Recursion schemes in Scala

and also fixed point data types

Yo

Today we will speak about freaky functional stuff, enjoy :)

Arthur Kushka // @Arhelmus

Functional programming

a method for structuring programs mainly as sequences of possibly nested function procedure calls.

```
((x: Array[Byte]) => x)
  .andThen(encodeBase64)
  .andThen(name => s"Hello $name")
  .andThen(println)
```

Recursive schemas

everywhere!

- **Lists and trees**
- **Filesystems**
- **Databases**

```
list match {
  // Cons(1, Cons(2, Cons(3, Nil)))
  case 1 :: 2 :: 3 :: Nil =>
  case Nil =>
}
```

01

Example

Math equasion

evaluation

- **Sum**
- **Multiply**
- **Divide**
- **Square**

Lets define our DSL

```scala
sealed trait Exp
case class IntValue(value: Int) extends Exp
case class DecValue(value: Double) extends Exp

case class Sum(exp1: Exp, exp2: Exp) extends Exp
case class Multiply(exp1: Exp, exp2: Exp) extends Exp
case class Divide(exp1: Exp, exp2: Exp) extends Exp
case class Square(exp: Exp) extends Exp
```

Example equation

4^2 + 3.3 * 4

```
Sum(
  Square(
    IntValue(4)
  ),
  Multiply(
    DecValue(3.3),
    IntValue(4)
  )
)
```

```scala
val evaluate: Exp => Double = {
  case DecValue(value) => value
  case IntValue(value) => value.toDouble
  case Sum(exp1, exp2) =>
    evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
    evaluate(exp1) * evaluate(exp2)
  case Divide(exp1, exp2) =>
    evaluate(exp1) / evaluate(exp2)
  case Square(exp) =>
    val v = evaluate(exp)
    v * v
}
```

```scala
val stringify: Exp => String = {
  case DecValue(value) => value.toString
  case IntValue(value) => value.toString
  case Sum(exp1, exp2) =>
    s"${stringify(exp1)} + ${stringify(exp2)}"
  case Square(exp) =>
    s"${stringify(exp)} ^ 2"
  case Multiply(exp1, exp2) =>
    s"${stringify(exp1)} * ${stringify(exp2)}"
  case Divide(exp1, exp2) =>
    s"${stringify(exp1)} / ${stringify(exp2)}"
}
```

It will work!

```
evaluate(expression) // prints 29.2
stringify(expression) // prints 4 ^ 2 + 3.3 * 4
```

but...

There are a lot of mess

```
case Sum(exp1: Exp, exp2: Exp) =>
  evaluate(exp1) + evaluate(exp2)
```

And not only...

Problem of partial interpretation

```
val optimize: Exp => Exp = {
  case Multiply(exp1, exp2) if exp1 == exp2 =>
    Square(optimize(exp1))
  case other => ???
}
```

Lets improve

02

going to result types

Generalize it

```
sealed trait Exp[T]
case class IntValue[T](value: Int) extends Exp[T]
case class DecValue[T](value: Double) extends Exp[T]

case class Sum[T](exp1: T, exp2: T) extends Exp[T]
case class Multiply[T](exp1: T, exp2: T) extends Exp[T]
case class Divide[T](exp1: T, exp2: T) extends Exp[T]
case class Square[T](exp: T) extends Exp[T]
```

Nesting types issue

```
val expression: Exp[Exp[Exp[Unit]]] =
Sum[Exp[Exp[Unit]]](
  Square[Exp[Unit]](
    IntValue[Unit](4)
  ),
  Multiply[Exp[Unit]](
    DecValue[Unit](3.3),
    IntValue[Unit](4)
  )
)
```

fixed point types

its like a hiding of part that doesn't matter

let's add typehack

```
case class Fix[F[_]](unFix: F[Fix[F]])
```

F[_] is a generic type with a hole, after wrapping of Exp with that we will have Fix[Exp] type.

Hacked code

```
val expression: Fix[Exp] = Fix(Sum(
  Fix(Square(
    Fix(IntValue(4))
  )),
  Fix(Multiply(
    Fix(DecValue(3.3)),
    Fix(IntValue(4))
  ))
))
```

Let's do it cleaner

```scala
case class Square[T](exp: T) extends Exp[T]
object Square {
  def apply(fixExp: Exp[Fix[Exp]]) = Fix[Exp](fixExp)
}

val expression: Fix[Exp] = Square(
   Square(
     Square(
       IntValue(4)
     )))
```

03

time to traverse our structure

Functor 911

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

Scalaz example

For cats you will have same code

```
case class Container[T](data: T)

 implicit val functor = new Functor[Container] {
  override def map[A, B](fa: Container[A])
                              (f: A => B): Container[B] =
    Container(f(fa.data))
 }

functor.map(Container(1))(_.toString) // "1"
```

```scala
implicit val functor = new Functor[Exp] {
  override def map[A, B](fa: Exp[A])(f: A => B): Exp[B] =
    fa match {
      case IntValue(v) => IntValue(v)
      case DecValue(v) => DecValue(v)
      case Sum(v1, v2) => Sum(f(v1), f(v2))
      case Multiply(v1, v2) => Multiply(f(v1), f(v2))
      case Divide(v1, v2) => Divide(f(v1), f(v2))
      case Square(v) => Square(f(v))
    }
}
```

04

catamorphism
anamorphism
hylomorphism

catamorphism

generalized folding operation

"Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire", by Erik Meijer

Lets define Algebra

```scala
type Algebra[F[_], A] = F[A] => A
val evaluate: Algebra[Exp, Double] = {
  case IntValue(v) => v.toDouble
  case DecValue(v) => v
  case Sum(v1, v2) => v1 + v2
  case Multiply(v1, v2) => v1 * v2
  case Divide(v1, v2) => v1 / v2
  case Square(v) => Math.sqrt(v)
}
```

So as a result

```
val expression: Fix[Exp] = Sum(
  Multiply(IntValue(4), IntValue(4)),
  Multiply(DecValue(3.3), IntValue(4))
)

import matryoshka.implicits._
expression.cata(evaluate) // 29.2
```

Extra profit

do you remember about partial interpretation?

```
val optimize: Algebra[Exp, Fix[Exp]] = {
  case Multiply(Fix(exp), Fix(exp2)) if exp == exp2 =>
    Fix(Square(exp))
  case other => Fix[Exp](other)
}

import matryoshka.implicits._
expression.cata(optimize).cata(stringify) // 4 ^ 2 + 3.3 * 4
```

```scala
val evaluate: Exp => Double = {
  case DecValue(value) => value
  case IntValue(value) => value.toDouble
  case Sum(exp1, exp2) =>
    evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
    evaluate(exp1) * evaluate(exp2)
  case Divide(exp1, exp2) =>
    evaluate(exp1) / evaluate(exp2)
  case Square(exp) =>
    val v = evaluate(exp)
    v * v
}
```

```scala
type Algebra[F[_], A]
val evaluate: Algebra[Exp, Double] = {
  case IntValue(v) => v.toDouble
  case DecValue(v) => v
  case Sum(v1, v2) => v1 + v2
  case Multiply(v1, v2) => v1 * v2
  case Divide(v1, v2) => v1 / v2
  case Square(v) => Math.sqrt(v)
}
```

Just good to know

catamorphism: F[_] => A
anamorphism: A => F[_]
hylomorphism: F[_] => A => F[_]

Summary

- Fixed point types is perfect to create DSLs
- Recursion schemas is composable
- All you need to use that stuff, you already know from Scala

thank you.

any questions?