Project Paper: Embedding generic monadic transformer into Scala

Can we merge monadic programming into mainstream?

Ruslan Shevchenko[0000−0002−1554−2019]

Institute of Software Systems
of National Academy of Sciences of Ukraine
ruslan@shevchenko.kiev.ua

Abstract. Dotty-cps-async is an open-source package that consists of scala macro, which implements generic async/await via monadic cps transform, and library, which provides monadic substitutions for higher-order functions from the standard library. It allows developers to use direct control flow constructions of the base language instead of monadic DSL for various applications. Behind well-known async/await operations, the package provides options for transforming higher-order function applications, generating call-chain proxies, and automatic coloring.

1 Introduction

It's hard to have evidence-based data about industry adoption. From the subjective observations, one of the barriers to industrial adoption of the Scala language is an unnecessary high learning curve. The tradition of using embedded DSL (*Hudak* [11]) instead of the base language leads to a situation when the 'cognitive load' of relatively simple development tasks, such as querying an extra resource, is higher than in mainstream languages. A programmer cannot use control flow constructions of base language but should learn a specific DSL and use a suboptimal embedding of this DSL, usually within monadic for comprehensions. Therefore, developers proficient in Java or TypeScript cannot be immediately proficient in Scala without additional training.

Can we provide a development environment that gives the programmer an experience comparable to the state of the art mainstream back-end programming, such as Kotlin structured concurrency, Swift async functions or F# computation expressions ? Dotty-cps-async intends to be an element of the possible answer. It provides the way to embed monadic expressions into base scala language using well-known async/await constructs, existing for nearly all mainstream programming languages. Although the main idea is not new, dotty-cps-async provides behind well-known interfaces a set of novel features, such as support of the generic monads, transformation of higher-order function applications, generation of call-chain proxies, and automatic coloring.

The package is open-source and can be downloaded from the GitHub repository https://github.com/rssh/dotty-cps-async.

The paper is organized as follows: In section 2 we briefly describe the async/await interface and the process of monadification; in 2.1 add monads parametrization and provide some examples of applying async/await transformation with non-standard monads; in 2.2 describe a support of using await inside the arguments of higher-order functions, then in 2.4 intriduce an optional automatic colouring facility and show an example of non-blocking file copying on 2.1 on page 7 elaborated with coloring on page 14. Short overview on related work in this area is in section 3.

2 Embedding generic monadic cps transform into Scala

Dotty-cps-async implements an interface similar to scala-async by (*Haller* [7]) based on optimized monadic CPS transform. It is implemented as Scala macros and provides a simple generic interface with a well-known async/await signature, slightly changed to support monad parametrization.

In simplified form this is a combination of two generic pseudofunctions:

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

def await[F[_],T]( x: F[T]):  T
```

where we can use `await` inside the `async` block .

Complete definitions are more complex, although most of this complexity is hidden from the application programmer. Let's look at it in detail and briefly describe the basics of the Scala 3 language features used in those definitions:

Full definition of `async` :

```
transparent inline
    def async[F[_]](using monad:CpsMonad[F])
                                        = InferAsyncArg(monad)

class InferAsyncArg[F[_]](am:CpsMonad[F]) {
    transparent inline def apply[T](f: am.Context ?=> T): F[T]
}
```

- `F[_]` is a type parameter of async. The compiler will deduce type parameters automatically from context if it is possible. Underscore inside square brackets in `F` means F is a higher-kinder type with one type parameter.
- `transparent inline` means a macro, which should be expanded during typing phase of the compiler.The compiler passes the typed tree to the macro and refines the output type of the result of the macro application.
- `using` clause at the beginning of the argument list, means that the compiler will substitute the appropriative parameters by the given instance of the parameter type, if such instance is defined in the current scope. Given instance can be introduced in scope via `given` clause. For example, the following definition will introduce `myMonad`. as given instance of `CpsMonad[Future]`.

```
given CpsMonad[Future]  = myMonad
```

Exists predefined function `summon[A]` which will give us a given instance of A if it's defined.

– Our `async` returns an object of the auxiliary class `InferAsyncArg` that defines `apply` method. In Scala, any object with the `apply` method can be applied as if it were a function. The auxiliary class is a work-around for the lack of multiple type parameters in function definitions. The ideal definition, without auxiliaries, would have looked as:

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

The optimizer should erase the constructor of the intermediate class.

– `f: (am.Context ?=>> T)]` is a context function. Here `f` is an expression of type `T` in context `am.Context`. Inside context function we can use `summon[am.Context]` to access a context parameter. Context provides an API that can be used only in `async` block.

– `am.Context` is a path-depended type. Type `Context` should be defined in the scope of `am`.

When the developer uses async pseudo function in the application code, the compiler will pass to the macro transformation typed tree with expanded implicit parameters and context arguments.

An example of original code:

```
async[Future] {
    val ec = summon[FutureContext].executionContext
     val x = doSomething()
     ....
}
```

Expansion passed to the macro:

```
async[Future](using FutureAsyncMonad()).apply{ (fc:FutureContext) ?=>
    val ec = fc.executionContext
    val x = doSomething()
    .....
}
```

Here we assume that `FutureAsyncMonad.` is a given monadic interface for `Future` where type `Context` is defined as alias to `FutureContext`.

Monadic operations are defined in the given `CpsMonad[F]` parameter which should implement the following typeclass:

```
trait CpsMonad[F[_]] {
  type Context <: CpsMonadContext[F]
  def pure[A](v:A): F[A]
  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]
}
```

Optionally extended by error generation and handling operations:

```
trait CpsTryMonad[F[_]] extends CpsMonad[F] {
  def error[A](e: Throwable): F[A]
  def flatMapTry[A,B](fa:F[A])(f: Try[A] => F[B]):F[B]
}
```

The `Cps` prefix here refers to the relation with continuation passing style for which transformation from direct to monadic style is closely related. This prefix is more related to a library rather than an interface: `dotty-cps-async` should coexist with generic functional programming frameworks, like `cats` or `scalaz` where other forms of `Monad` are defined, so we need to have another name, to prevent name resolution conflict in application code.

Now let us look at the full definition of `await`:

```
def await[G[_],T,F[_]](x:G[T])
                      (using CpsMonadConversion[G,F],
                             CpsMonadContext[F]):T
```

Here `G`. is a type which we awaiting, `F` – monad which is used in enclosing `async`. Note that F and G can be different; if the given instance of conversion morphism from $G[\_]$ to $F[\_]$ is defined in the current scope, then `await[F]` can be used inside `async[G]`.

This morphism is represented by the `CpsMonadConversion` interface:

```
trait CpsMonadConversion[G[_],F[_]] {
  def apply[T](gt:G[T]): F[T]
}
```

`CpsMonadContext[F]` is an upper bound of `CpsMonad[F].Context` with one operation defined:

```
trait CpsMonadContext[F[_]] {
    def adoptAwait[T](v:F[T]):F[T]
}
```

The work of `adoptAwait` is to pass information from the current monad context into the awaiting monad. For example, this can be a cancellation event in the implementation of a structured concurrency framework.

Underlying source transformation is an optimized version of monadification (Ervig, Martin and others [5]), similar to translating terms into continuations monad (Syme [9]). This translation is limited to the code block inside an async argument.

We will use notations $F.op$ as shortcut for appropriative operation over monad typeclass for F. $C_F[\![code]\!]$ is a translation of code $code$ in the context of F.

Let us recap the basic monadification transformations adopted to scala control-flow construction:

trivial	$$\dfrac{C_F[\![t]\!] \text{ where } t \text{ ia a constant or an identifier}}{F.pure(t)}$$
sequential	$$\dfrac{C_F[\![\{a;b\}]\!]}{F.flatMap(C_F[\![a]\!])(\_\Rightarrow C_F[\![b]\!])}$$
val definition	$$\dfrac{C_F[\![val\ a{=}b;c]\!]}{F.flatMap(C_F[\![a]\!])(a'\Rightarrow C_F[\![b_{[a\leftarrow a']}]\!])}$$
condition	$$\dfrac{C_F[\![if\ a\ then\ b\ else\ c]\!]}{F.flatMap(C_F[\![a]\!])(a'\Rightarrow if\ (a')\ then\ C_F[\![b]\!]\ else\ C_F[\![c]\!])}$$
match	$$\dfrac{C_F[\![a\ match\{\ case\ r_1\Rightarrow v_1...r_n\Rightarrow v_n\}]\!]}{F.flatMap(C_F[\![a]\!])\{a'\Rightarrow match\{case\ r_1\Rightarrow C_F[\![v_1]\!]...r_n\Rightarrow C_F[\![v_n]\!]\}\}}$$
while	$$\dfrac{C_F[\![while(a)\{b\}]\!]}{whileHelper(C_F[\![a]\!],C_F[\![b]\!])}$$

where *whileHelper* is a helper function, defined as

```
def whileHelper(cond: F[Boolean], body: F[Unit]):F[Unit] =
   F.flatMap(cond){ c =>
      if (c) {
         F.flatMap(body){ _ => whileHelper(cond, body) }
      } else {
         F.pure(())
      }
   }
```

try/catch	$$\dfrac{C_F[\![try\{a\}\{catch\ e\Rightarrow b\}\{finally\ c\}]\!]}{\begin{array}{l}F.flatMap(\\ \quad F.flatMapTry(C_F[\![a]\!])\{\\ \quad\quad case\ Success(v) => F.pure(v)\\ \quad\quad case\ Failure(e) => C_F[\![b]\!]\\ \quad\}\\ )\{x \Rightarrow F.map(C_F[\![c]\!], x)\}\end{array}}$$
throw	$$\dfrac{C_F[\![throw\ ex]\!]}{F.error(ex)}$$
lambda	$$\dfrac{C_F[\![a\Rightarrow b]\!]}{a\Rightarrow C_F[\![b]\!]}.$$
application	$$\dfrac{C_F[\![f(a)]\!] \text{ where } a \text{ is non-functional type}}{F.flatMap(C_F[\![a]\!])(x{=}{>}f(x))}$$
	$$\dfrac{C_F[\![f(a)]\!]\text{where } f \text{ is a lamba-function.}}{F.flatMap(C_F[\![a]\!])(x{=}{>}C_F[\![f]\!](x))}$$
	$$\dfrac{C_F[\![f(a)]\!] \text{ where } \exists f' \text{ is an external-provided shifted variant of } f}{F.flatMap(C_F[\![a]\!])(x{=}{>}f'(x))}$$

$$\text{await} \quad \frac{C_F[\![await_G(a)(m,c)]\!] \text{ where } F{=}G}{c.adoptAwait(a)}$$

$$\frac{C_F[\![await_G(a)(m,c)]\!] \text{ where } F{\neq}G}{c.adoptAwait(CpsMonadConversion[F,G](a))}$$

The mechanism for definition and substitution of shifted functions is described in 2.2.

Implementation differs from the basic transformation, by few optimizations, which are direct applications of unit monad law:

- few sequential blocks with trivial CPS transformations are merged into one:

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

- translation of control flow operators are specialized for the cases when transformations of some subterms are trivial. In the best case, control-flow construction is lifted inside monad barriers. For example, the transformation rules for `if-then-else` taking into account optimizations will look like:

$$\frac{C_F[\![if\, a\, then\, b\, else\, c]\!], C_F[\![a]\!]{\neq}F.pure(a){\wedge}C_F[\![b]\!]{\neq}F.pure(b){\wedge}C_F[\![c]\!]{\neq}F.pure(c)}{F.flatMap(C_F[\![a]\!])(v{=}{>}if(v)then\, C_F[\![b]\!]\, else\, C_F[\![c]\!])}$$

$$\frac{C_F[\![if\, a\, then\, b\, else\, c]\!], C_F[\![a]\!]{=}F.pure(a){\wedge}(C_F[\![b]\!]{\neq}F.pure(b){\vee}C_F[\![c]\!]{\neq}F.pure(c))}{if\, a\, then\, C_F[\![b]\!]\, else\, C_F[\![c]\!]}$$

$$\frac{C_F[\![if\, a\, then\, b\, else\, c]\!], C_F[\![a]\!]{=}F.pure(a){\wedge}C_F[\![b]\!]{=}F.pure(b){\wedge}C_F[\![c]\!]{=}F.pure(c)}{F.pure(if\, a\, then\, b\, else\, c)}$$

Such specializations are defined for each control-flow construction.

In the resulting code, the number of monadic binds is usually the same as a number of awaits in the program, which made performance characteristics of code, written in a direct style and then transformed to the monadic style, the same, as the code manually written by hand in monadic style.

2.1 Monads parametrization

.

Async expressions are parameterized by monads, which allows the CPS macro to support behind the standard case of asynchronous processing other more exotic applications, such as processing effects(*Syme* [19]), (*Brachthäuser* [2]), logical search (*Kiselyov and Shan* [12]), or probabilistic programming (*Adam and Ghahramani and others*[1]). Potentially, list of problem domains from F# computation expression Zoo (*Tomas and Syme* [14]) is directly applicable to dotty-cps-async.

In practice, most used monads constructed over `Future`, effect wrappers, like IO and constructions over effect classes extended by additional custom logic.

Let's look at the following example:

```
val prg = async[[X] =>> Resource[IO,X]] {
  val input = await(open(Paths.get(inputName),READ))
  val output = await(open(outputName,WRITE, CREATE, TRUNCATE_EXISTING))
  var nBytes = 0
  while
      val buffer = await(read(input, BUF_SIZE))
      val cBytes = buffer.position()
      await(write(output, buffer))
      nBytes += cBytes
      cBytes == BUF_SIZE
  do ()
  nBytes
}
```

Here `[X] =>> Resource[IO,X]` is a type-lambda, which represent the computational effect monad IO, extended by the acquisition and release of resources of type of the argument `X`.

Inside `async[[X] => Resource[IO,X]]`, `input` and `output`. resources will be automatically closed at the end of the appropriative scope.

Without async/await, one would have to write the following:

```
(
  for{
   input <- open(Paths.get(inputName),READ)
   output <- open(outputName,WRITE, CREATE, TRUNCATE_EXISTING)
  } yield (input, output)
).evalTap{ case (input, output) =>
    var nBytes = 0
    def step(): IO[Unit] = {
      read(input, BUF_SIZE).flatMap{ buffer =>
        val cBytes = buffer.position()
        write(output, buffer).flatMap{ _ =>
          nBytes += cBytes
          if (cBytes == BUFF_SIZE)
             step()
          else
             IO.pure(())
        }
      }
    }
    step().map{ _ => nBytes }
}
```

The next example illustrate a monadic representation of combinatorial search. Monad `[X] =>> ReadChannel[Future,X]` represent a CSP[Communicating Sequential Processes]-like channel *Hoare* [10], where monadic combinators apply the functions over the stream of possible states.

We want to solve the classical N-Queens puzzle: placing N queens on a chessboard so that no two figures threaten each other.

Let us represent the chessboard state as a vector of queens second coordinates *queens* with additional helper method `isUnderAttack` with obvious semantics. The i-th queen is situated at $(i, queens[i])$ location.

```
type State = Vector[Int]

extension(queens:State) {

  def isUnderAttack(i:Int, j:Int): Boolean =
    queens.zipWithIndex.exists{ (qj,qi) =>
      qi == i || qj == j || i-j == qi-qj || i+j == qi+qj
    }

  def asPairs: Vector[(Int,Int)] =
   queens.zipWithIndex.map(_.swap)

}
```

Function `putQueen`. generate from one starting state a channel of possible next states. `async[Future]` in `putQueen` spawns a concurrent process for enumerating the next possible steps in N-Queens solution.

```
def  putQueen(state:State): ReadChannel[Future,State] =
  val ch = makeChannel[State]()
  async[Future] {
    val i = state.length
    if i < N then
      for{ j <- 0 until N  if !state.isUnderAttack(i,j) }
        ch.write(state appended  j )
    ch.close()
  }
  ch
```

And we can recursive explore all possible steps with help of the `solution` function, which returns the stream of finish states:

```
def solutions(state: State): ReadChannel[Future,State] =
 async[[X] =>> ReadChannel[Future,X]] {
   if(state.length < N) then
     val nextState = await(putQueen(state))
     await(solutions(nextState))
   else
     state
 }
```

(runnable version is available at `https://github.com/rssh/scala-gopher/blob/master/shared/src/test/scala/gopher/monads/Queens.scala`)

The computation is directed by reading from the stream of solutions. In `putQueen`, the computations inside the loop will suspend after each write to the

output channel, until all descendent states are explored. The suspension point `await` is hidden inside `ch.write` inside for loop.

`ch.write` is defined in `ReadChannel[F,A]` as

```scala
transparent inline def write(inline a:A): Unit =
     await(awrite(a))(using CpsMonad[F])
```

transparent inline macros in scala are expanded in code at the same compiler phase before enclosing macro, so async code transformer process this expression in for loop instead `ch.write`.

In such way `solutions(State.empty).take(2)` will return the first two solutions without performing a breadth-first search.

2.2   Translation of higher-order functions

Supporting cps-transformation of higher-order functions is important for a functional language because it allows `await` expression inside loops and arguments of common collection operators. As an example, in the previous section await inside `for` loop was used for asynchronous channel write. Using await inside higher-order function enables idiomatic functional style, such as

```scala
val v = cache.getOrElse(url, await fetch(url))
```

Local cps transform changes the type of a lambda function. If the runtime platform supports continuations, we can keep the shape of the arguments in the application unchanged by defining 'monad-escape' function transformers, which can restore the view of $cps(f) : A \Rightarrow F[B]$ back to $A \rightarrow B$.

However, for the platform without continuation support, higher-order functions from other module is a barrier for local async transformations. For the time of the writing of this article, none of the available Scala runtimes (i.e. JVM, Js, or Native) have continuations support. For JVM exists a plan to implement continuations support via Project Loom [13], but it is not available for production use yet. JavaScript runtime is significantly smaller than JVM, and runtime semantics is precisely defined asynchronous.

For those runtimes and for cases when semantic of monad does not allow us to build such escape function, dotty-cps-async implements limited support of higher-order functions. Macro performs a set of transformations, which allows developers to describe the substitution for the origin higher-order function in their code.

Let us have a first-order function: $f : A \Rightarrow B$ which have form $\lambda x : code_f(x)$ and higher-order method $o.m : (A \Rightarrow B) \Rightarrow C..$ For simplicity, let's assume that o is reference to external symbol and not need cps-transformation itself, since we want to show only function call transaltions here. Async transformation transform $code : X$ into $C_F[code] : F[X]$, where F is our monad.

Let us informally describe a set of transformations used to translate function call:

unchanged
$$\frac{C_F[\![o.m(f)]\!], C_F(code_f) == F.pure(code_f)}{F.pure(o.m(f))}$$

monadic
$$\frac{C_f[\![o.m(f)]\!], B = F[B']}{F.pure(o.m(\lambda x{:}A \Rightarrow CpsMonad[F].flatMap(C_F[\![code_f]\!]))(identity))}$$

asyncShift-fo
$$\frac{C_F[\![o.m(f)]\!], \exists asyncShift_O = summon[AsyncShift[O]]:}{asyncShift_O.m{:}O \times CpsMonad[F] \Rightarrow (A \Rightarrow F[B]) \Rightarrow F[C]}$$
$$\frac{}{asyncShift_O.m(monad,o)(x \Rightarrow C_F[\![code_f(x)]\!]}$$

asyncShift-o
$$\frac{C_F[\![o.m(f)]\!], \exists asyncShift_O = summon[AsyncShift[O]]:}{asyncShift_O.m : O \Rightarrow (A \Rightarrow F[B]) \Rightarrow F[C]}$$
$$\frac{}{asyncShift_O.m(o)(x \Rightarrow C_F[\![code_f(x)]\!]}$$

inplace-f
$$\frac{.C_f[\![o_m(f)]\!], \exists m_{asyncShift} \in methods(O){:}CpsMonad[F] \Rightarrow (A \Rightarrow F[B]) \Rightarrow F[C]}{m_{asyncShift}(C_F[\![code_f(x)]\!])}$$

inplace
$$\frac{.C_f[\![o_m(f)]\!], \exists m_{asyncShift} \in methods(O){:}(A \Rightarrow F[B]) \Rightarrow F[C]}{m_{asyncShift}(C_F[\![code_f(x)]\!])}$$

Explanation:

- In **u**nchanged case we can leave the call unchanged because no cps transformation was needed. Note, that this handle a case when we have no acccess to the source of the f argument: if x is defined externally it can't contains `await`.
- In **m**onadic case is possible to reshape function arguments, to keep the same signature to receive.
- case **a**syncShift-fo define call substitution: If we have instance $asyncShift_O$ of marker typeclass `AsyncShift[O]`, which provide a substitution methods. m_{shift} with additional parameter list where we pass original object and target monad.

  —.e. let we have class with higher-order function, for example:

```
class  Cache[K,V]  {
    def getOrUpdate(k: K, whenAbsent:  =>V): V
}
```

and want to use this class in asynchronous environment like next code fragment:

```
async[Future] {
    ...
    cache.getOrUpdate(k, await(fetchValue(k)))
}
```

where fetchValue return `Future[V]`.

For defined async substitution for `getOrUpdage` method we should define a given instance of marker typeclass $asyncShift_{Cache}$ when shifted method is defined.

```
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] =
      ....

}
given CacheAsyncShift[K,V]()
```

Here substitution method have one additional list of arguments `(o:Cache[K,V],m:CpsMonad[F])`, where we pass original object itself and our target monad. Since monad parameter is generic, we also have additional type parameter `F`. Functional call will be transformed to

```
 summon[AsyncCache[Cache[K,V]]]
     .getOrUpdate[Future](cache,monad)(k, ()=>fetchValue(k))
```

and `summon[AsyncCache[Cache[K,V]]]` will be resolved to `CacheAsyncShift[K,V]` by implicit resolution rules, so resulting expression will be

```
   CacheAsyncShift[K,V]()
       .getOrUpdate[Future](cache,monad)(k, ()=>fetchValue(k))
```

- case **a**syncShift-o is a modification of the previous rule for the situation when our target monad already parametrizes the substitution class, so we do not need an extra type parameter and monad instance in the additional parameter list.
- case **i**nplace-f and **i**nplace describe a situation when the author of a class is aware of the existence of dotty-cps-async and define a shifted method in the same scope as the original method. By convention, such shifted methods are prefixed with `Async` suffix.
Example:

```
class  Cache[K,V]  {

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

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

}
```

Such substitutors for most higher-order functions from Scala standard library are supplied with dotty-cps-async runtime. Also, developers can provide their substitution for third-party libraries.

The return type of substituted function can be:

- C, the same as the origin
- $F[C]$ origin return type wrapped into the monad.
- `CallChainAsyncShiftSubst[F,C,F[C]]`. This is a special marker interface for call chain substitution, which wich will be described later.

– Exists method in O with name `m_async` or `mAsync` which accept shifted argument $f : A \Rightarrow F[B]$. The conventions for the return type are the same as in the previous case. This case is helpful for the fluent development of API, which is accessible in both synchronous and asynchronous forms.
– If none of the above is satisfied, the macro generates a compile-time error.

These rules are extended to multiple parameters and multiple parameters list, assuming that if we have one higher-order async parameter, then all other parameters should also be transformed, having only one representation of the asynchronous method.

Functions with order more than two can be supported if we will extend typing rules for shifted functions arguments: argument of type $A \to B \to C$ will have asynchronous form with type $A \to F[B \to F[C]]$.

2.3 Call-chain substitutions

As shown in previous section, one of the possible variant of return method of substituted higher-order function is
`CallChainAsyncShiftSubst[F[_],B,F[B]]`. The developer can use this variant to delay applying $F[\_]$ till the end of the call chain.

For example, let's look at the next block of code:

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

wich is desugared as

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

The programmer expects that the behavior of the code should be the same, regardless of using `await` inside a loop, so the list of URL-s will be iterated once. However, if the result of `withFilter` has form `F[List.WithFilter]`, two iterations are performed - one for filtering the list of URLs and the other over the filtered list to perform fetching data. User objects for call-chain substitution can accumulate the sequence of higher-order functions in one batch and perform iteration once.

This block of code is transformed as follows:

```
summon[AsyncShift[List[String]].withFilter[F](urls,m)(
     url => m.map(score(url))(x=>x>limit)
   )                 // CallChainAsyncShiftSubst[F,WithFilter,F[A]]
  .mapAsync(url => fetchData)    // function added to builder
  ._finishChain()                        // finally eval all.
```

2.4 Automatic coloring

.

Automatic coloring is the way to free the developer from writing boilerplate await statements. Since most industrial code is built with some asynchronous framework, await expressions are often situated literally in each line. Those expressions do not carry out business logic; when writing code, we should not care how an object is coming to code, synchronously or asynchronously, the same as we do not care how memory to our objects should be allocated and deallocated. Apart from performance-critical applications (e.g., developing a web server for which the developer requires complete control of low-level concurrency details), it is preferable to use higher-level constructs hiding low-level details. . When writing business logic using some low-level system framework, we expect that framework provides a reasonable generic concurrency model and abstracts away from manual coloring.

We can provide implicit conversion from $F[T]$ to T. Can we make such conversion safe and preserve semantics with automatic coloring?

It is safe when $F[\_]$ is a `Future`, because multiple calls of `await` on `Future` produce the same effect as one call – after the result value will be available after first await, it will be returned immediately after other calls of awaits of the same `Future`. I.e., we can say, that `Future` is cached.

For other types of monads, where each `await` can perform a new computation, such implicit conversion will be unsafe - the behavior of the following code snippets will be different:

```
val a = x
f1(await(a))
f2(await(a))
```

and

```
val a = await(x)
f1(a)
f2(a)
```

To overcome this, we can provide memoization of execution, by embedding the memoization into the transformation of val definitions.

Let us have block of code { **val** v =expr; $tail_v$ }, *expr* return value of type $F[T]$ and exists `CpsMemoization[F]` with method `apply[T](F[T]):F[F[T]]`.

Cps transformer can check the variable type and rewrite this to.

`summon[CpsMonad[F]].flatMap(CpsMemoization[F]`$(expr))(v1 \Rightarrow cps(tail_{v1}))$

Implicit conversions are often criticized as an unsafe technique, which can be a source of bugs and maintainability problems. In our case, uncontrolled usage of implicit conversions can break the semantics of building complex effects, where some building parts can be automatically memoized. Dotty-cps-async implements preliminary analysis of automatically generated conversion, which emits errors when detecting potentially unsafe usage.

To make transformation safe, we should check that developer cannot pass memoized value to API, which expects a delayed effect. Preliminary analysis ensures that all usages of memoized values are in synchronous context by forcing the next rules:

– If some variable is used only in a synchronous context (i.e., via await), the macro will color it as synchronous (i.e., cached if used more than once).
– If some variable is passed to other functions as an effect - it is colored as asynchronous (i.e., uncached).
– If the variable is simultaneously used in synchronous and asynchronous contexts, we cannot deduce the programmer's intention, and the coloring macro will report an error.
– If the variable, defined outside of the async block, is used in synchronous context more than once - the macro also will report an error.

Behind providing implicit conversion, automatic coloring should also care about value discarding: expressions that provide only side-effects are not an assignment to some value but discarded. When we do automatic coloring, the monad with side-effect generation becomes the value of an expression. So, we should also transform statements with value discard to insert awaits there. Dotty-cps-async interfaces has a `ValueDiscard[T]` typeclass. The statement inside async block can discard value of type T only if exists implementation of `ValueDiscard[T]` interfaces: in such case macro transforms value discard into `summon[ValueDiscard[T]].discard(t)`.

A special marker typeclass `AwaitValueDiscard[F[T]]` is used when this value discard should be a call to await.

If we will apply automatic coloring to our example with copying file, we will see that difference between synchronous and asynchronous code become invisible.

```
val prg = async[[X] =>> Resource[IO,X]] {
  val input = open(Paths.get(inputName),READ)
  val output = open(outputName,WRITE, CREATE, TRUNCATE_EXISTING)
  var nBytes = 0
  while
     val buffer = read(input, BUF_SIZE)
     val cBytes = buffer.position()
     write(output, buffer)
     nBytes += cBytes
     cBytes == BUF_SIZE
  do ()
  nBytes
}
```

3 Related work

The idea of 'virtual' program flow encapsulated in a monad is tracked to (*Claessen* [4]), which become a foundation for Haskell concurrent library. Later `F#` computation expressions were implemented as further development of do-notation.

Furthermore, `C#` moves async/await from virtual monadic control-flow to 'normal control-flow,' which becomes a pattern for other languages (*Syme* [18]). (*Tomas and Syme* [14]) provides an overview of computation expression usage in different areas.

Generic monadic operation pairs [reify/reflect] and links between monadic and cps transformations are described in (*Filinski* [6].)

In scala land, the first cps transformer was implemented as a compiler plugin (*Rompf, Maier, Odersky*)[17]. It provides quite a powerful but complex interface based on delimited continuations. Scala-Async (*Haller* [7]) provides a more familiar interface for developers for organizing asynchronous processing by compling async control flow to state machines. The main limitation is the absence of exception handling. Recently, a Lightbend team moved implementation of scala-async from macro to compiler plugin and extended one to support external 'Future systems' such as IO or Monix. Although dotty-cps-async is internally based on another type of transformation, it can be viewed as an extension of the scala-async interface for the next language version with a similar role in the Scala ecosystem. The new facilities are a generic monad interface, support of try/catch, and limited support for higher-order functions.

In (*Haller and Miller* [8]) scala-async model is extended to handle reactive streams. Scala coroutines (*Prokopec and Fengyun* [15]) provides a model which allows to build async/await interface on top of coroutines. Scala Virtualized (*Rompf, Amin, Moors, Haller* [16]) devotes to solving a more general problem: providing deep embedding not only for monadic costructions but for arbitrary language. Scala Effekt (*Brachthäuser and others* [2]) allows interpretation of effect handlers inside control monad with delimited continuations. The same authors released a monadic reflection library for scala3 (*Brachthäuser and others* [3]), using the capabilities of yet not released support for continuations in JVM in (*Project Loom*[13]). This approach can be a convenient way for implementing async/await like functionality for the future versions of the JVM, for monads, which can be implemented on top of the one-shot continuations. Note, that example of combinatorial search from section 2.1 on page 7, cannot be implemented with the runtime monadic because combinatorial search, as other applications of non-determenism requires multiple-shot continuations, where captured continuations can be invoked more than once.

4 Conclusion and further work.

Prerelease versions of dotty-cps-async have been available as open-source for more than a year, and we have some information based on actual usage in application projects. Macro library is used in open-source chatbot server with `Future` based stack. An experimental proofspace.id internal microservice for connecting PostgreSQL database to VoIP server is built with the help of dotty-cps-async with `cats-effect` stack. The most frequently used monad here is `[X] =>> Resource[IO,X]` (the common pattern is to acquire database connection from the pool for each request). Also, we can point to the port of scala-gopher

to scala3, which provides an implementation of concurrent sequential process primitives on top of generic monadic API and experimental TypeLevel project, which brings the direct style into the cats-effect ecosystem.

Overall feedback is mostly positive. Reported issues usually have form of inability to compile some specific tree and often tracked down to the issues in the compiler. Note, that scala3 compiler also was higthly experimental at this time. The number of reports about ergonomic and diagnostics is relative low. Particulary this is because scala3 macros is applied after typing, so usually type errors are catched before macro applications. In some cases we have an error during retyping of transformed tree: for this case the path for better error diagnostics was submitted and merged into scala3 compiler. The work of the patch is extending error message by showing the tree, which was transformed by current macros, in addition to the code position, when -explain compiler option is enabled.

Performance issues are not reported at all. Dotty-cps-async does not provide its own asynchronous runtime but is used with some existing runtime. Furthermore, if synchronous code remains unchanged and asynchronous code does not add extra runtime operations, then any benchmark will show a performance of the underlying runtime.

Ability to use direct control-flow on top of some library is a one half of programming experience. The other part is the library itself. Currently, we have a set of asynchronous scala runtimes with a different sets of capabilities and it would be interesting to build some uniform facilities for concurrency programming. One of the open questions is to extend eager Future runtime to support structured concurrency; Problem from the other side – users of effect stacks, such as IO, need to wrap impure API into effects. Can we automate this process?

Also it will be interesting to adopt using of runtime continuations instead compile-time transformations for some type of monads, which will eliminate the need to manually write substitutions for higher-order functions on continuation-enabled platforms. Another direction is the expressivity of internal language, which can be extended by building appropriate wrapper control monad.

References

1. Adam, Ghahramani, Z., Gordon, A.D.: Practical probabilistic programming with monads. SIGPLAN Not. **50**(12), 165–176 (Aug 2015). https://doi.org/10.1145/2887747.2804317, https://doi.org/10.1145/2887747.2804317
2. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. J. Funct. Program. **30**, e8 (2020). https://doi.org/10.1017/S0956796820000027, https://doi.org/10.1017/S0956796820000027
3. Brachthäuser, J., Boruch-Gruszecki, A., Odersky, M.: Representing monads with capabilities. Tech. rep., Higher-Order Programming with Effects (HOPE) (2021), https://github.com/lampepfl/monadic-reflection
4. Claessen, K.: A poor man's concurrency monad. Journal of Functional Programming **9**(3), 313–323 (1999). https://doi.org/10.1017/S0956796899003342

5. Erwig, M., Ren, D.: Monadification of functional programs. Sci. Comput. Program. **52**(1–3), 101–129 (Aug 2004). https://doi.org/10.1016/j.scico.2004.03.004, `https://doi.org/10.1016/j.scico.2004.03.004`

6. Filinski, A.: Representing monads. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 446–457. POPL '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/174675.178047, `https://doi.org/10.1145/174675.178047`

7. Haller, P., Lightbend: scala-async (2013), `https://github.com/scala-async/scala-async`

8. Haller, P., Miller, H.: A formal model for direct-style asynchronous observables. CoRR **abs/1511.00511** (2015), `http://arxiv.org/abs/1511.00511`

9. Hatcliff, J., Danvy, O.: A generic account of Continuation-Passing Styles. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 458–471. POPL '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/174675.178053, `https://doi.org/10.1145/174675.178053`

10. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International Series in Computer Science, Prentice Hall (1985), `http://www.usingcsp.com/cspbook.pdf`

11. Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv. **28**(4es), 196–es (dec 1996). https://doi.org/10.1145/242224.242477, `https://doi.org/10.1145/242224.242477`

12. Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). SIGPLAN Not. **40**(9), 192–203 (Sep 2005). https://doi.org/10.1145/1090189.1086390, `https://doi.org/10.1145/1090189.1086390`

13. Oracle: Project loom (2018-2022), `https://openjdk.java.net/projects/loom/`

14. Petricek, T., Syme, D.: The F# computation expression zoo. In: Proceedings of Practical Aspects of Declarative Languages. PADL 2014 (2014)

15. Prokopec, A., Liu, F.: Theory and Practice of Coroutines with Snapshots. In: Millstein, T. (ed.) 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, pp. 3:1–3:32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.3, `http://drops.dagstuhl.de/opus/volltexte/2018/9208`

16. Rompf, T., Amin, N., Moors, A., Haller, P., Odersky, M.: Scala-virtualized: Linguistic reuse for deep embeddings. Higher Order Symbol. Comput. **25**(1), 165–207 (Mar 2012). https://doi.org/10.1007/s10990-013-9096-9, `https://doi.org/10.1007/s10990-013-9096-9`

17. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In: Hutton, G., Tolmach, A.P. (eds.) Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009. pp. 317–328. ACM (2009). https://doi.org/10.1145/1596550.1596596, `https://doi.org/10.1145/1596550.1596596`

18. Syme, D.: The early history of F#. Proc. ACM Program. Lang. **4**(HOPL) (Jun 2020). https://doi.org/10.1145/3386325, `https://doi.org/10.1145/3386325`

19. Wadler, P., Thiemann, P.: The marriage of effects and monads. ACM Trans. Comput. Logic **4**(1), 1–32 (Jan 2003). https://doi.org/10.1145/601775.601776, `https://doi.org/10.1145/601775.601776`