Migrating from scala2 SIP22 async/await to Scala3 with dotty-cps-async. (Chat-server)
    - Monad: Future

- Utility server for integration of freeswitch with external database.
    - Monad: [X] =>> Resource[IO,X]
    - Stack [postgres, skunk, http4s].

- Port of scala library for implementing csp-channels. [scala-gopher]
    - (Non-trivial usage, inline syntax sugar api inside async block)
    - Monad: generic + ReadChannel

Issues:

- (near 1/2) are compiler issues.
- other: ergonomica and handling specialised cases.

Embedding Generic Monadic Transformer into Scala.

https://github.com/rssh/dotty-cps-async

{Questions ? }

Ruslan Shevchenko, Kyiv, Ukraine.
<ruslan@shevchenko.kiev.ua>
proofspace.id