УкрПрог-2024

Representation of monadic effects in the non-monadic form.

Ruslan Shevchenko
<ruslan@shevchenko.kiev.ua>

Introduction: what is a monadic effect ?

Reminder: Monads

$T : End(C)$ - Endofunctor on C, $(f : X \to Y) \to (f' : F[X] \to F[Y])$; map

$\eta : 1_C \to T$ - unit (pure, return)

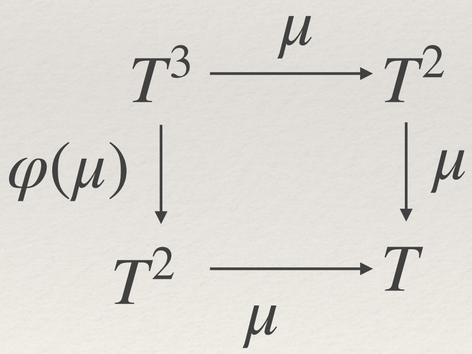
$\mu : T^2 \to T$ - join (flatten, join)

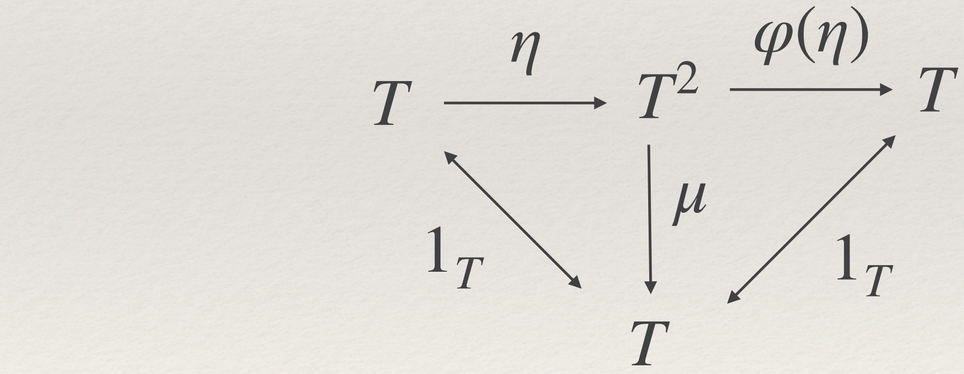
$\varphi : T \to T, \varphi(f) = \mu \circ T(f)$ - flatMap, bind

$\mu \circ \mu = \mu \circ \varphi(\mu)$

$\mu \circ \eta = 1_T$

$\mu \circ \varphi(\eta) = 1_T$





Introduction: what is a monadic effect ?

Reminder: Monads

Scala:

```
trait Monad[F[_]]:

  def pure[A](a:A): F[A]

  def flatten[A](t:F[F[A]]): F[A] =
     t.flatMap(identity)

  def flatMap[A,B](fa:F[A])(f: A=>F[B]): F[B]
```

η

μ

φ

```
def flatMap'[A,B](f: A=>F[B]): F[A]=>F[B]
```

Haskell:

```
class Monad m where
  (>>=)  :: m a -> (  a -> m b) -> m b // bind
  return ::   a                 -> m a
```

φ

η

Introduction: what is a monadic effect ?

Reminder: Monadic effect

```
trait Monad[F[_]]:

  def pure[A](a:A): F[A]

  def flatten[A](t:F[F[A]]): F[A] =
     t.flatMap(identity)

  def flatMap[A,B](fa:F[A])(f: A=>F[B]): F[B]
```

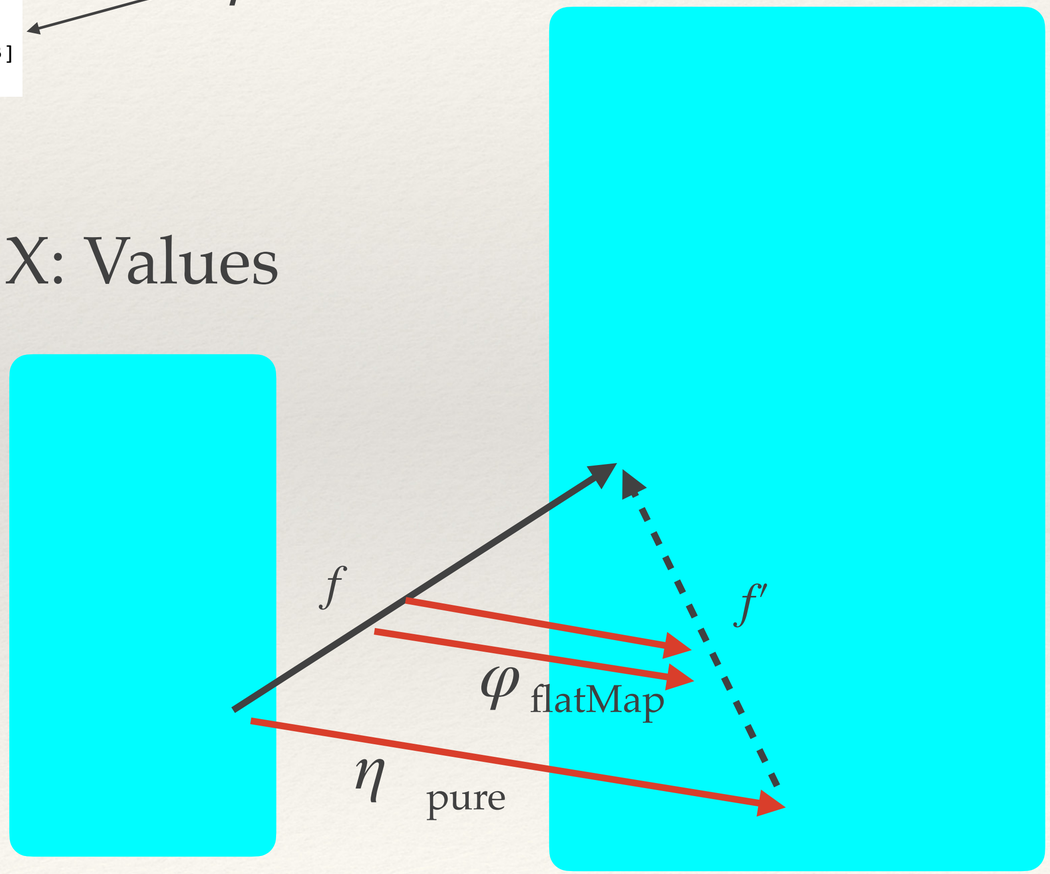
η

μ

φ

F[X]: Values + Effects.

X: Values



Introduction: what is a monadic effect ?

Reminder: Monadic effect

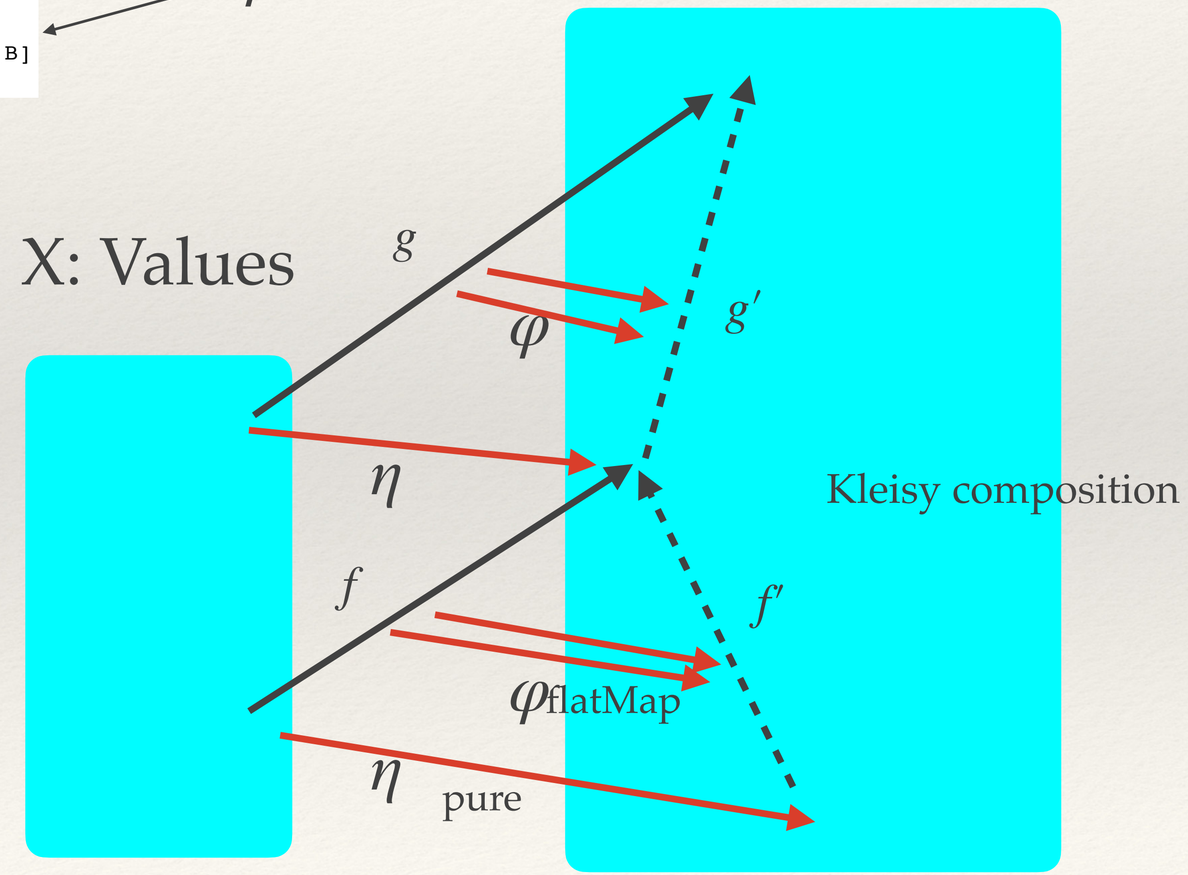
```
trait Monad[F[_]]:

  def pure[A](a:A): F[A]

  def flatten[A](t:F[F[A]]): F[A] =
     t.flatMap(identity)

  def flatMap[A,B](fa:F[A])(f: A=>F[B]): F[B]
```

η

μ

φ

F[X]: Values + Effects.



Introduction: what is a monadic effect ?

Reminder: Monadic effect

```
trait Monad[F[_]]:

  def pure[A](a:A): F[A]

  def flatten[A](t:F[F[A]]): F[A] =
     t.flatMap(identity)

  def flatMap[A,B](fa:F[A])(f: A=>F[B]): F[B]
```

η

μ

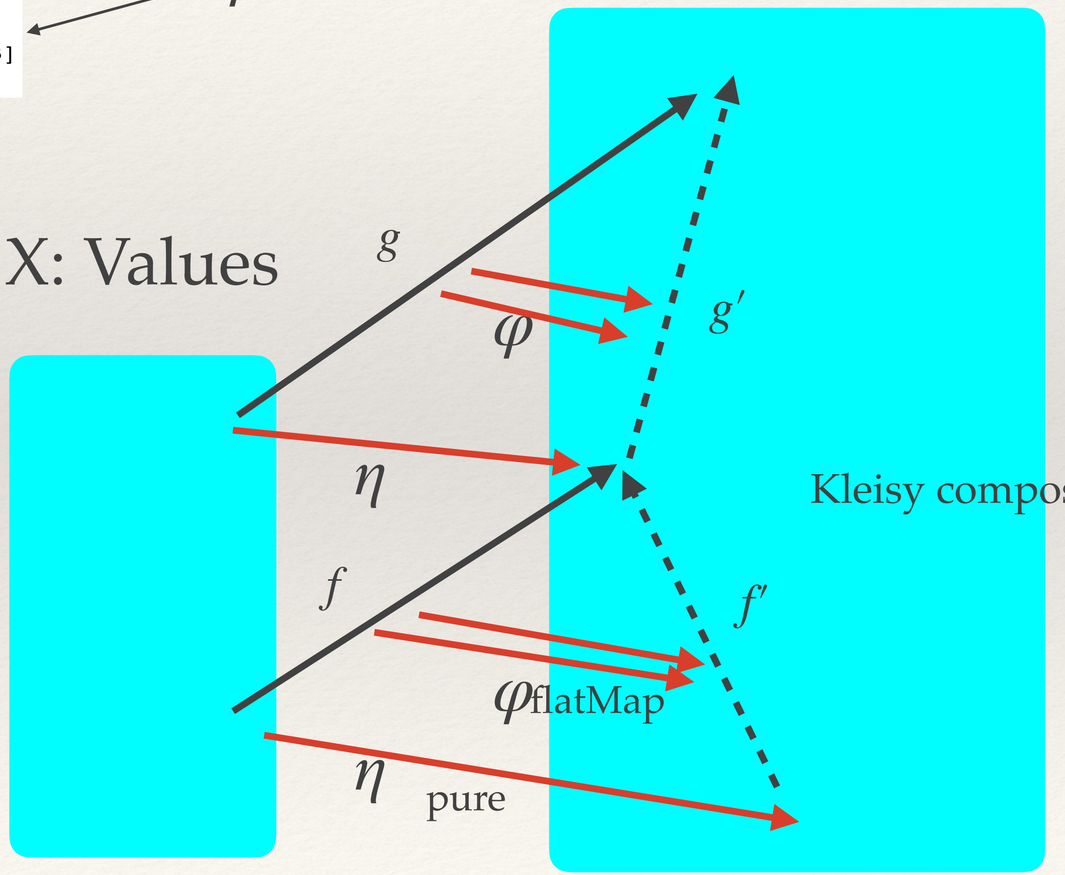
φ

F[X]: Values + Effects.

$f : X \to F[X]$

Program

X: Values



Try[_]

IO[_]

Task[_]

Future[_]

Choice[_]

List[_]

φ flatMap: - compose programs

η pure: - trivial program which return result

Introduction: what is a monadic effect ?
Reminder: Monadic effect

```
trait Monad[F[_]]:

  def pure[A](a:A): F[A]

  def flatten[A](t:F[F[A]]): F[A] =
     t.flatMap(identity)

  def flatMap[A,B](fa:F[A])(f: A=>F[B]): F[B]
```

$$f : X \to F[X]$$

Program

φ flatMap: - compose programs

η pure: - trivial program which return result

Effects / Monads:

Try[X] - result or exception

IO[_] - result with input / output trace

Task[_]. - frozen computation

Future[_] - running computation

Choice[_] - evaluate few times

List[_] - evaluate for each possible input

From monad to effect systems:

- We have multiple effects in the same time:
  - Can throw exception and input/output and have asynchronous operations.
    - Task[Try[IO[_]]] - monad transformer approach.
    - IO_{cats} - GOD monad
    - Eff[…..] - compose monad from multiple effects which implements some type class

Monadic/Direct syntax

- Monadic Syntax:
  - Effect represents by monads
    - Problem: extra syntax layer, DSL for composing

```
def log(message: String):IO[Unit] = ….
```

- Direct Syntax
  - Effect represented as special operations, interpreted by language, enclosing monad is implicit

```
def log(message: String): Unit = ….
```

    - Problem: not possible for all runtimes and effects.
      - Async in pre-Loom JVM, JS
      - Logic (Choice effect (multishot continuation)).

(Monads: async/await <=> monadic inside expression)

How to translate async/await into monadic form:

$M[A]$ Computation over A

```
async[M] {

  operations1
    ...

  val x = await(expr1)

  operations2
    ...

}
```

```
val m = summon[CpsMonad[M]]
m.apply{ (ctx) =>
  operations1
    …
  expr1.map{ x =>
    operations2
      ...
  }
}
```

Dotty-cps-async

How to translate async/await into monadic form:

- Recursive apply rules for each language construct wich translate Scala Typed AST into CpsTree (IR)
- Optimize CpsTree (merge synchronous parts into one, .. etc)
- Generate Scala Typed AST back.

$$\frac{[[if(cond)\,x\,elsey]]}{m\,.flatMap(cond)\{cond => if(cond)[[x]]else[[y]]\}} \qquad cond \in Async, x \in Async, y \in Async$$

Full set of rules: TFP2022 preprint

https://rssh.github.io/dotty-cps-async/References.html#presentations

Live code: show few constructions in macroses

Asynchronous control flow - Scala industrial usage.

```
def myFun(id: IDType):IO[Data] =
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")
        else
          IO.println("abort")
} yield r2.data
```

Monadic DSL on top of some effect system or. Future

```
def myFun(id: IDType):IO[Data] =
reify[IO] {
  val r1 = cacheService.getOrUpdate(id,
                    reflect(talkToServer(id)))
  val r2 = talkToServer(r1.data)
  if (r2.isOk) then
    reflect(writeToFile(r1.data))
    reflect(IO.println(“done”))
  else
    reflect(IO.println(“abort”))
  reflect(r2.data)
}
```

"Mainstream" Control-Flow

over some effect system or Future

Asynchronous control flow - Scala industrial usage.
+ automatic colouring

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

Problem: syntax noise, too many reflect

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

With automatic colouring

Monadic/Direct syntax

- Monadic Syntax:
  - Effect represents by monads
    - Problem: extra syntax layer, DSL for composing

```
def log(message: String):IO[Unit] = ….
```

- Direct Syntax
  - Effect represented as special operations, interpreted by language, enclosing monad is implicit

```
def log(message: String): Unit = ….
```

    - Problem: not possible for all runtimes and effects.
      - Async in pre-Loom JVM, JS
      - Logic (Choice effect (multishot continuation)).
- Direct context encoding.
  - Effect represented as context function.
  - Dotty-cps-async: first implementation via compile-time transform.

```
def log(message: String): CpsDirect[IO] ?=> Unit = ….
```

Asynchronous control flow - Scala industrial usage.
+ automatic colouring

```
def myFun(id: IDType):IO[Data] =
reify[IO] {
  val r1 = cacheService.getOrUpdate(id,
                        talkToServer(id))
  val r2 = reflect(talkToServer(r1.data)
  if (r2.isOk) then
    writeToFile(r1.data)
    IO.println("done")
  else
    IO.println("abort")
  reflect(r2.data)
}
```

Problem: not always applicable,
implicit conversions are not always intuitive

```
def myFun(id: IDType)(using IOContext):Data =
{
  val r1 = cacheService.getOrUpdate(id,
                        talkToServer(id))
  val r2 = reflect(talkToServer(r1.data)
  if (r2.isOk) then
    writeToFile(r1.data)
    IO.println("done")
  else
    IO.println("abort")
  r2.data
}
```

Direct context encoding

Dotty-cps-async

- Macros which provide limited cps-transform for expressions.
- Scala-compiler plugin which transform direct context encoding into monadic form.
- Open source: https://github.com/rssh/dotty-cps-async
    - Thanks for listening.
    - Questions?