Scala io

Kleisli

Un ami qui vous veut du bien

FABERNOVEL TECHNOLOGIES

tinyclues

LUNATECH

Teads.tv

ebiznext

criteo labs

vente-privee GROUP

Valraiso

lizeo group

mfg labs

Software Engineer

<3 challenges tech

Passionné Scala

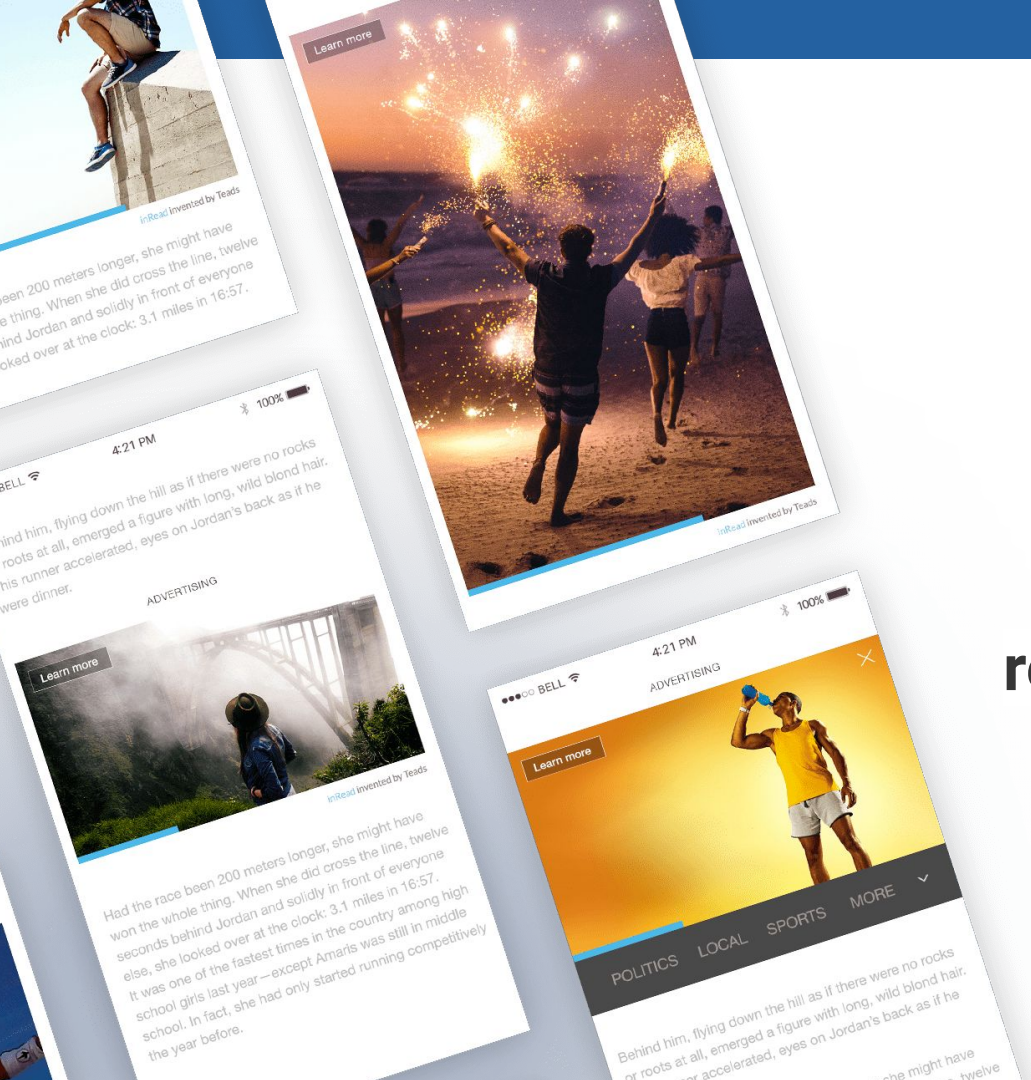
@TristanSoullz

Engineering manager

formateur Scala / Akka

aventurier de la FP

@xbucchiotty



Teads.tv

Join us to

reinvent video advertising

Chaque diffusion est choisie en temps réel

contenu éditorial

contenu publicitaire

appareil

50 <> 500 ms

500 000 rq/s

~30 rules per req

Évolution du domaine

1
```
type ExecutionResult    = Boolean
type Rule[T]            = T => ExecutionResult[T]
```

2
```
type ExecutionResult[T] = Either[String, T]
type Rule[T]            = T => ExecutionResult[T]
```

3
```
type ExecutionResult[T] = Either[String, T]
type Rule[T]            = T => Future[ExecutionResult[T]]
```

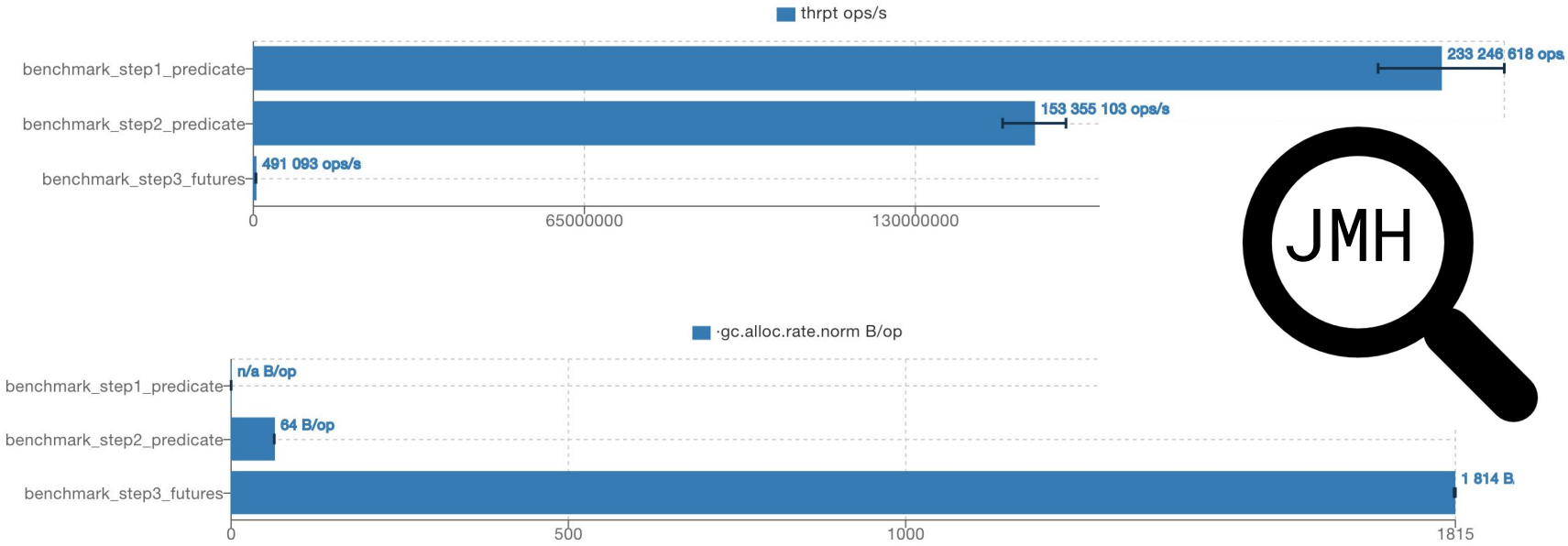
Problèmes

1. Composition difficile à cause des Futures (*Future.successful*)

```
def deviceRule(device: Device): Rule[Ad] = {
  ad => Future.successful(
    Either.cond(
      ad.device == device, ad,
      "Device does not match"
    )
  )
}
```

Problèmes

2. Performance

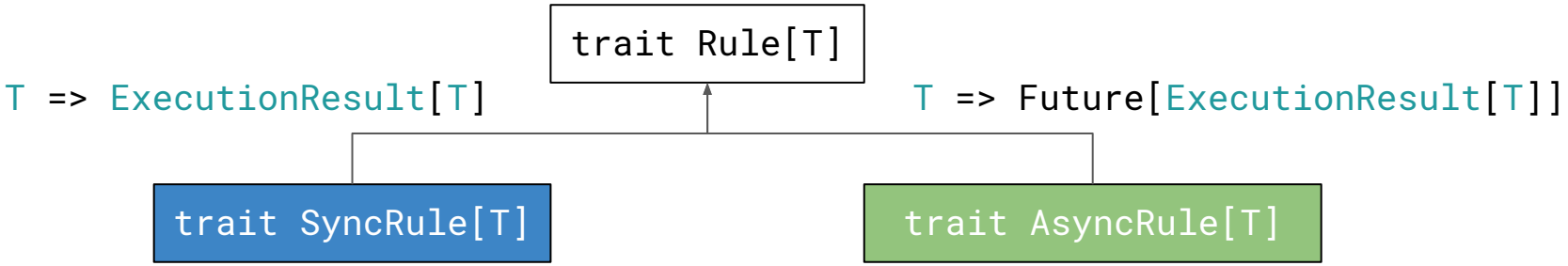


http://jmh.morethan.io/

LET'S

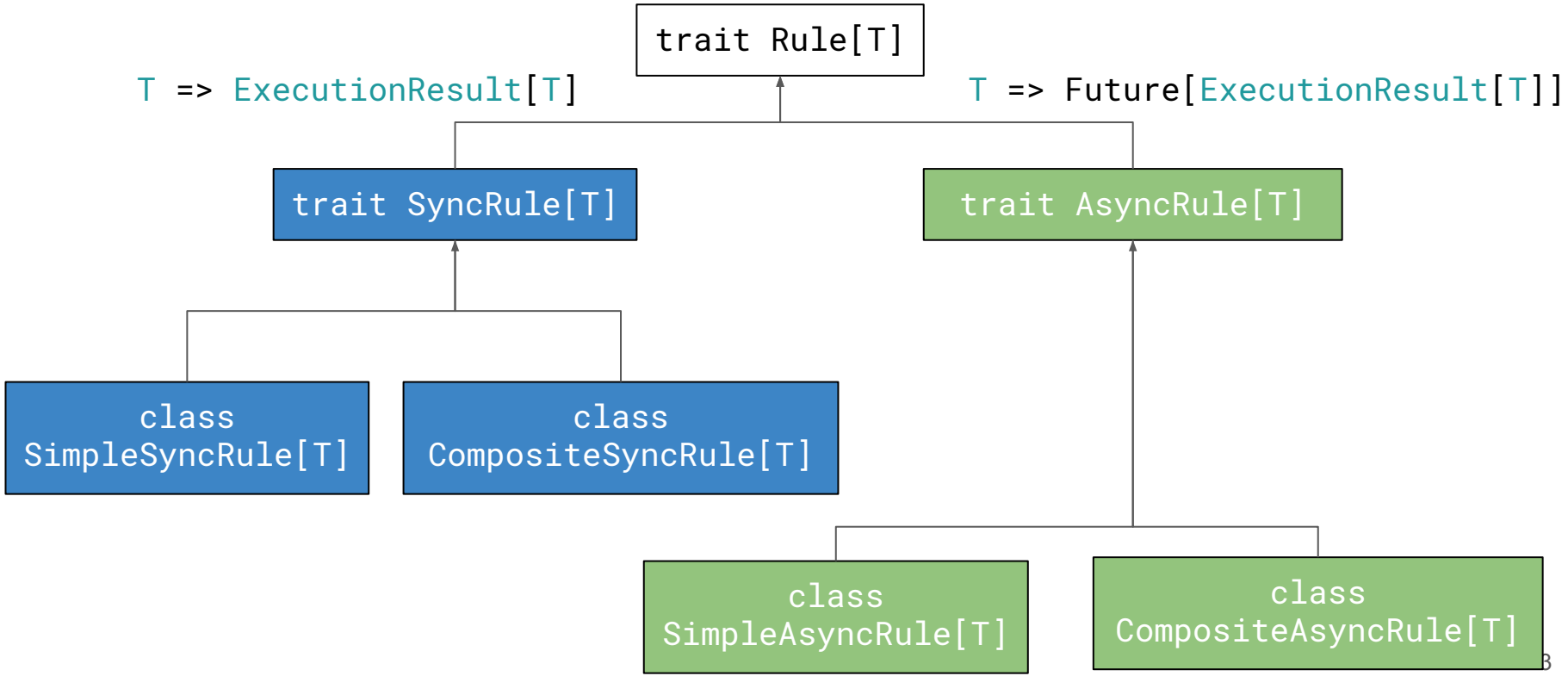
REFACTOR

Orienté objet

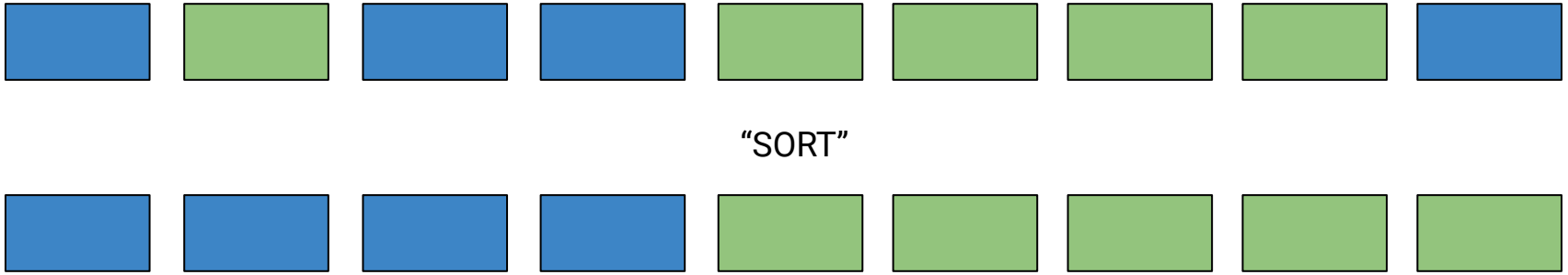


Orienté objet

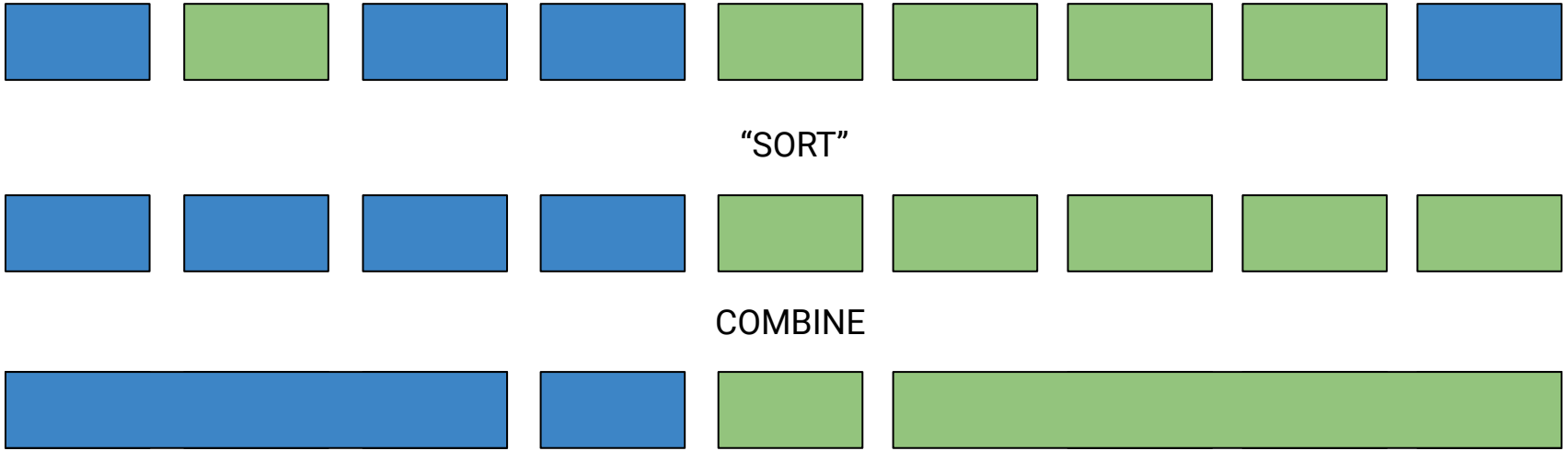
Orienté objet



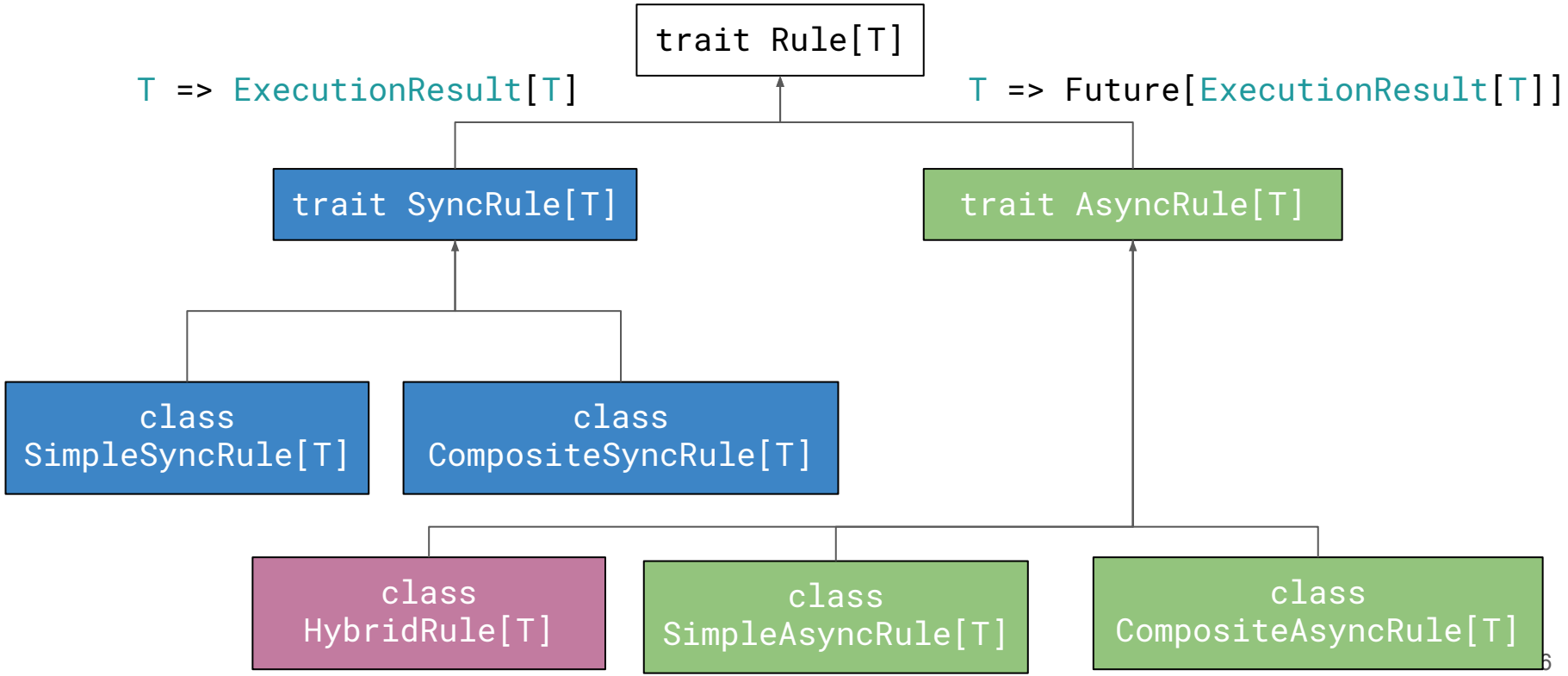
Orienté objet



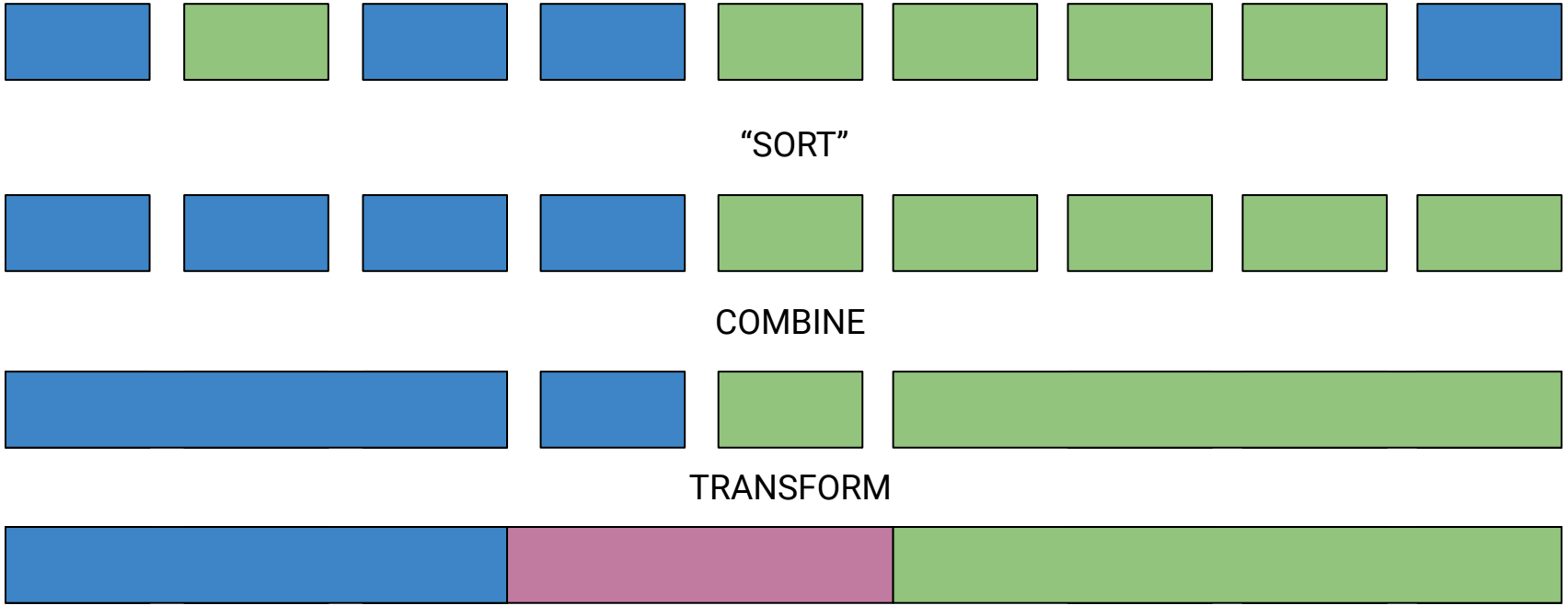
Orienté objet

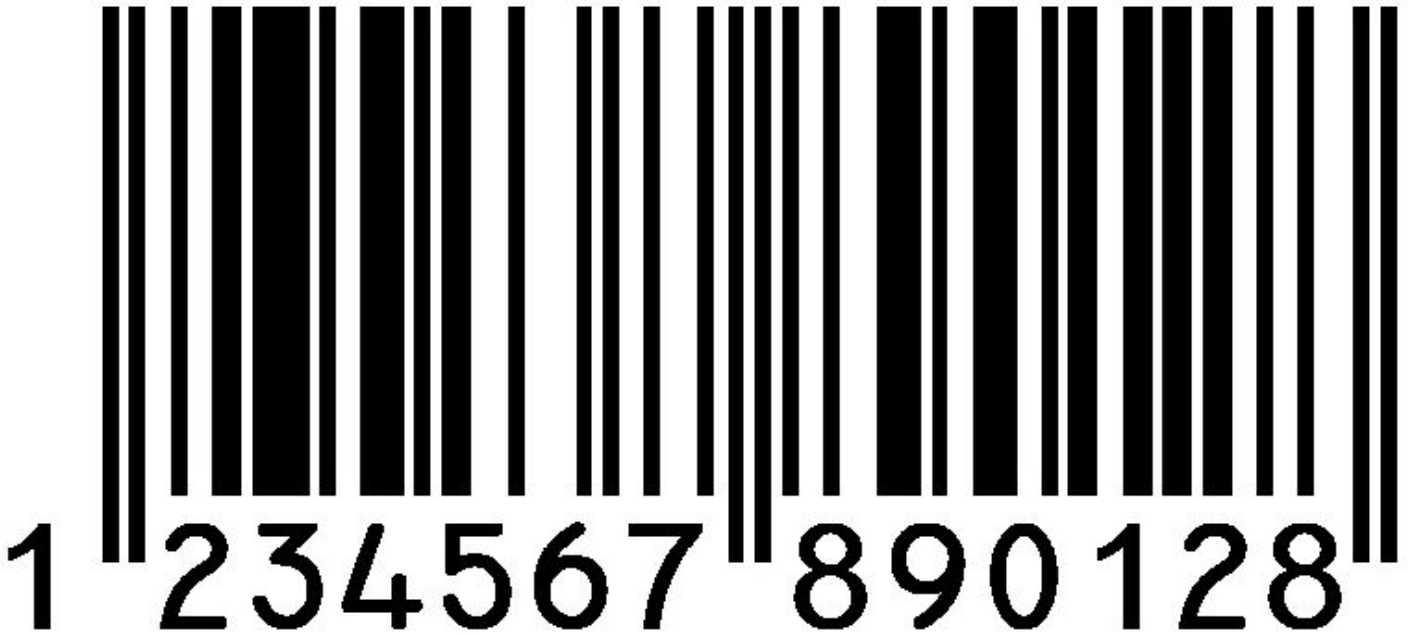


Orienté objet

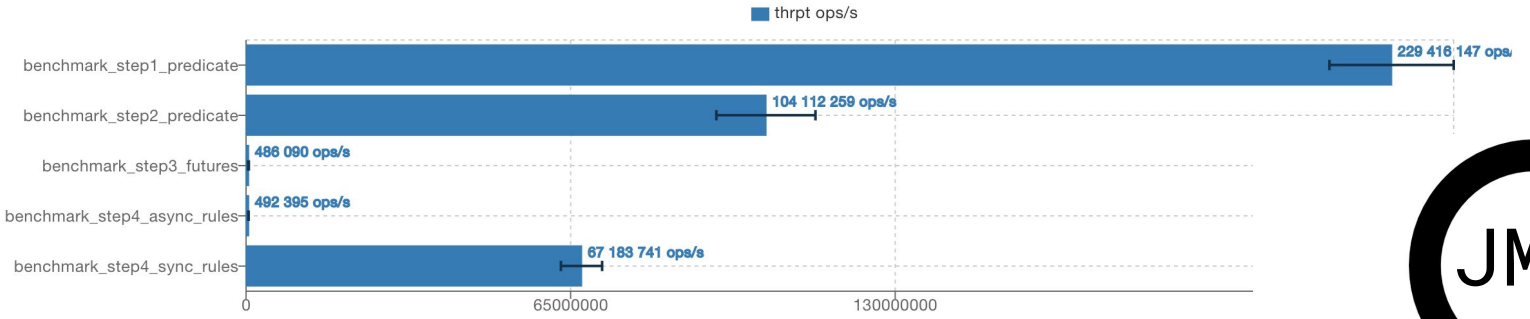


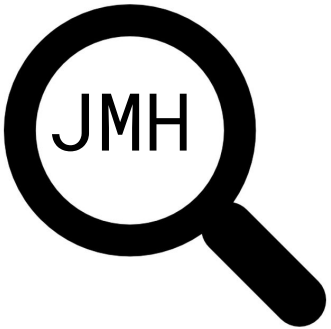
Orienté objet

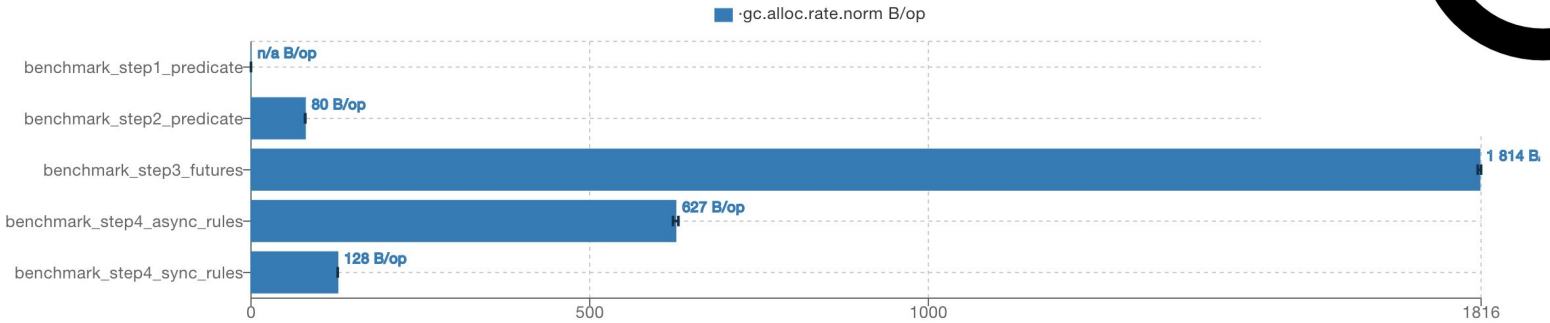



1 234567 890128

Performance







Composition

```
def combine[T](x: Rule[T], y: Rule[T]): Rule[T] = (x, y) match {

  case (l: SyncRule[T], r: SyncRule[T])      => SyncRule.combine(l, r)
  case (l: SyncRule[T], r: AsyncRule[T])     => AsyncRule.transform(l, r)
  case (l: AsyncRule[T], r: SyncRule[T])     => AsyncRule.transform(r, l)
  case (l: AsyncRule[T], r: AsyncRule[T])    => AsyncRule.combine(l, r)

}
```

Composition

```
def combine[T](x: Rule[T], y: Rule[T]): Rule[T] = (x, y) match {

  case (l: SyncRule[T], r: SyncRule[T])      => SyncRule.combine(l, r)
  case (l: SyncRule[T], r: AsyncRule[T])     => AsyncRule.transform(l, r)
  case (l: AsyncRule[T], r: SyncRule[T])     => AsyncRule.transform(r, l)
  case (l: AsyncRule[T], r: AsyncRule[T])    => AsyncRule.combine(l, r)

}
```

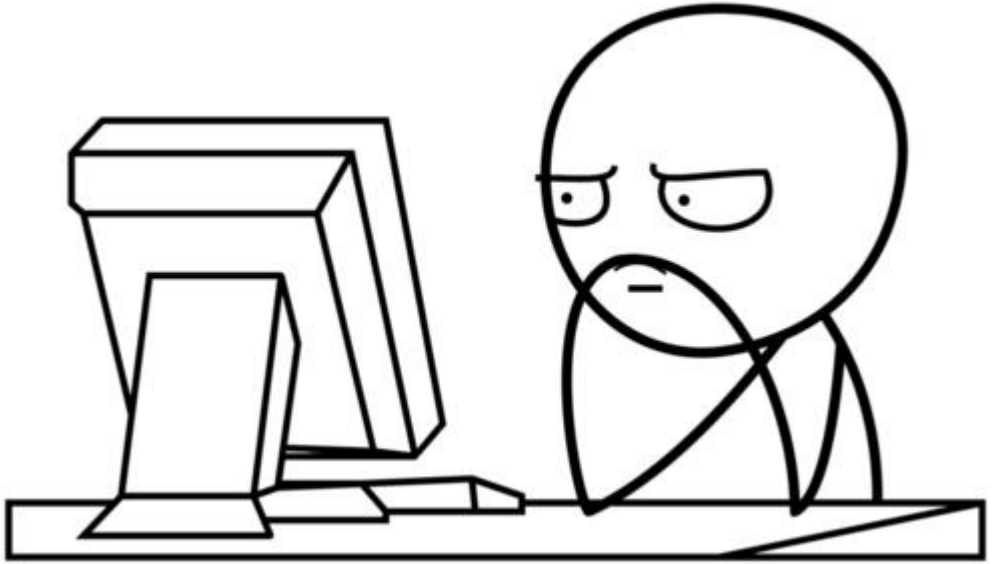
TYPE CHECKING

Composition

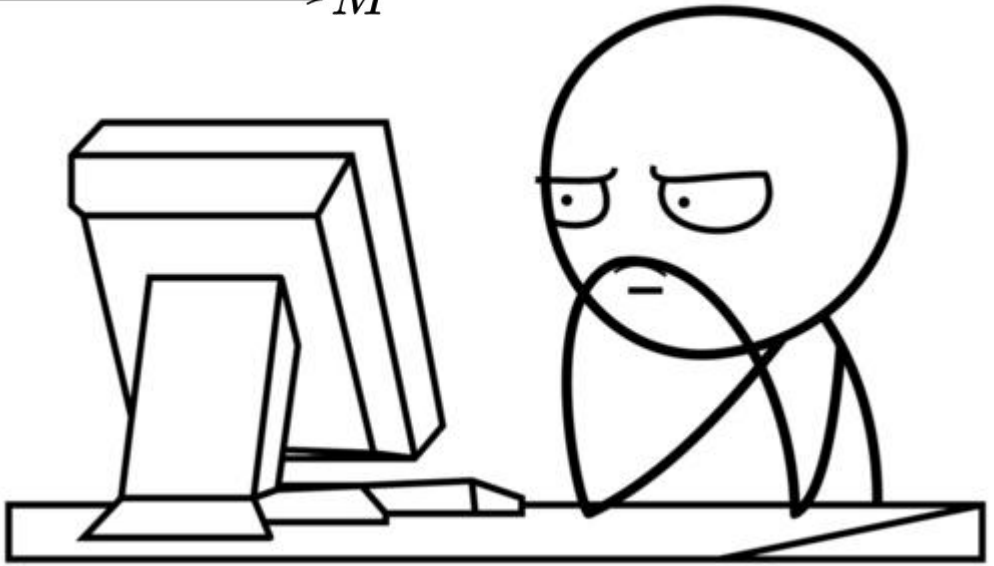
```
def combine[T](x: Rule[T], y: Rule[T]): Rule[T] = (x, y) match {

  case (l: SyncRule[T], r: SyncRule[T])      => SyncRule.combine(l, r)
  case (l: SyncRule[T], r: AsyncRule[T])     => AsyncRule.transform(l, r)
  case (l: AsyncRule[T], r: SyncRule[T])     => AsyncRule.transform(r, l)
  case (l: AsyncRule[T], r: AsyncRule[T])    => AsyncRule.combine(l, r)

}
```

TYPE CHECKING

HIÉRARCHIE COMPLEXE

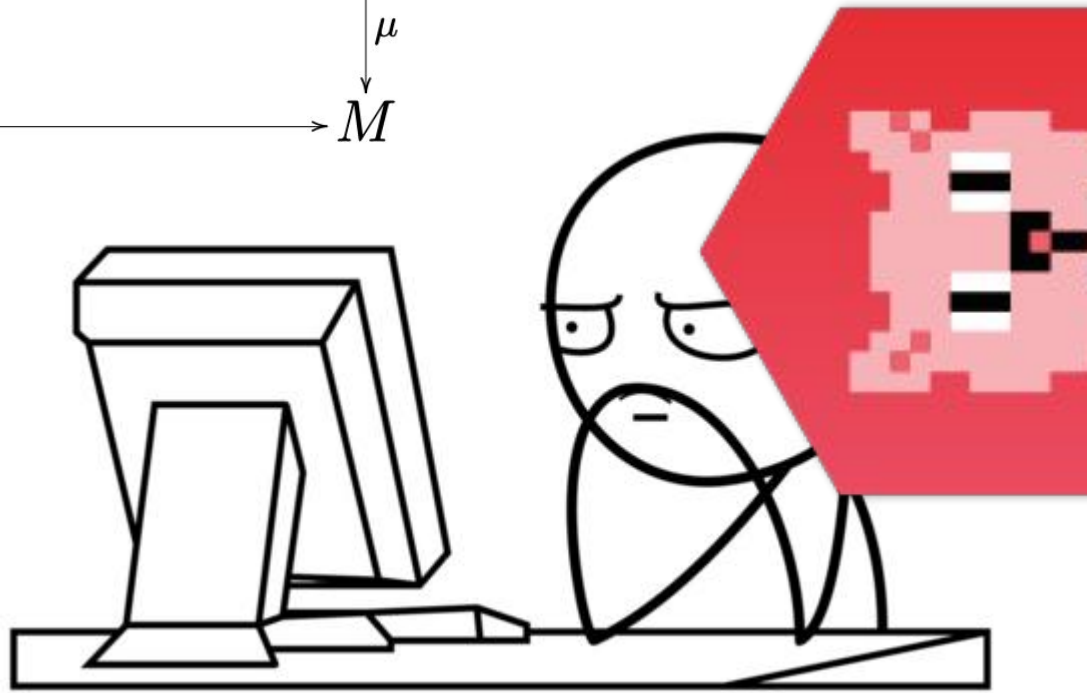


$$\begin{array}{ccccc} (M \otimes M) \otimes M & \xrightarrow{\alpha} & M \otimes (M \otimes M) & \xrightarrow{1 \otimes \mu} & M \otimes M \\ \downarrow{\scriptstyle \mu \otimes 1} & & & & \downarrow{\scriptstyle \mu} \\ M \otimes M & & \xrightarrow{\quad\mu\quad} & & M \end{array}$$



$$\begin{array}{ccccc} (M \otimes M) \otimes M & \xrightarrow{\alpha} & M \otimes (M \otimes M) & \xrightarrow{1 \otimes \mu} & M \otimes M \\ \downarrow{\scriptstyle \mu \otimes 1} & & & & \downarrow{\scriptstyle \mu} \\ M \otimes M & & \xrightarrow{\quad\mu\quad} & & M \end{array}$$

Monoid



Composition

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}
```

https://typelevel.org/cats/typeclasses/semigroup.html

Monoid

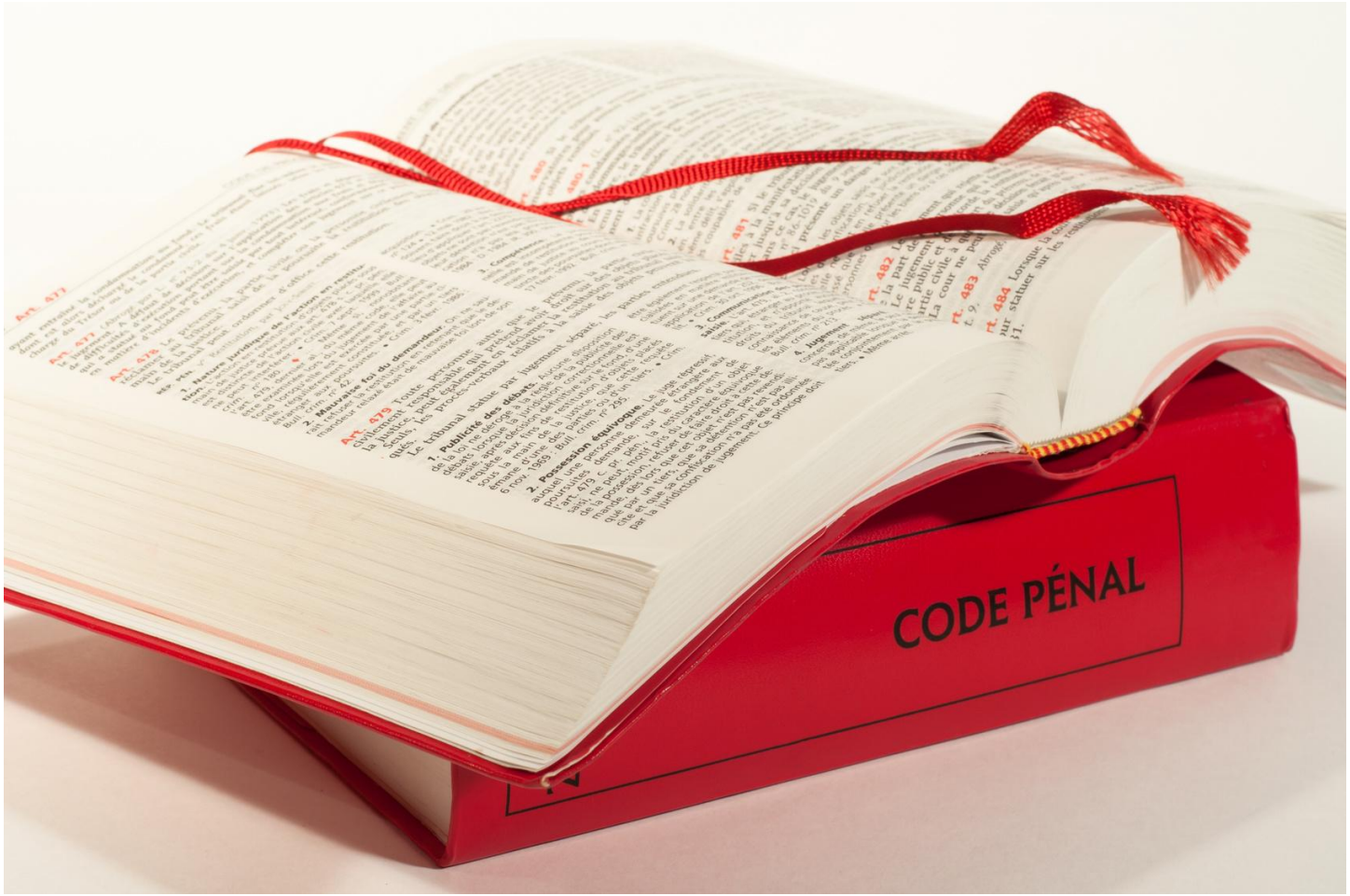
```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

https://typelevel.org/cats/typeclasses/monoid.html

Monoid on kind

```
trait Monoid[A] {
  def combine(x: A, y: A): A
  def empty: A
}

trait MonoidK[F[_]] {
  def combineK[A](x: F[A], y: F[A]): F[A]
  def empty[A]: F[A]
}
```

https://typelevel.org/cats/typeclasses/monoidk.html


CODE PÉNAL

Évolution du domaine

```
type SyncRule[T]        = T => ExecutionResult[T]

type AsyncRule[T]       = T => Future[ExecutionResult[T]]
```

Évolution du domaine

```
type SyncRule[T]        = T =>        ExecutionResult[T]

type AsyncRule[T]       = T => Future[ExecutionResult[T]]
```

Évolution du domaine

```
type SyncRule[T]        = T =>     Id[ExecutionResult[T]]

type AsyncRule[T]       = T => Future[ExecutionResult[T]]


type Rule[Effect[_],T] = T => Effect[ExecutionResult[T]]
```

Kleisli

```
final case class Kleisli[F[_], A, B](
   run: A => F[B]
)
```

https://typelevel.org/cats/datatypes/kleisli.html

Function

```
trait Function[A,B] {
  def apply(x: A): B
}
```

https://typelevel.org/cats/datatypes/functionk.html

Function on kind

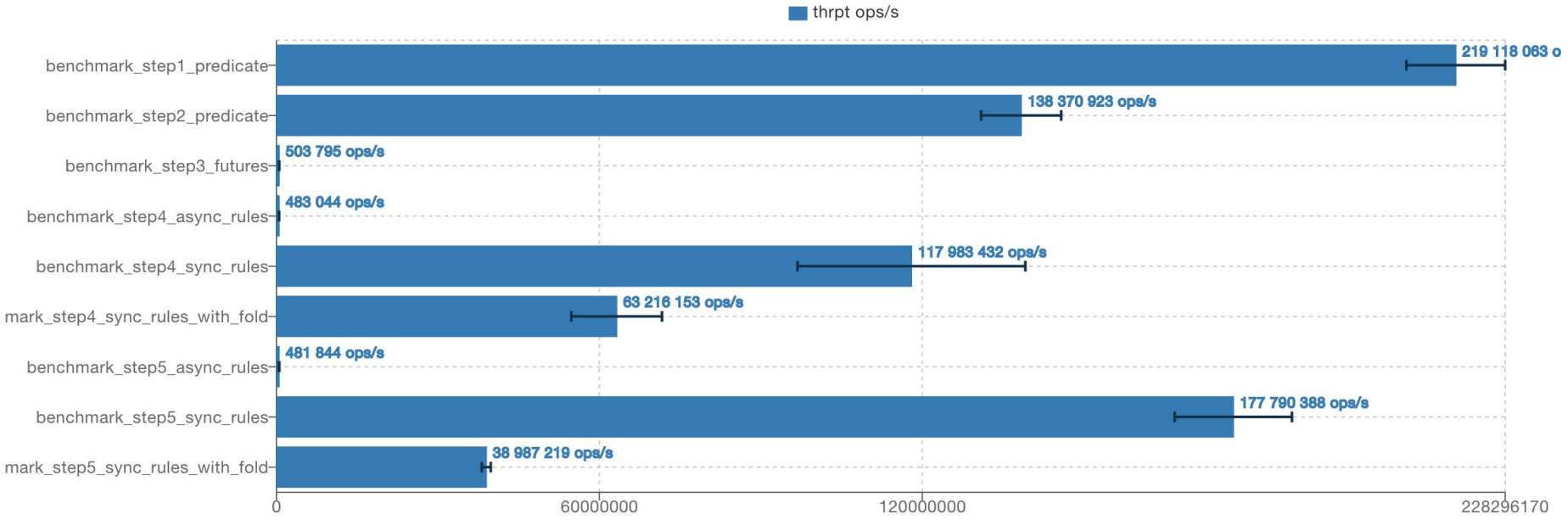
```
trait Function[A,B] {
  def apply(x: A): B
}

trait FunctionK[F[_],G[_]] {
  def apply[A](x: F[A]): G[A]
}
```

https://typelevel.org/cats/datatypes/functionk.html

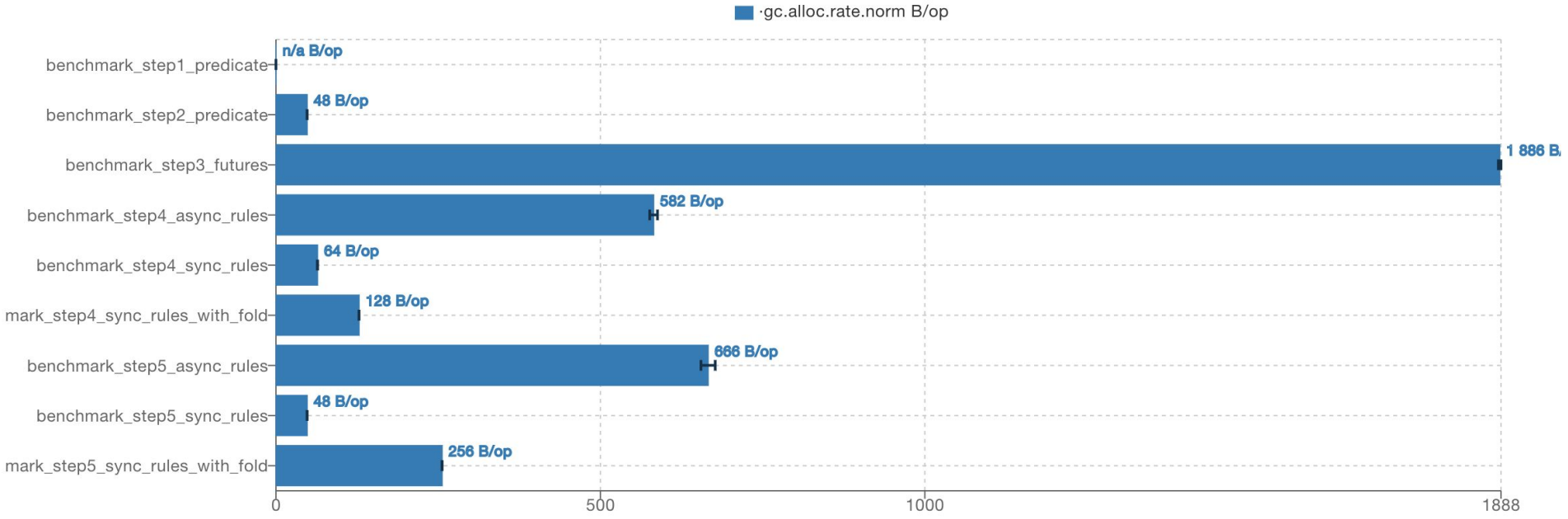
Composition

```
type SyncRule[T]  = Rule[Id, T]

type AsyncRule[T] = Rule[Future, T]

type TryRule[T]   = Rule[Try, T]
```

Performance



Performance



Scala io

CONCLUSION

Abstraction

"The purpose of **abstraction** is **not** to be **vague**, but to create a new semantic level in which one can be **absolutely precise**."

Edsger W. Dijkstra

Ouverture

```
type Rule[Effect[_],T]                = T => Effect[ExecutionResult[T]]


type Rule[Effect[_],Result[_],T]  = T => Effect[Result[T]]
```

Scala io

QUESTIONS

https://github.com/teads/scalaio-kleisli

FABERNOVEL TECHNOLOGIES

tinyclues

vente-privee GROUP

LUNATECH

Valraiso

Teads

lizeo group

ebiznext

mfg labs

criteo labs