Liberate Your Monads

Introduction To Free Monad

Jiří Jakeš

December 21, 2016

Scala Taiwan Meetup

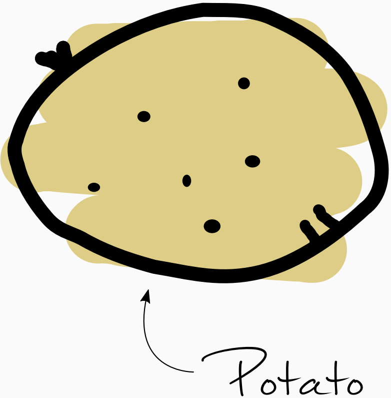
Agenda

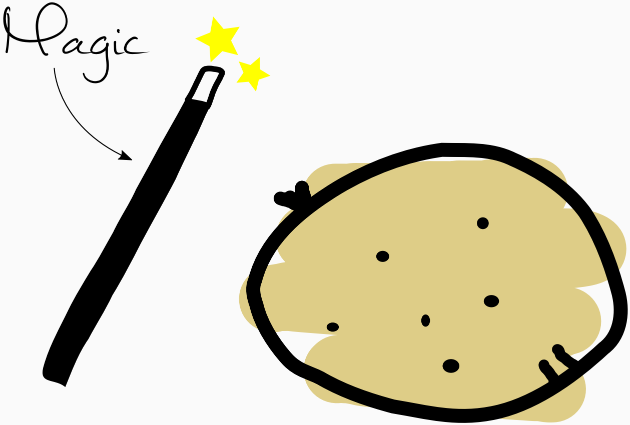
Monad Reminder

Today's Reality

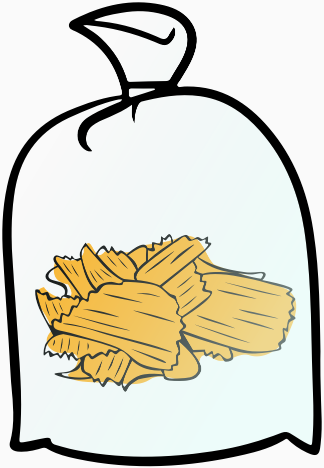
Free Monad

Monad Reminder

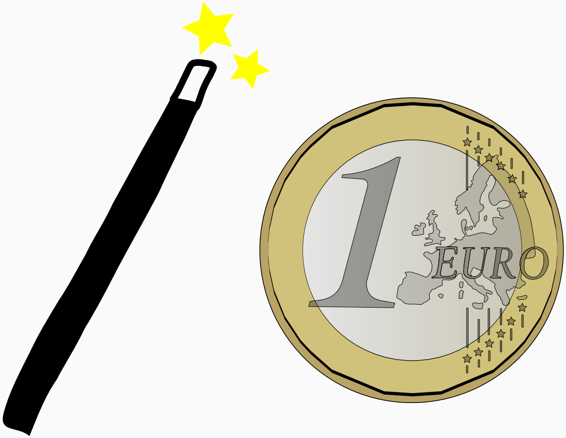

Potato

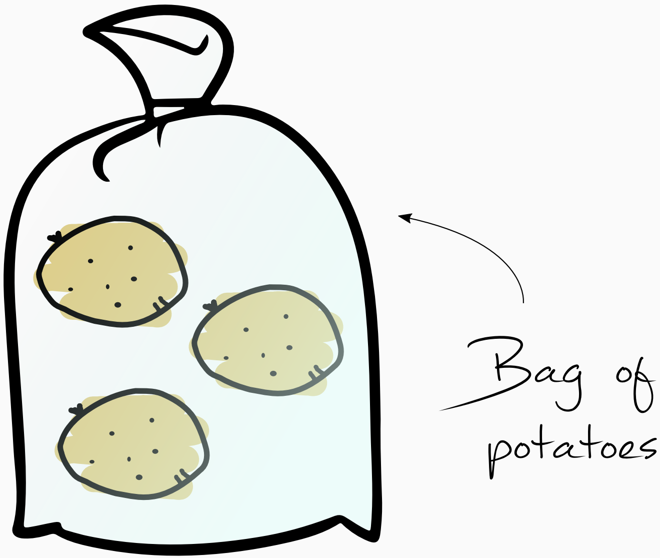

Magic

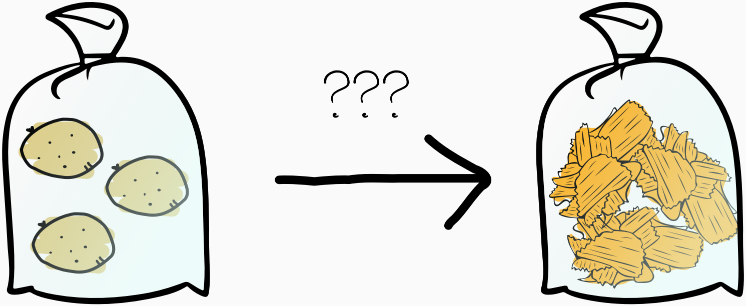
Bag of
potato chips



Coin

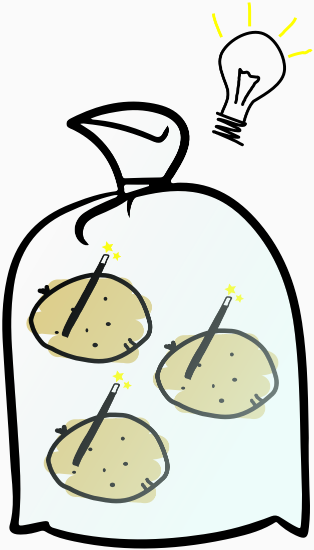

1
EURO


Bag of
potatoes



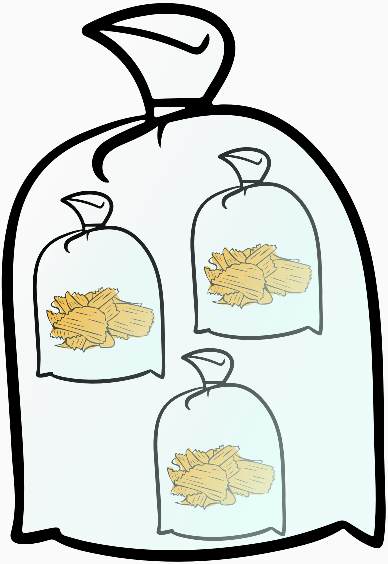
Having a bag of potatoes, how can we transform it into bag of chips?

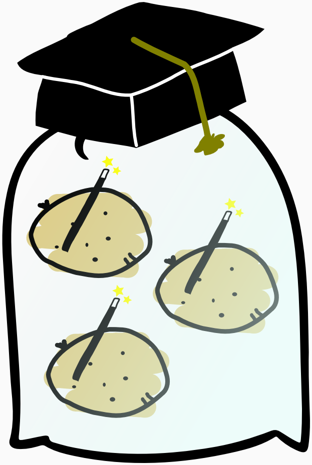
We cannot apply magic to bag of potatoes, only to the potatoes themselves.



Luckily the bag is a smart bag and can help us to apply magic to its content one by one.

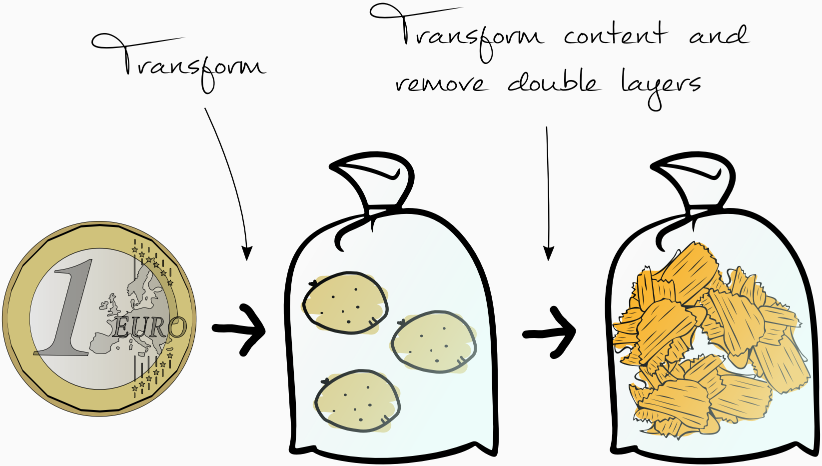
However, that gives us bag of bags of chips.



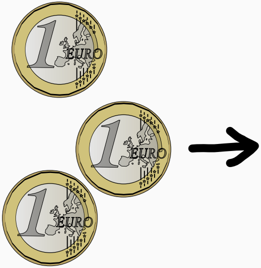


Fortunately, the bag is really really smart and it can not only apply magic but also remove double layer of bags.

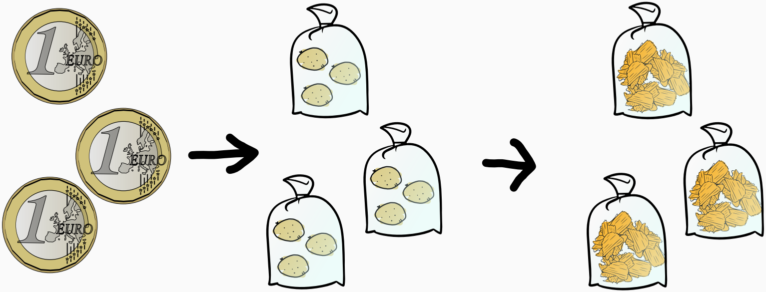
If we ask it to do so.



We can now transform a coin into a bag of chips.



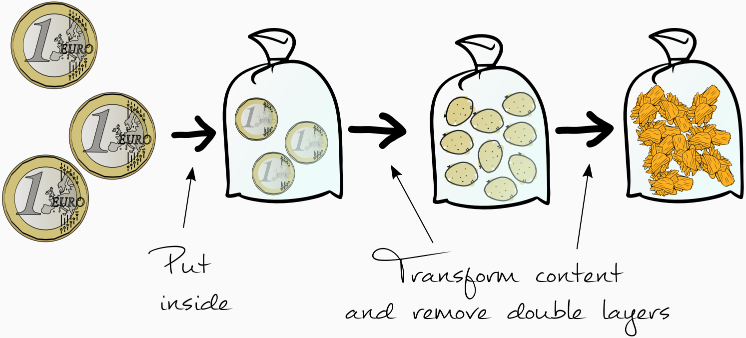
What if we have more coins?

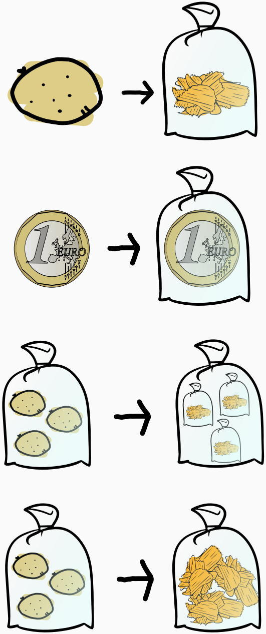


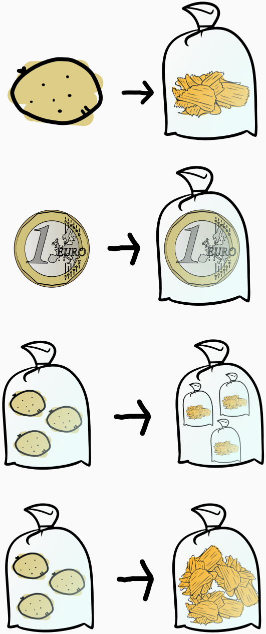
They are transformed into more bags of chips. That is not what we want. How can we have only one bag at the end?

We need coins inside the bag. Smart bag can help us.

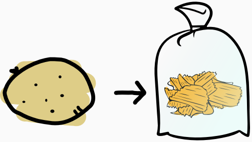
We need coins inside the bag. Smart bag can help us.




1 EURO
1 EURO

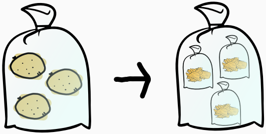


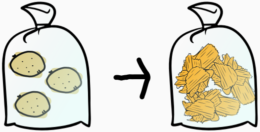
```
makeChips: Potato => Bag[Chips]
```

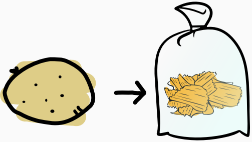


```
makeChips: Potato => Bag[Chips]
```


```
pure: Coin => Bag[Coin]
```

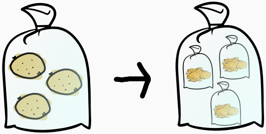




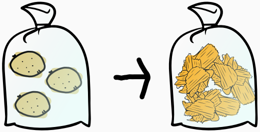


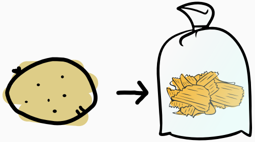
```
makeChips: Potato => Bag[Chips]
```


```
pure: Coin => Bag[Coin]
```



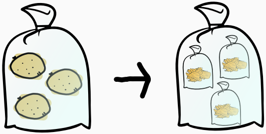
```
val potatoes: Bag[Potato] = ...
val bbc: Bag[Bag[Chips]] =
            potatoes.map(makeChips)
```



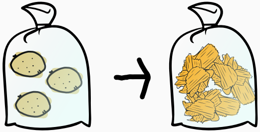


```
makeChips: Potato => Bag[Chips]
```


```
pure: Coin => Bag[Coin]
```



```
val potatoes: Bag[Potato] = ...
val bbc: Bag[Bag[Chips]] =
             potatoes.map(makeChips)
```



```
val potatoes: Bag[Potato] = ...
val bc: Bag[Chips] =
          potatoes.flatMap(makeChips)
```

```scala
trait Bag[A] {
  def map[B](f: A => B): Bag[B]
  def flatMap[B](f: A => Bag[B]): Bag[B]
}

object Bag {
  def pure[A](a: A): Bag[A] = ...
}
```

Exactly one type parameter

```scala
trait Bag[A] {
  def map[B](f: A => B): Bag[B]
  def flatMap[B](f: A => Bag[B]): Bag[B]
}

object Bag {
  def pure[A](a: A): Bag[A] = ...
}
```

Exactly one type parameter

map

```scala
trait Bag[A] {
  def map[B](f: A => B): Bag[B]
  def flatMap[B](f: A => Bag[B]): Bag[B]
}

object Bag {
  def pure[A](a: A): Bag[A] = ...
}
```

Exactly one type parameter

map

```scala
trait Bag[A] {
  def map[B](f: A => B): Bag[B]
  def flatMap[B](f: A => Bag[B]): Bag[B]
}

object Bag {
  def pure[A](a: A): Bag[A] = ...
}
```

flatMap

Exactly one type parameter

map

```scala
trait Bag[A] {
  def map[B](f: A => B): Bag[B]
  def flatMap[B](f: A => Bag[B]): Bag[B]
}

object Bag {
  def pure[A](a: A): Bag[A] = ...
}
```

flatMap

pure

Exactly one type parameter

map

```scala
trait Bag[A] {
  def map[B](f: A => B): Bag[B]
  def flatMap[B](f: A => Bag[B]): Bag[B]
}
```

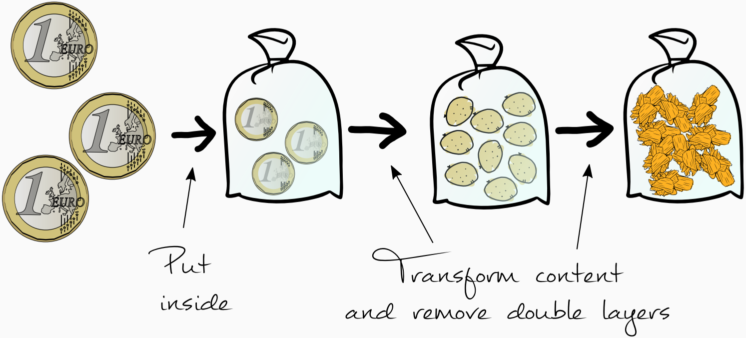
flatMap

```scala
object Bag {
  def pure[A](a: A): Bag[A] = ...
}
```

pure

⇒ Monad!

Side note: `map` is not necessary as it can be expressed using `flatMap` and `pure`.




```
Bag.pure(coins).flatMap(buyPotatoes).flatMap(makeChips)
```

Today’s Reality

- Accessing external services
- (Distributed) databases
- Manipulating files
- Using microservices
- …

Let us now consider distributed key-value store.

```scala
def doBusinessLogic(input: String): Int = {
  ...
  val data =
    try {
      transform(db.getString(key))
    } catch {
      case _: NonFatal => defaultValue
    }
  ...
  val result = compute(data, input)
  log.debug("Result: {}", result)
  db.store(result)
  result
}
```

How such code can be tested?

- Mocking/stubbing database
    - May lead to complex dependency injection.
    - Differencies between mock and real system.
- Using real database
    - Do we want to test how database works or our business logic?
    - Annoying setup, setdown, regular unit tests, more developers etc.
    - Property-based testing (Scalacheck) impossible.
    - Parallel testing may not be possible.
    - In some cases (cloud databases) impractical and expensive.

How such code can be changed?

- In-place changes
    - Maintenance nightmare.
- Abstract class/interface
    - Synchronous/asynchronous?
    - May lead to complex dependency injection.
    - May be very complicated and large interface.
    - Inheritence often brings more troubles than it solves.

Main problem is very low or no separation of concerns.

Main problem is very low or no separation of concerns.

It could be very helpful if we could only focus on implementing business logic and would not have to bother with its execution.

Let us try to find a way how we can access key-value store given these requirements:

- Absolute separation of business logic and execution.
- Same code for production and testing, only execution differs.
- Easy change of execution logic (different database system, synchronous/asynchronous, …)

Free Monad

Our key-value store requirements:

- Keys are strings, values are strings.
- Operations: retrieve, store, delete key, delete all.

We will design our *embdedded domain specific language* (DSL) which we use for programming. This DSL is translated into *abstract syntax tree* (AST). At the end, AST is interpreted/ executed.

We will use Free Monad as provided by Typelevel Cats.

Representation of primitives (AST):

```scala
sealed trait Ast[A]    // A is type of result
case class Put(k: String, v: String) extends Ast[Unit]
case class Get(k: String) extends Ast[String]
case class Delete(k: String) extends Ast[Unit]
case object Truncate extends Ast[Unit]
```

Type of our domain specific language:

```scala
import cats.Free

type Dsl[A] = Free[Ast, A]
```

Type of DSL instruction is free monad using set of primitives `Ast`, its result is of type `A`.

Instructions of our DSL:

```scala
import cats.Free.liftF

def put(k: String, v: String): Dsl[Unit] = liftF(Put(k, v))
def get(k: String): Dsl[String] = liftF(Get(k))
def delete(k: String): Dsl[Unit] = liftF(Delete(k))
def truncate: Dsl[Unit] = liftF(Truncate)
```

`liftF` creates a free monad from a given primitive. These instructions are now monads.

AST primitives are representation of DSL in memory (as value).

When programming, we use DSL instructions, free monad then translates them into AST primivites.

DSL instructions are monads…

DSL instructions are monads… they have `map` and `flatMap`.

DSL instructions are monads… they have `map` and `flatMap`. We can use them to derive new instructions, for example an instruction to obtain integer number:

DSL instructions are monads… they have `map` and `flatMap`. We can use them to derive new instructions, for example an instruction to obtain integer number:

```
def int(k: String): Dsl[Int] = get(k).map(_.toInt)
```

Or an instruction to read country from two different keys:

```scala
case class Country(code: String, name: String, people: Int)

def country(code: String): Dsl[Country] = for {
  name   <- get(s"$code.name")
  people <- int(s"$code.people")
} yield Country(code, name, people)
```

Or an instruction to copy country under different code:

```
def copyCountry(from: String, to: String): Dsl[Unit] = for {
  c <- country(from)
  _ <- put(s"$to.name", c.name)
  _ <- put(s"$to.people", c.people.toString)
} yield ()
```

We have instructions of our DSL, time to use them to write a program.

Program that uses DSL instructions we defined:

```
val program: Dsl[Boolean] = for {
  _    <- put("tw.name", "Taiwan")
  _    <- put("tw.people", "23519518")
  orig <- country("tw")
  _    <- copyCountry("tw", "roc")
  copy <- country("roc")
} yield orig.people == copy.people
```

Nothing is executed, program is just a description of things that will be done when executed.

However, we did not specify how this program should be executed. To perform some real action, we need to write an *interpreter* of AST.

In Cats' Free, the most straightforward way to write an interpreter is a *natural transformation*.

In Cats' Free, the most straightforward way to write an interpreter is a *natural transformation*.

Function

`Function1[A, B]` or `A => B`

E.g.: `String => Int` or `List[Double] => Boolean`

In Cats' Free, the most straightforward way to write an interpreter is a *natural transformation*.

Function

`Function1[A, B]` or `A => B`

E.g.: `String => Int` or `List[Double] => Boolean`

Natural transformation

`FunctionK[F[_], G[_]]` or `F ~> G`

E.g.: `List ~> Option` or `Set ~> Future`

Function

```scala
// A => B
trait Function1[A, B] {
  def apply(a: A): B
}
```

Natural transformation

```scala
// F ~> G
trait FunctionK[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}
```

Example of natural transformation `List ~> Option`:

```scala
val listToOption = new (List ~> Option) {
  def apply[A](fa: List[A]): Option[A] = {
    if (fa.isEmpty) None else Some(fa.head)
  }
}
```

Example of natural transformation `List ~> Option`:

```
val listToOption = new (List ~> Option) {
  def apply[A](fa: List[A]): Option[A] = {
    if (fa.isEmpty) None else Some(fa.head)
  }
}
```

```
listToOption(List(1, 2, 3)) = Some(1)
listToOption(List("a", "b")) = Some("a")
listToOption(List.empty[DateTime]) = None
```

Interpreter of our AST is natural transformation `Ast ~> M` where `M` must be monad.

Interpreter of our AST is natural transformation `Ast ~> M` where `M` must be monad.

We cannot write `Ast ~> Boolean` as `Boolean` is not monad. For this purpose, Cats provides `Id` monad which represents simple type but at the same time is a monad.

Shortly about `Id` (identity) monad:

```scala
import cats.Id

// Id is defined in Cats
// Id is monad
type Id[A] = A

val a: Int = 1
val b: Id[Int] = a
val c: Int = b
```

Interpreter to **Id** using in-memory map:

```scala
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {






}
```

Interpreter to `Id` using in-memory map:

```
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {
  val map = mutable.Map.empty[String, String]







}
```

Interpreter to **Id** using in-memory map:

```
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {
  val map = mutable.Map.empty[String, String]

  override def apply[A](fa: Ast[A]): A =




}
```

Interpreter to **Id** using in-memory map:

```scala
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {
  val map = mutable.Map.empty[String, String]

  override def apply[A](fa: Ast[A]): A = fa match {
    case Put(k, v) =>
    case Get(k)    =>
    case Delete(k) =>
    case Truncate  =>
  }
}
```

Interpreter to **Id** using in-memory map:

```
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {
  val map = mutable.Map.empty[String, String]

  override def apply[A](fa: Ast[A]): A = fa match {
    case Put(k, v) => map.update(k, v)
    case Get(k)    =>
    case Delete(k) =>
    case Truncate  =>
  }
}
```

Interpreter to **Id** using in-memory map:

```scala
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {
  val map = mutable.Map.empty[String, String]

  override def apply[A](fa: Ast[A]): A = fa match {
    case Put(k, v) => map.update(k, v)
    case Get(k)    => map(k)
    case Delete(k) =>
    case Truncate  =>
  }
}
```

Interpreter to `Id` using in-memory map:

```
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {
  val map = mutable.Map.empty[String, String]

  override def apply[A](fa: Ast[A]): A = fa match {
    case Put(k, v) => map.update(k, v)
    case Get(k)    => map(k)
    case Delete(k) => map.remove(k); ()
    case Truncate  =>
  }
}
```

Interpreter to `Id` using in-memory map:

```
import cats.{~>, Id}

def idInterp = new (Ast ~> Id) {
  val map = mutable.Map.empty[String, String]

  override def apply[A](fa: Ast[A]): A = fa match {
    case Put(k, v) => map.update(k, v)
    case Get(k)    => map(k)
    case Delete(k) => map.remove(k); ()
    case Truncate  => map.clear()
  }
}
```

And now we can finally interpret our program:

```scala
// type Dsl[A] = Free[Ast, A]
// val program: Free[Ast, Boolean] = ...
val program: Dsl[Boolean] = ...
val idInterp: Ast ~> Id = ...

val result: Boolean = program.foldMap(idInterp)
// result = true
```

We can write as many interpreters as we want:

- `Ast ~> Id` for unit tests
- `Ast ~> Future` for integration tests, local database
- `Ast ~> Future` connecting to production database
- `Ast ~> Either[String, ?]` doing validations
- `Ast ~> State[List[String], ?]` to track logs
- `Ast ~> LowLevelAst` translating to another Ast

Program remains the same, only the way it is executed changes.

Mixing more DSL's

- Does not work by itself (different types)
- Requires some more coding
- Under heavy research

- Injecting in Cats
- Freek

Injecting in Cats (incomplete, simplified):

```scala
sealed trait DbAst[A]
sealed trait LogAst[A]

type BothAst[A] = Coproduct[DbAst, LogAst, A]

class DbDsl[F[_]](implicit I: Inject[DbAst, F] {
  def get(k: String): Free[F, String] = Free.inject(Get(k))
}

def prg(implicit I: DbDsl[BothAst]): Free[BothAst, Unit] = {

  import I._

  for {
    v <- get("key")
    _ <- logDebug("Retrieved key")
    ...
  }
}
```

Composing interpreters:

```
val interp: BothAst ~> Id = DbInterpreter or LogIntepreter
```

Composing in Freek:

```scala
sealed trait DbAst[A]
sealed trait LogAst[A]

type Both = DbAst :|: LogAst :|: NilDSL
val Both = DSL.Make[Both]

val program: Free[Both.Cop, String] = {
  for {
    _ <- logDebug("Retrieving key").freek[Both]
    v <- get("key").freek[Both]
    ...
  } yield ...
}
```

Composing interpreters:

```
val interp = DbInterpreter :&: LogInterpreter

val result = program.interpret(interp)
```

Pros

- Separation of concerns
- Reusability
- Composability
- Stack-safe

Cons

- Performance
- Requires some studying
- Sometimes boilerplatish

Thank you for your attention


台灣製造・品質可靠
歐德
Purely Functional
iF
Order® 沙發
萬商大樓
行人優先
Order
沙發
床墊
8