Intro to Recursion Schemes

Pepe García pepe.garcia@47deg.com

2021-02-09

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

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

Recursion

Let's see how we would represent this SparQL query in our AST using primitive recursion. A parser would be the process involved in the conversion from a String like this to a Scala datatype.

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

Recursion

Here's the datatype we're going to focus on during the whole talk. It represents a SparQL algebra.

```scala
sealed trait Expr

object Expr {
  final case class BGP(triples: Seq[Triple]) 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.

Recursion

After parsing we would get something like this.

```
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

Here we can see how we create a function that counts nodes in the tree using **primitive recursion**.

```scala
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`**.

```scala
trait Visitor[T] {
  def visitBGP(triples: Seq[Expr.Triple]): T
  def visitTriple(s: String, p: String, o: String): T
  def visitUnion(l: T, r: T): T
  def visitJoin(l: T, r: T): T
  def visitConstruct(vars: Seq[String], bgp: T, r: T): T
  def visitSelect(vars: Seq[String], r: T): T
}
```

Notice how, every time recursion appeared, now we're using our generic type **`T`**.

Visiting nodes

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

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

Visiting nodes

```scala
val countNodesVisitor: Visitor[Int] = new Visitor[Int] {
  def visitBGP(triples: Seq[Expr.Triple]) = 1 + triples.length
  def visitTriple(s: String, p: String, o: String) = 1
  def visitUnion(l: Int, r: Int) = 1 + l + r
  def visitJoin(l: Int, r: Int) = 1 + l + r
  def visitConstruct(vars: Seq[String], bgp: Int, r: Int) = 1 + bgp + r
  def visitSelect(vars: Seq[String], r: Int) = 1 + r
}
// countNodesVisitor: Visitor[Int] = repl.MdocSession$App$$anon$1@53fc68f7

applyVisitor(expr, countNodesVisitor)
// res1: Int = 7
```

Recursion schemes

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.

```scala
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]
}
```

We'll substitute **primitive recursion** with the newly introduced generic type.

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

```scala
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")))
//     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Abstracting recursion away

```scala
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")))
//     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Warning

Now we can't express recursion anymore, 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`**.

```scala
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(
//   JoinF(
//     Fix(BGPF(List(Triple("?s", "?p", "?o")))),
//     Fix(BGPF(List(Triple("?s", "?p", "?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.

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

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

trait Basis[F[_], R] extends Embed[F, R] with Project[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!

```scala
import cats.implicits._
import higherkindness.droste._
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
}
```

`@deriveFixedPoint`

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

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

folds

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 + l + r
  case JoinF(l, r) => 1 + l + 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(
//   List("?d", "?src"),
//   BGP(
//     List(
//       Triple(
//         "?d",
//         "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
//         "http://id.gsk.com/dm/1.0/Document"
//       ),
//       Triple("?d", "http://id.gsk.com/dm/1.0/docSource", "?src")
//     )
//   ),
//   BGP(
//     List(
//       Triple(
//         "?d",
//         "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
//         "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 appilies 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.

Zipping

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

```scala
// 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)
```

Zipping

```
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/Do
// (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/Do
// (triple ?d http://id.gsk.com/dm/1.0/docSource ?src)))"""
// )
```

Optimizations

Something very interesting we can do with recursion schemes is optimizations to our AST. We'll use the `Trans` datatype from Droste.

Optimizations

Something very interesting we can do with recursion schemes is optimizations to our AST. We'll use the `Trans` datatype from Droste.

\begin{alertblock}{This is a showcase IDK if it is semantically correct for SparQL}
Let's say we want to translate Join operations of BGP to just BPGs.
\end{alerblock}

Optimizations

Something very interesting we can do with recursion schemes is optimizations to our AST. We'll use the `Trans` datatype from Droste.

\begin{alertblock}{This is a showcase IDK if it is semantically correct for SparQL} Let's say we want to translate Join operations of BGP to just BPGs. \end{alerblock}

```scala
def joinsToBGPs[T](implicit T: Basis[Expr2F, T]): Trans[Expr2F, Expr2F, T] =
  Trans {
    case j @ JoinF(l, r) =>
      (T.coalgebra(l), T.coalgebra(r)) match {
        case (BGPF(t1), BGPF(t2)) => BGPF(t1 ++ t2)
        case _ => j
      }
    case otherwise => otherwise
  }
```

Optimizations

```scala
val join = Expr2.Join(
  Expr2.BGP(List(Expr2.Triple("?s", ":type", ":Doc"))),
  Expr2.BGP(List(Expr2.Triple("?s", ":references", ":Ontology"))),
)

val transformAndLispify = scheme.hylo(lispify, joinsToBGPs.coalgebra)
```

```scala
transformAndLispify(join)
// res8: String = """(bgp
// (triple ?s :type :Doc)
// (triple ?s :references :Ontology))"""
```

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`.

References

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