

Intro to Recursion Schemes

Pepe García pepe.garcia@47deg.com

2021-02-09

Intro to recursion schemes

This talk introduces the topic of recursion schemes. At the end of it we use the Droste library.

Materials

You can see the slides, code, and infrastructure needed to build these slides in <https://github.com/pepegar/intro-recursion-schemes>

Recursion appears in a lot of different interesting problems we find in programming, from databases to compilers, graphics, etc.

Recursion

```
sealed trait Expr

object Expr {
  final case class BGP(triples: Seq[Tuple]) extends Expr
  final case class Triple(s: String, p: String, o: String) extends Expr
  final case class Union(l: Expr, r: Expr) extends Expr
  final case class Join(l: Expr, r: Expr) extends Expr
  final case class Graph(g: String, e: Expr) extends Expr
  final case class Construct(vars: Seq[String], bgp: Expr, r: Expr) extends Expr
  final case class Select(vars: Seq[String], r: Expr) extends Expr
}
```

A slightly modified version of the original one, I pruned some cases off the AST for brevity.

Let's see how we would represent this SparQL query in our AST using primitive recursion. A parser would be the process involved in this conversion.

```
CONSTRUCT
{
  ?d a dm:Document .
  ?d dm:docSource ?src .
}
WHERE
{
  ?d a dm:Document .
  ?d dm:docSource ?src .
}
```

Recursion

```
val expr: Expr = Expr.Construct(
  vars = List("?d", "?src"),
  bgp = Expr.BGP(List(Expr.Triple(
    "?d",
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
    "http://id.gsk.com/dm/1.0/Document"
  ),
  Expr.Triple("?d", "http://id.gsk.com/dm/1.0/docSource", "?src")))
),
r = Expr.BGP(List(Expr.Triple(
  "?d",
  "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
  "http://id.gsk.com/dm/1.0/Document"
),
Expr.Triple("?d", "http://id.gsk.com/dm/1.0/docSource", "?src")))
)
)
```

Counting the nodes in our AST

We could create a function that counted all nodes we have in our AST, let's see how!

```
def countNodes(expr: Expr): Int = expr match {  
  case Expr.BGP(triples) => 1 + triples.length  
  case Expr.Triple(s, p, o) => 1  
  case Expr.Union(l, r) => 1 + countNodes(l) + countNodes(r)  
  case Expr.Join(l, r) => 1 + countNodes(l) + countNodes(r)  
  case Expr.Construct(vars, bgp, r) => 1 + countNodes(bgp) + countNodes(r)  
  case Expr.Select(vars, r) => 1 + countNodes(r)  
}  
  
countNodes(expr)  
// res0: Int = 7
```

Visiting nodes

This pattern, recursing an AST in a bottom-up fashion, is so common that there's even a GoF entry for it, the **Visitor**.

```
trait Visitor[T] {  
  def visitBGP(x: Expr.BGP): T  
  def visitTriple(x: Expr.Triple): T  
  def visitUnion(x: Expr.Union): T  
  def visitJoin(x: Expr.Join): T  
  def visitConstruct(x: Expr.Construct): T  
  def visitSelect(x: Expr.Select): T  
}
```


Visiting nodes

With **applyVisitor** we create a general way of applying any **Visitor[T]** to our expression.

```
def applyVisitor[T](expr: Expr, visitor: Visitor[T]): T = expr match {  
  case x @ Expr.BGP(triples) => visitor.visitBGP(x)  
  case x @ Expr.Triple(s, p, o) => visitor.visitTriple(x)  
  case x @ Expr.Union(l, r) => visitor.visitUnion(x)  
  case x @ Expr.Join(l, r) => visitor.visitJoin(x)  
  case x @ Expr.Construct(vars, bgp, r) => visitor.visitConstruct(x)  
  case x @ Expr.Select(vars, r) => visitor.visitSelect(x)  
}
```

Visiting nodes

```
val countNodesVisitor: Visitor[Int] = new Visitor[Int] {  
  def visitBGP(x: Expr.BGP) = 1 + x.triples.length  
  def visitTriple(x: Expr.Triple) = 1  
  def visitUnion(x: Expr.Union) = 1 + countNodes(x.l) + countNodes(x.r)  
  def visitJoin(x: Expr.Join) = 1 + countNodes(x.l) + countNodes(x.r)  
  def visitConstruct(x: Expr.Construct) = 1 + countNodes(x.r)  
  def visitSelect(x: Expr.Select) = 1 + countNodes(x.r)  
}  
  
// countNodesVisitor: Visitor[Int] = repl.MdocSession$App$$anon$1@20a4d361  
  
applyVisitor(expr, countNodesVisitor)  
  
// res1: Int = 4
```

Recursion schemes are a way to formalize the concepts we're already used to from recursive programming.

Let's dive into some of the concepts we'll need for applying them.

Abstracting recursion away

In order to apply recursion schemes, we need to **factor recursion out** of our datatypes. There will be other types (**Fixpoint** types) in charge of tying the recursion knot.

```
sealed trait ExprF[A]
```

```
object ExprF {
```

```
  final case class BGPF[A](triples: Seq[Expr.Triple]) extends ExprF[A]
```

```
  final case class TripleF[A](s: String, p: String, o: String) extends ExprF[A]
```

```
  final case class UnionF[A](l: A, r: A) extends ExprF[A]
```

```
  final case class JoinF[A](l: A, r: A) extends ExprF[A]
```

```
  final case class GraphF[A](g: String, e: A) extends ExprF[A]
```

```
  final case class ConstructF[A](vars: Seq[String], bgp: Expr.BGP, r: A) extends ExprF[A]
```

```
  final case class SelectF[A](vars: Seq[String], r: A) extends ExprF[A]
```

```
}
```

Abstracting recursion away

We have abstracted recursion by introducing a new type parameter to our datatype, this made it possible to implement several interesting typeclasses for it:

```
Functor[ExprF] // will allow us to use 'map' on it  
Traverse[ExprF] // so that we can use 'traverse' on 'ExprF' values
```

This is why we will call this type our ***Pattern functor***.

Abstracting recursion away

But now our trees are not able to express recursion anymore!

```
def expr[A]: ExprF[A] =
  ExprF.JoinF[A](
    ExprF.BGPF[A](Seq(Expr.Triple("?s", "?p", "?o"))),
    ExprF.BGPF[A](Seq(Expr.Triple("?s", "?p", "?o")))
  )
// error: type mismatch;
// found   : repl.MdocSession.App.ExprF.BGPF[A]
// required: A
//      ExprF.BGPF[A](Seq(Expr.Triple("?s", "?p", "?o"))),
//      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// error: type mismatch;
// found   : repl.MdocSession.App.ExprF.BGPF[A]
// required: A
//      ExprF.BGPF[A](Seq(Expr.Triple("?s", "?p", "?o")))
//      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We don't have a **ExprF[A]**, but a **ExprF[ExprF[...]]**...

Fixpoint types

Fixpoint types will make it possible to express recursion again with our parametric ASTs.

There are several fixpoint types, the most common one is `Fix`.

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

Fixpoint types

Now that we have fixpoint types, we can express recursion in our AST again. We'll just need to interleave **Fix**.

```
val exprF: Fix[ExprF] =  
  Fix(  
    ExprF.JoinF(  
      Fix(ExprF.BGPF(Seq(Expr.Triple("?s", "?p", "?o")))),  
      Fix(ExprF.BGPF(Seq(Expr.Triple("?s", "?p", "?o"))))  
    )  
  )  
  
// exprF: Fix[ExprF] = Fix(  
//   unFix = JoinF(  
//     l = Fix(unFix = BGPF(triples = List(Triple(s = "?s", p = "?p", o = "?o")))),  
//     r = Fix(unFix = BGPF(triples = List(Triple(s = "?s", p = "?p", o = "?o"))))  
//   )  
// )
```


Fixpoint types

Fix is not the only fixpoint type. There's **Attr** for example, with which we can add annotations to nodes in our tree.

And there's also **Expr**. Our first ADT, **Expr** is a fixpoint of **ExprF**!

Fixpoint types

Droste provides a couple of typeclasses for that relate **pattern functors** and **fixpoint types** together.

```
trait Embed[F[_], R] {  
  def embed(fa: F[R]): R  
}  
  
trait Project[F[_], R] {  
  def embed(r: R): F[R]  
}
```

Embed takes an instance of the pattern functor and puts it inside the fixpoint type.

Project takes a fixpoint type and peels of a layer, exposing the pattern functor.

@deriveFixedPoint

All the boilerplate we have generated by parametrizing our tree, we can generate it at compile time with the **@deriveFixedPoint** macro annotation from droste!

```
import cats.implicits._
import higherkindness.droste.macros.deriveFixedPoint

@deriveFixedPoint sealed trait Expr2

object Expr2 {
  final case class BGP(triples: List[Expr2]) extends Expr2
  final case class Triple(s: String, p: String, o: String) extends Expr2
  final case class Union(l: Expr2, r: Expr2) extends Expr2
  final case class Join(l: Expr2, r: Expr2) extends Expr2
  final case class Construct(vars: Seq[String], bgp: Expr2, r: Expr2) extends Expr2
  final case class Select(vars: Seq[String], r: Expr2) extends Expr2
}
```

This macro annotation will generate a new object `fixedpoint` inside the companion object of our ADT with all the boilerplate.

```
// ...  
object fixedpoint {  
  implicit val embed: Embed[ExprF, Expr2] = ???  
  implicit val project: Project[ExprF, Expr2] = ???  
}
```

We have learned how to setup our datatypes in order to apply recursion schemes.
Now let's see some actual recursion schemes!

Folding (consuming trees)

In order to consume a recursive structure, we can use a **catamorphism**.

Catamorphisms consume a recursive value and produce something out of it. `fold`, or `reduce` are catamorphisms.

Algebras are like Visitors, but generic on the pattern functor.

```
/**
 * type Algebra[F[_], A] = F[A] => A
 */
val countNodes: Algebra[Expr2F, Int] = Algebra {
  case BGPF(triples) => 1 + triples.length
  case TripleF(s, p, o) => 1
  case UnionF(l, r) => 1 + 1 + r
  case JoinF(l, r) => 1 + 1 + r
  case ConstructF(vars, bgp, r) => 1 + bgp + r
  case SelectF(vars, r) => 1 + r
}

val count = scheme.cata(countNodes)
```

Folding (consuming trees)

```
count(expr2)  
// res3: Int = 7
```

Unfolding (producing new trees)

Unfolding is the **dual** of folding, meaning that we'll produce new recursive expressions of plain values. We can use an anamorphism for unfolding.

Coalgebras are the dual of Algebras.

```
/**  
 * type Coalgebra[F[_], A] = A => F[A]  
 */  
val toPatternFunctor: Coalgebra[Expr2F, Expr2] =  
  Coalgebra[Expr2F, Expr2] {  
    case Expr2.BGP(triples) => BGPF(triples)  
    case Expr2.Triple(s, p, o) => TripleF(s, p, o)  
    case Expr2.Union(l, r) => UnionF(l, r)  
    case Expr2.Join(l, r) => JoinF(l, r)  
    case Expr2.Construct(vars, bgp, r) => ConstructF(vars, bgp, r)  
    case Expr2.Select(vars, r) => SelectF(vars, r)  
  }  
  
val convert = scheme.ana(toPatternFunctor)
```


Unfolding (producing new trees)

```
convert(expr2)
// res4: Expr2 = Construct(
//   vars = List("?d", "?src"),
//   bgp = BGP(
//     triples = List(
//       Triple(
//         s = "?d",
//         p = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
//         o = "http://id.gsk.com/dm/1.0/Document"
//       ),
//       Triple(s = "?d", p = "http://id.gsk.com/dm/1.0/docSource", o = "?src")
//     )
//   ),
//   r = BGP(
//     triples = List(
//       Triple(
//         s = "?d",
//         p = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
//         o = "http://id.gsk.com/dm/1.0/Document"
//       )
//     )
//   )
// )
```

re-folding (folding after unfolding)

We can already compose our functions to convert our datatype and then calculate the number of nodes.

```
val composed: Expr2 => Int = expr => count(convert(expr))  
// composed: Expr2 => Int = <function1>  
  
composed(expr2)  
// res5: Int = 7
```

re-folding (folding after unfolding)

However with recursion schemes we can use a **refold**, that is, a recursion scheme that applies a fold after an unfold! The most common one is the **hylomorphism**:

```
val refold = scheme.hylo(countNodes, toPatternFunctor)
// refold: Expr2 => Int = <function1>

refold(expr2)
// res6: Int = 7
```

The benefit we get from `hylo` is that it applies **fusion**, a technique in which intermediate values are not fully built, but passed from the **coalgebra** to the **algebra** as soon as they're created.

Zippping

Something else that recursion schemes provide is zipping so that in a single pass we can apply more than one Algebra.

```
// create _something like_ Apache Jena representation of SparQL algebra
val lispify: Algebra[Expr2F, String] = Algebra {
  case BGPF(triples) => s"(bgp\n${triples.mkString("\n")})"
  case TripleF(s, p, o) => s"(triple $s $p $o)"
  case UnionF(l, r) => s"(union\n$l $r)"
  case JoinF(l, r) => s"(join\n$l $r)"
  case ConstructF(vars, bgp, r) => s"(construct (${vars.mkString(" ")})\n$bgp\n$r)"
  case SelectF(vars, r) => s"(select (${vars.mkString(" ")})\n$r)"
}

// Zip algebras together
val countNodesAndLispify: Algebra[Expr2F, (Int, String)] = countNodes.zip(lispify)

val rerefold = scheme.hylo(countNodesAndLispify, toPatternFunctor)
```

Ziping

```
rerefold(expr2)
// res7: (Int, String) = (
//   7,
//   ""(construct (?d ?src)
// (bgp
// (triple ?d http://www.w3.org/1999/02/22-rdf-syntax-ns#type http://id.gsk.com/dm/1.0/D
// (triple ?d http://id.gsk.com/dm/1.0/docSource ?src))
// (bgp
// (triple ?d http://www.w3.org/1999/02/22-rdf-syntax-ns#type http://id.gsk.com/dm/1.0/D
// (triple ?d http://id.gsk.com/dm/1.0/docSource ?src)))""
// )
```

More things we get with recursion schemes

- Use a different fixpoint type to add different semantics. (Attr to add annotations to nodes in the AST, Coattr to add annotations to leaves...)
- Visualize our ASTs with droste-reftree.

- Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire
- Recursion schemes fundamentals
- Recursion schemes series, by Patrick Thompson