DSLs and Recursion Schemes

Marek Kidoň

10.11.2018

Introduction

TOC

Domain

Domain Expert View

We will need to implement a health card check. Come on how hard can it be?
– Domain Expert

Hey a customer requires us to check vocational certificates. Come on how hard can it be when you already did those health cards?
– Domain Expert

I need . . .
– Domain Expert

Hey man I'm not touching this. I will give you a tool so you can define those yourself. . .
– Software Engineer

Feature

A distinctive attribute or aspect of something.
– Google

- Date of birth
- Category B driving license
- FIT VUT student
- Well built figure (suitable for carrying heavy stuff)

Are among the examples...

Feature Predicate

We would like to build a predicate language upon *Feature*s

1. Each `Feature Predicate` is bound to exactly one `Feature`
2. A simple arithmetic DSL sould do the trick
3. Every DSL should yield a `Boolean` value

   Are you an adult?

```
$featureVal + years(18) >= $now
```

DSLs for Help (the naive approach)

DSL

Arithmetic *DSLs* are *recursive* in its nature:

```
sealed trait Expr

object Expr {
  final case class Op(op: OpType, opl: Expr, opr: Expr)
    extends Expr
  final case class Const(lit: Literal) extends Expr
  final case class Var(name: String) extends Expr
}
```

Feature Predicate Language

Operations

```scala
sealed trait OpType

object OpType {
  final case object Add extends OpType
  final case object Eq  extends OpType
}
```

Literals

```scala
object Literal {
  final case class IntLit(int: Int) extends Literal
  final case class BoolLit(bool: Boolean) extends Literal
}
```

Operations performed upon this DSL: `optimize`

The following algebras should be optimized:

- `(Int, +)`
- `(Int, ==)`
- `(Bool, ==)`

```
val optimize: Expr => Expr = {
  // Recurse: BOILERPLATE...
  case Op(op, lop, rop) =>
    optimize(Op(op, optimize(lop), optimize(rop)))
  // Many more cases...
  case ...
}
```

Operations performed upon this DSL: `subst`

Before evaluation DSL expr a `variables` should be substituted.

```
type Env = Map[String, Expr]

def subst: Env => Expr => Expr = env => {
  case v@Var(name)       => env get name getOrElse v
  // Recurse: BOILERPLATE...
  case Op(op, lop, rop) =>
    Op(op, subst(env)(lop), subst(env)(rop))
  case e: Expr           => e
}
```

Operations performed upon this DSL: eval

```
def eval: Env => Expr => Expr =
  subst(_) andThen optimize
```

Limits of this model

Unofortunately it does not compose well:

- `eval` requires to traverse `Expr` tree twice (`optimize`, `subst`)
- How can I decouple `optimizer` into separate pieces?

In Scala `optimizer` can be defined in terms of `PartialFunction`

- It is a *sort of* solution
- Still does not compose well with other operations

Coupling `eval` into single feature:

- Hard to test, it is utterly complex,
- Code duplicity (2x `optimize`)

Recursion schemes

Fixed point (Fixpoint)

TODO
– *Wikipedia*

Abstract Over Recursion

Recursive grammar

```scala
sealed trait IntList

case class IntCons(i: Int, is: IntList) extends IntList
case object IntNil                      extends IntList
val intList = IntCons(3, IntCons(2, IntCons(1, IntNil)))
```

Abstract Over Recursion

Recursion can be abstracted away revealing the true primitives.

1. Find a non-recursive grammar *precursor*
2. Find its fixed-point data type

Abstract Over Recursion

Non-recursive *precursor*

```
sealed trait IntListF[A]

case class IntConsF[A](i: Int, is: A) extends IntListF[A]
case class IntNilF[A]()                extends IntListF[A]
```

Fixed point data type

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

Grammar

```
type IntList = Fix[IntListF]
val intList1 = Fix(IntConsF(3, Fix(IntConsF(2, ... ))))
```

Evaluation for Free

> *Evaluation is a recipe for extracting a single value from an expression.*
> *– Bartosz Milewski*

```
val sum: IntList => Int = ???
```

In order to do that we need to:

1. Find a recipe to produce a single value from `IntListF` and `Fix`
2. Combine them

Evaluation for Free: Algebra

Function of type:

```
type Algebra: F[A] => A
```

is called an `Algebra` (`F-Algebra`) where:

1. `F[_]` is a functor
2. `A` is a carrier type

Evaluation for Free: Algebra

There are many `Algebras` based on a given `IntListF[_]`

```scala
def sumAlgebra: Algebra[IntListF, Int] = {
  case IntConsF(i, is) => i + is
  case IntNilF()       => 0
}

def printAlgebra: Algebra[IntListF, String] = {
  case IntConsF(i, is) => s"[$i $is]"
  case IntNilF()       => ""
}
```

Evaluation for Free: Initial Algebra

But there is one algebra to rule them all. The `InitialAlgebra`:

```
type InitialAlgebra[F[_]] = Algebra[F, Fix[F]]
```

1. It is not lossy
   - Always preservers the structure
   - Does not produce a summary
2. It is at least as powerful as all other algebras

```
val addOneAlgebra: InitialAlgebra[IntListF] = {
  case IntConsF(i, is) => Fix(IntConsF(i + 1, is))
  case IntNilF()       => Fix(IntNilF())
}
```

Evaluation for Free: Initial Algebra

The property to die for: `InitialAlgebras` do compose:

```
def compose[F[_], A](
    phi: InitialAlgebra[F],
    psi: InitialAlgebra[F]
): InitialAlgebra[F] =
  phi compose unFix compose psi

val addTwoAlgebra: InitialAlgebra[IntListF] =
  compose(addOneAlgebra, addOneAlgebra)
```

Evaluation for Free: Eval

TODO: Introduce catamorphism... possibly via

1. Initial algebra Homomorphism proof
2. Or a diagram

DSLs for Help (better approach)

DSL - Revisited

Non-recursice precursor of our DSL.

```scala
sealed trait ExprF[A]

object ExprF {
  case class Op[A](op: OpType, opl: A, opr: A)
    extends ExprF[A]
  case class Const[A](lit: Literal) extends ExprF[A]
  case class Var[A](name: Name) extends ExprF[A]
}

type Expr = Fix[ExprF]
```

DSL - Revisited: `optimize`

Define `optimize` in terms of an `InitialAlgebra`:

```scala
val optimizeIntAddA: InitialAlgebra[ExprF] = {
  case Op(Add, ConstInt(i1), ConstInt(i2)) =>
    int(i1 + i2)
  case e: ExprF[Expr] => Fix(e)
}

val optimizeA: InitialAlgebra[ExprF] = andThenAll(
  List(optimizeIntEqA, optimizeIntAddA, optimizeBoolEqA)
)
```

DSL - Revisited: subst

Define subst in terms of an InitialAlgebra:

```
val substA: Env => InitialAlgebra[ExprF] = env => {
  case v@Var(name)    => env get name getOrElse Fix(v)
  case e: ExprF[Expr] => Fix(e)
}
```

DSL - Revisited: `eval`

Define `eval` in terms of an `InitialAlgebra`:

```
val evalA: Env => InitialAlgebra[ExprF] =
  env => andThen(substA(env), optimizeA)

val eval: Env => Expr => Expr = env => _ cata evalA(env)
```

Conclusion

How about `typeCheck`

```
def typeCheck[F[_]: Monad]: Expr => F[Expr]
```

Typechecking in general

1. Is not total
2. Can't be expressed via catamorhism

However

1. There is an `InitialMAlgebra`
2. There is more than just catamorhism

Toolbox

Generalized Recursion Schemes libraries

1. Entire ZOO of morphisms
2. Define numerous `Fix[F[_]]`-like recursion types (`Free`, `Cofree`, `Mu`)

Matryoshka (Scala)
recursion-schemes (Haskell)