@author Sylvain Ductor, 2022

# Norme

L'objet central de GreenTea est la norme. Une norme est:

- Une fonction d'un type $A$ vers un type $B$,
- qui exploite un canal de communication entre un couple d'agents : $i$ /> $j$,
- qui exporte un model catégoriel de A vers B,
- qui requiert (en entrée) un contexte logique/un état, , dépendant de A et du canal
- qui garantit (en sortie) un contexte logique/un état, , dépendant de B et du canal

La norme à été formalisé pour être à l'intersection de

- la théorie des catégories : les normes formes une catégorie monoidale, elles sont assemblables avec les opérateurs «andThen» , «and», «or» et «xor», - cf https://stringfixer.com/fr/Sieve_(category_theory) ),
- des systèmes multi-agent : elle représente un message associé chaque couple d'agent, en fonction d'un contexte et d'une entrée fournit - autrement dit un protocole à état,
- de la sémantique de Kripke : elle représente la transition entre deux mondes chacun caractérisé par l'ensemble de propriétés qu'il vérifie, respectivement, et .

# Interpreteur

L'implémentation à été conçus par maximiser la malléabilité et la réexpolitabilité de la norme. Pour cela chaque «aspect» de la norme à été «habillé» par un type la généralisant tout en la soumettant à un «contrat» vis à vis de son utilisation. Tous ces types abstrait sont réunis au sein d'un unique objet, *l'interpréteur*, qui doit être fournit à la norme pour que celle-ci soit exécuté. Il s'agit de:

- : le type des données. Ainsi une norme n'est pas une fonction de $A$ => $B$ mais plutôt de *[A] => [B].* peut matérialiser le fait que A peut résulter d'un calcul et par là être une exception, peut être une liste d'objet, une distribution statistique, un accès distant, un objet crypter homomorphiquement, ...,
- , le pendant de pour les données logique ( , ),
- : l'environnement de calcul. Immédiat ou paresseux, sur une plateforme tel que Jade ou Akka ou sur une infrastructure tel que Cuda ou JNI,
- : l'extracteur qui permet d'obtenir une valeur depuis l'environnement de calcul,
- |> : le canal de communication,

- , Ω, les *holons* faisant de cette norme un contrat. Ils généralisent les fonctions définient entre tout couples d'agents en une fonction définit pour la société grâce à la spécification d'un protocole de consensus (e.g. calcul parallèle, vote, contract-net protocol, etc)
- : l'environnement partagé par les agents impliqué dans le contrat résultant de l'utilisation de la norme. Il doit être accessible en lecture et en écriture, et cela de façom individualisé à chaque couple d'agent,
- Δ traduit *a posteriori* une norme en un modèle catégoriel permettant soit le prototypage (e.g. une norme devient une fabrique de normes) soit l'interprétation (e.g. des métriques d'analyse, de certifications, etc.)

Les interpréteurs sont définit en tant que librairie et chargé pour instancier une norme à partir d'une prénorme (cf prochain tuto). Tout interpréteur n'est cependant pas compatible avec toute prénorme.

## Modalité

Le type de retour d'une norme est aussi encapsuler au sein d'un type que nous appelons «modalité». En effet il permet d'implémenter la logique modale. Ce type ne dépends pas de l'interpréteur mais est défini par l'utilisateur.

En logique aléthique, base de la sémantique de Kripke on considère 4 modalités du vrai: - La nécessité : une proposition est toujours vrai. Ceci correspond au type “Id” c'est a dire pas d'encapsulation : l'objet est retourné tel quel - La possibilité : une proposition est parfois vrai. Nous représentons cela avec le type “Maybe” qui correspond au retour d'un calcul qui peut échouer en une exception - La contingence : une proposition peut etre vrai ou pas. Nous representons ceci avec le type “Option” - L'impossibilité : une proposition n'est jamais vrai. Nous représentons cela conformément à l'approche agent (cf FIPA-ACL) en considérant la cause de cette impossiblité. Soit un échec du calcul, une impossibilité de le réaliser ou un refus de l'agent.

La logique aléthique, à travers la sémantique de Kripke, couvre la logique métier de nombreux domaines (cf https://fr.wikipedia.org/wiki/S%C3%A9mantique_de_Kripke). L'intégration de la récursivité nous permet de plus de modéliser la logique temporelle et de raisonner sur les executions possibles. De plus, on peut graphiquement assembler des «plans de secours» (valeur par défaut en cas de contingence, ou résolution d'exception en cas de possibilité) pour rendre une norme nécessairement vrai. Enfin, il est naturel d'utiliser d'autre modalité que celle décrite dans la logique aléthique si le besoin se fait.

/*

```
*/
import cloud.greentea.{*,given}
import cats.{~> as -->, *}
import cats.data._
```

```
/*
```

# Prenormes

@author Sylvain Ductor, 2022

## Déclaration

Une prénorme est un constructeur de normes. Une prénorme est construite en annotant une fonction Scala. Voici différente déclarations équivalentes en Scala :

```
*/
def increment_def(a : Int) : Int   = a + 1

val increment_verbose : Int => Int = (a : Int) => a + 1

val increment_compact : Int => Int = (_ : Int) + 1
/*
```

Une prenorme est déclaré en annotant une telle fonction Scala avec l'indication du type d'entrée et de retour de la norme :

```
*/
val addOne = ( (a : Int) => a + 1 ).~>[Int,Int]

val addOne_fromDef = increment_fun.~>[Int,Int]

val addOne_fromVerbose = increment_verbose.~>[Int,Int]

val addOne_fromCompact = increment_compact.~>[Int,Int]
/*
```

Le type de retour est automatiquement calculé. Il est possible de le spécifier explicitement, mais ceci est plus laborieux pour le programmeur. Cependant, s'il le spécifie il peut choisir de ne pas fournir l'annotation:

```
*/
val addOne_exp  : (Int ~> Int)[Int => Int]    = (_ : Int) + 1
/*
```

Le type d'une prenorme se décompose en deux parties :

- `(Int => Int)` est le type d'entrée/sortie attendu de la futur norme,
- `[Int => Int]` est le type de la fonction Scala supportant la norme.

## Assemblage

Des prenormes peuvent être assemblées en prénormes à l'aide des opérateurs *andThen*, *and*, *ior*, *xor*.

```
*/
val addTwo              = addOne andThen addOne

val addThree            = addOne andThen addOne andThen addOne

val addOne2Couple       = addOne and addOne

val addOne2Triple       = addOne and addOne and addOne

val addOneORAddThree    = addOne xor addThree

val addOneOrAddTwo      = addOne ior (addOne andThen addOne)
/*
```

- *andThen* va séquencer le calcul : "addThree" va incrémenter une fois, *puis* une deuxième fois *puis* une troisième fois.
- *and* parallelise le calcul : les types d'entrées et de sortie de "addOne2Couple" sont des couples d'entiers. Le calcul utilisera toujours les capacité de parallelisation fournit par l'interpreteur
- *xor* branche le calcul : il permet à l'utilisateur d'un contrat de choisir entre une branche de gauche et de droite.
- *ior* se comporte tantôt comme *xor* tantôt comme *and* selon l'entrée fournit

## Construction de normes

Pour obtenir une norme il faut spécifier un interpréteur et une modalité. Trois méthodes existent pour spécifier un interpréteur. Nous présentons ici la plus propice à l'utilisation en console et les autres au fil des tutos.

```
*/
given   : interpretor.default = interpretor.default()
/*
```

En Scala, le mot clé `given` permet d'intéragir avec le compilateur. Il s'agit en quelque sorte du pattern singleton au niveau du type. Ici on déclare au compilateur un objet par défaut de type `: interpretor.default`, instancié par l'appel au constructeur `interpretor.default()` (`interpretor.default` est un interpreteur qui se veux un juste compromis entre minimalisme et efficacité). On pourra optionnellement se référer à cet objet à l'aide de la variable  (chi), même si cela à peu de chance de nous être utile. Les méthodes que nous verrons ci-dessous necessite un interpréteur qu'elles vont requérir au compilateur. Si deux interpréteurs différents ont été chargé avec `given` le compilateur détectera une ambiguité et refusera de compiler. Ainsi l'approche par le `given` bien que la plus ergonomique nous impose une unique interpreteur durant toute la session.

Une fois l'interpréteur accessible, il faut exprimer la modalité désirée. Imaginons que celle-ci soit `Option`. Une norme est déclaré avec :

```
*/
val addOne_normeContingente  = addOne.|[Option]
/*
```

Félicitation vous venez de déclarer votre première norme!

Considérons la signature de la fonction scala, les types d'entrées/sorties déclaré par la prenorme, le type de l'interpréteur fournit et le type de la modalité demandé. Si la combinaison de ceux-ci n'est pas cohérente le compilateur refusera de compiler. De plus les types des contextes d'entrées et de sortie ( , ) sont automatiquement calculé par le compilateur comme tout ce qui à avoir avec la logique.

Nous fournissons quelques alias:

- Nécessité : `.yes` pour `.|[Id]`
- Possibilité : `.often` pour `.|[Validated[Throwable,?]]`
- Contingence : `.maybe` pour `.|[Option]`
- Impossibilité : `.no` pour `.|[None]`

## Execution

C'est à son execution que le canal d'agent concerné est déclaré. Plusieurs routines ont été définies pour faciliter l'appel d'une norme. Nous présentons ici `execWith` qui ne considère pas de canal d'agent. Le autres seront introduites dans le tuto sur les holons.

```
*/
val addOne_simple  = addOne.yes.execWith(1)

val addOne_couple  = addOne2Couple.yes.execWith((1,2))

val addOne_xor     = addOneORAddThree.yes.execWith(Left(1))// (1.left)

val addOne_ior     = addOneOrAddTwo.yes.execWith(Ior.Right(2)) //(2.right)

val addOne_iorB     = addOneOrAddTwo.yes.execWith(Ior.Both(1,2))

val addOne_andthen = addThree.yes.execWith(1)
/*
```

Une fois l'entrée chargée, les valeurs ci-dessus sont virtuellement des résultats encapsuler dans leur environnement d'exécution. `getResult()` va *effectivement* effectuer le calcul et en extraire le résultat.

```
*/
//@main def tuto_prenorme = println(addOne_simple.getResult())
/*
*/
```

```
/*
*/
import cloud.greentea.{*,given}
import cats.effect.std.Console
import cats.effect._
import cats.{~> as -->, arrow as ar, *}
/*
```

# API

@author Sylvain Ductor, 2022

## Déclaration

Une norme implémente une catégorie monoidale. Elle peut être étendue en une catégorie de Ductor afin d'intégrer des effets de bords. Pour cela elle considère: - Un environnement , accessible en lecture et en écriture et cela en fonction du couple d'agent considéré, - Des (pré)sources (ou axiomes), `+[A]` lisant cette environnement et retournant un type `A` - Des (pré)puits (ou preuves), `-[B]` écrivant un objet de type `B` dans l'environnement.

A titre illustratif, supposons que nous souhaitons une interface de communication avec le terminal. Nous choisissons d'utiliser pour cela l'objet `Console[IO]` de la librairie `cats.effect` sur lequel se fonde l'implémentation de `GreenTea` (voir https://typelevel.org/cats-effect/api/3.x/cats/effect/std/Console.html)

### Source

Nous commençons par la source, c'est à dire la flèche qui va lire et retourner l'entrée de l'utilisateur. Nous souhaitons une invite de commande puis que celle ci retourne la commande entrée.

```
*/
val console_reader = ((  : Console[IO]) =>
     .println("requesting :") >>  .readLine
).+[String]
/*
```

A l'instar d'une prénorme, nous étiquetons une fonction Scala avec `+[String]` pour indiquer qu'il s'agira d'une source de `String`. Cette fonction est de type `[Console[IO] => IO[String]]` et va être généralisé en une source de type `( ~> String)` lors de l'intégration d'un environnement qui fournira `Console[IO]` en lecture.  représente le type «zero».

Le symbole `>>` provient de l'encapsulateur de classe `Applicative` (voir https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html) contractuellement implémenté par `IO`. Il séquence les deux calculs en "oubliant"

le premier. Sémantiquement il est similaire à un `;` , seulement, s'il avait été un `;` l'effet de bord `println` n'aurait pas été effectué du fait du fonctionnement de `scala.cats.effect.IO`.

### Puit

Le puit quand à lui considère un objet à écrire et met à jour l'environnement.

```
*/
val console_printer = ((b : String,   : Console[IO]) =>
         .println(b).map(_ =>  )
).-[String]
/*
```

La fonction Scala est cette fois ci du type `[(String,Console[IO]) => IO[Console[IO]]]` et va être généralisé en un puit de type `(String ~> )`

### API

Une api est un couple (source,puit) portant sur le même environnement.

```
*/
val console_api = (console_reader,console_printer).api
/*
```

## Usage

Considérant une prenorme, `sayHello`, demandant le nom de son interlocuteur et lui disant bonjour.

```
*/
val sayHello = ( (name : String) => s"Hello $name" ).~>[String,String]
/*
```

Nous introduisons ici une deuxième méthode pour déclarer un interpretor: en faisant un objet l'étendre. Au sein de l'objet (et seulement en son sein), non seulement l'interpréteur est chargé, mais de plus certains types et routines sont accessible (par exemple Effect). Cette approche est la plus pratique hors d'une console sbt. A l'instar de python, maintenir la tabulation indique a Scala que l'on est toujours dans l'environnement objet.

```
*/
object tuto_api extends interpretor.default :

    println(sayHello.yes.execWith("World").getResult())
/*
```

Tant une présource qu'un prépuit sont des alias de prénormes et se comporte donc similairement.

```
*/
    val entry : Maybe[String] = //devrait être Often
        console_reader.yes.exec.getResult()

    console_printer.yes.execWith(s"Il a écrit : $entry").run()
/*
```

Une norme peut charger une api en lecture écriture ou les deux. Son type est alors modifié de façon cohérent. Elle devient une source si elle charge l'api en lecture, un puit si elle la charge en écriture ou un effet (`( ~> )`) si elle charge les deux. Ainsi, les différentes flèches de la catégorie de Ductor (norme, source, puit, effet) forment une cloture transitive. Sous forme d'effet la norme est dite «indépendante», elle est capable d'obtenir par elle même son type d'entrée et de sortie et peut alors s'assembler avec d'autre norme plus librement. En effet «Les monades sont des monoides dans la catégorie des endofunctеurs».

```
*/
    val helloEffect : Effect[Id] =
        sayHello.yes.effectOn(console_api)
    helloEffect.exec.getResult()

    sayHello.yes.readFrom(console_api).exec.getResult()

    sayHello.yes.writeTo(console_api).execWith("World").getResult()
/*
```

En l'absence de type d'entrée, `exec` est utilisé en place de `execWith` pour tranformé virtuellement la norme en son résultat.

Les environnement sbt console et scastie sont connus pour être limité au niveau de la lecture en console. Le programme ci dessus peut etre correctement exécuté avec `sbt run` qui va checher la définition annoté par @main et executer son contenu, c'est à dire charger l'objet en l'executant ligne par ligne

```
*/
@main
def main_tuto_api = tuto_api
/*
*/
```