YIN-YANG

PROGRAMMING DSLS MADE SIMPLE

By Vojin Jovanovic / @vojjov

DOMAIN-SPECIFIC PROGRAMMING LANGUAGE

Programming language restricted to a narrow domain in order to improve both productivity and performance.

EXAMPLE: STANDARD QUERY LANGUAGE

```
SELECT name, email
FROM   gmail_accounts as m, github_account as g
WHERE  m.username = g.username
  AND prog_language = "Scala"
  AND lang_level > 30
```

SQL: DOMAIN KNOWLEDGE

Rewrites can change performance by a factor of 1000:

- $R \bowtie S = S \bowtie R$
- $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

A DSL SHOULD BE:

- User Friendly
- Fast

MEET THE *DSL USER*

However, a DSL *author* should:

- Implement a parser
- Implement a typechecker
- Implement a debugger
- Provide IDE integration

MEET THE *DSL AUTHORS*

EMBEDDED DOMAIN-SPECIFIC LANGUAGES

Use an existing general purpose *host language* to embed a DSL.

DIRECT EMBEDDING

Values in the embedded language are directly represented by the values in the host language.

DIRECT EMBEDDING - LOW PERFORMANCE

Large abstraction overhead. Example the Scala collections:

```
val v1 = Seq(1,2,3)
val v2 = Seq(4,5,6)
val sum = (v1 zip v2) map { case (x, y) => x + y }
val (min, max) = (sum.max, sum.min)
```

DEEP EMBEDDING

Values in the embedded language are symbolically represented by the values in the host language.

```
// symbolic representation
trait Rep[T]
case class Const(v: Int) extends Rep[Int]
case class Minus(l: Rep[Int], r: Rep[Int]) extends Rep[Int]
// lifting
implicit def lift(i: Int): Rep[Int] = Const(i)
implicit class IntOps(l: Rep[Int]) {
  def -(r: Rep[Int]): Rep[Int] = Minus(l, r)
}

val y: Rep[Int] = 1024
val x: Rep[Int] = y - 1
println(x)
```

DEEP EMBEDDING - TROUBLES DEBUGGING

```
}
val y = pow(2, 10)
val x = y - 1
println(x)
```

```
implicit def lift(i: Int): Rep[Int] = Const(i)
implicit class IntOps(lhs: Rep[Int]) {
  def -(rhs: Rep[Int]): Rep[Int] = Minus(lhs, rhs)
}
```

DEEP EMBEDDING - CRYPTIC TYPE ERRORS

```
Query(Coffees).map(c =>
  if(c.origin == "CH")
    "Good"
  else
    c.quality
)
```

- Don’t know how to unpack Any to T and pack to G
- Not enough arguments for method map: (implicit shape: scala.ql.lifted.Shape[Any,T,G]):scala.slick.lifted.Query[G,T]

DIRECT EMBEDDING VS. DEEP EMBEDDING

	Deep	Direct
Friendly	X	✓
Fast	✓	X

INSIGHT

User friendliness matters during program development.

Performance matters in production.

Automatically *translate* a friendly direct embedding to a fast deep embedding before deployment!

AUTOMATIC TRANSLATION

SPOT THE DIFFERENCE BEHIND THE SCENES

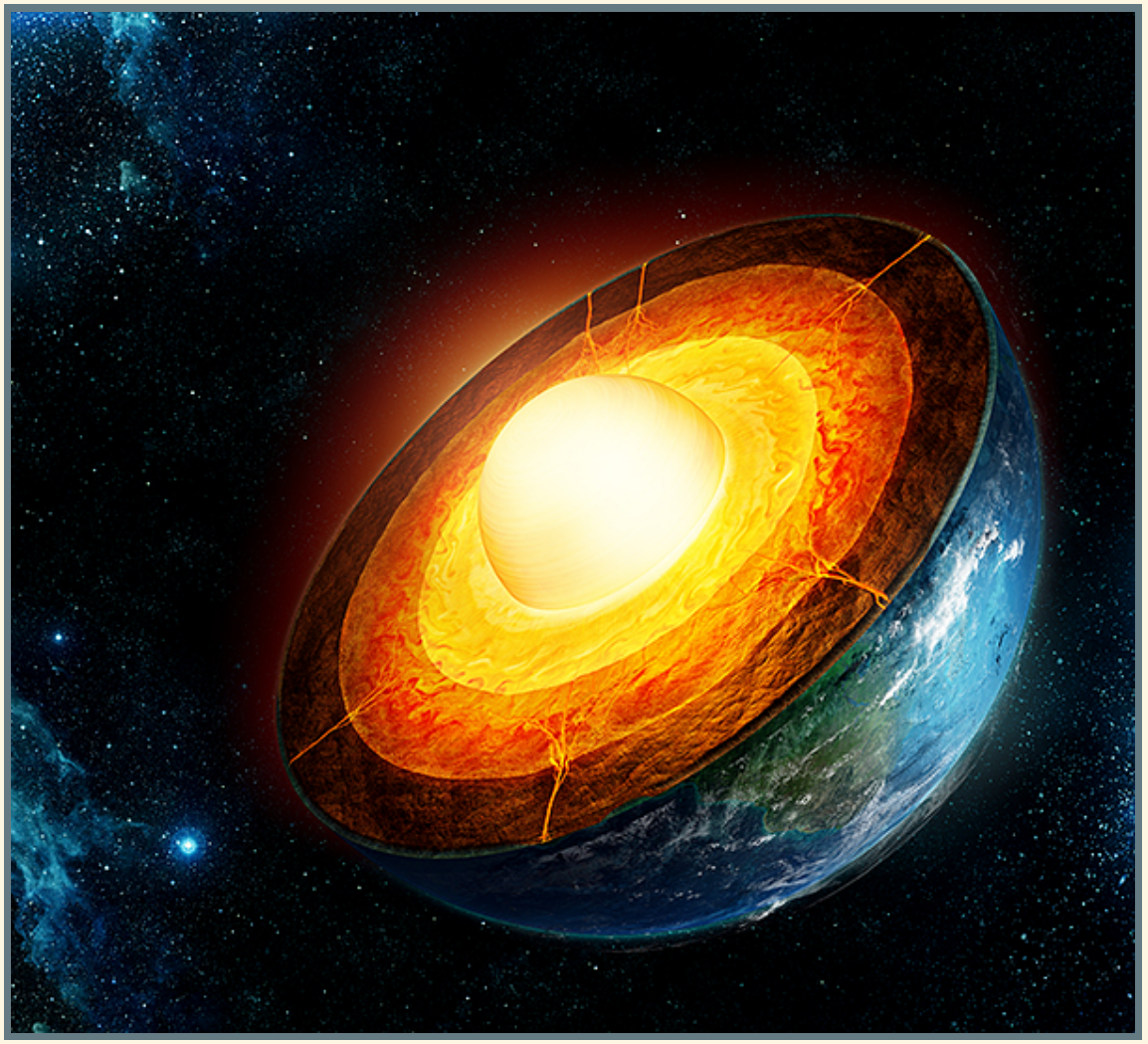
Direct

```
{
  Query(Coffees).map(c =>
    if(c.origin == "CH")
      "Good"
    else
      c.quality
  )
}
```

Deep

```
query { // scala-virtualized
  Query(Coffees).map(c =>
    if(c.origin == "CH")
      "Good"
    else
      c.quality
  )
}
```

THE CORE TRANSLATION



LANGUAGE VIRTUALIZATION

	Direct	Deep
Language Constructs		
	if(c) t else e	__if(c, t, e)
	while(c) b	__while(c, b)
`AnyRef` Methods		
	x.hashCode	infix_hashCode(x)
	a == b	infix_==(a, b)

DSL INTRINSIFICATION

Converts DSL intrinsics from the direct embedding into their deep counterparts.

OPERATION TRANSLATION

- Injects the whole program into the DSL component
- Replaces prefixes with `this`

Direct

```
query {
  Query(Coffees).map(c =>
    if(c.origin == "CH")
      "Good"
    else
      c.quality
  )
}
```

Deep

```
new QueryDSL {
  this.Query(Coffees).map(c =>
    if(c.origin == "CH")
      "Good"
    else
      c.quality
  )
}
```

CONSTANTS AND CAPTURED VARIABLES

```
def lift[T, Ret](v: T)
          (implicit liftEv: LiftEvidence[T, Ret]): Ret =
      liftEv.lift(v)
```

```
new QueryDSL {
  this.Query(Coffees).map(c =>
    if(c.origin == "CH")
      "Good"
    else
      c.quality
  )
}
```

TYPE TRANSLATION: GENERIC EMBEDDING

$$\tau(T, \texttt{targ}) = T$$

$$\tau(T, \texttt{other}) = Rep[T]$$

TYPE TRANSLATION: AUTOMATIC INLINING

$$\tau(T_1 \Rightarrow T_2, \texttt{targ}) = \texttt{error}$$

$$\tau(T_1 \Rightarrow T_2, \texttt{other}) = \tau(T_1, \texttt{other}) \Rightarrow \tau(T_2, \texttt{other})$$

$$\tau(T, \texttt{targ}) = T$$

$$\tau(T, \texttt{other}) = \mathit{Rep}[T]$$

CORRECTNESS

Finally Tagless, Partially Evaluated

Tagless Staged Interpreters for Simpler Typed Languages

Jacques Carette, Oleg Kiselyov and Chung-chieh Shan

Abstract

We have built the first *family* of tagless interpretations for a higher-order typed object language in a typed metalanguage (Haskell or ML) that require no dependent types, generalized algebraic data types, or postprocessing to eliminate tags. The statically type-preserving interpretations include an evaluator, a compiler (or staged evaluator), a partial evaluator, and call-by-name and call-by-value CPS transformers.

LANGUAGE RESTRICTION ANALYSIS

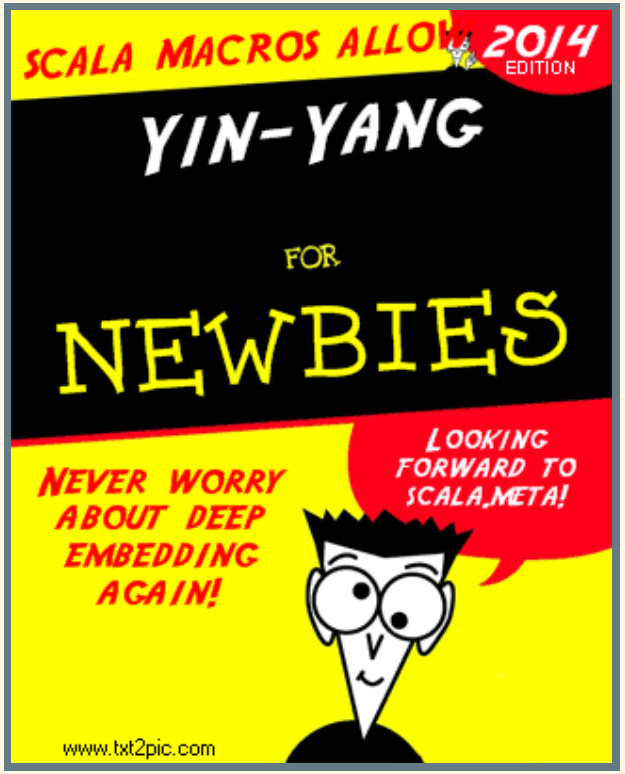
Direct embedding API can be richer than the deep embedding.

```
new QueryDSL {
  this.Query(lift(Coffees)).map { c: Column[T] =>
    if(c.origin == lift("CH")) {
      lift("Good")
    } else
      c.quality
  }
}
```

LANGUAGE RESTRICTION

Check if the direct embedding complies with the deep embedding in a fine-grained way and report comprehensible error messages.

```
new QueryDSL {
  this.Predef.println(().asInstanceOf[Rep[String]])
}
```


SCALA MACROS ALLOW
2014
EDITION
YIN-YANG
FOR
NEWBIES
NEVER WORRY ABOUT DEEP EMBEDDING AGAIN!
LOOKING FORWARD TO SCALA.META!
www.txt2pic.com

DSL USER VIEW

```
query {
  Query(Coffees).map(c =>
    if(c.origin == "CH")
      "Good"
    else
      c.quality
  )
}
```

DSL AUTHOR VIEW

```
object YYTransformer {
  def apply[C <: Context, T](c: C)(
    dslName: String,
    tpeTransformer: TypeTransformer[c.type],
    config: Map[String, Any] = Map()) = ...
}
```

```
def query[T](block: => T): T = macro _query[T]
def _query[T](c: Context)(block: c.Expr[T]): c.Expr[T] =
  YYTransformer[c.type, T](c)(
    "slick.Query",
    new RepTransformer[c.type](c))(block)
```

EXPERIMENT: SLICK FRONT-END

SLICK

Compiles functional programs into efficient SQL. Existing deep embedding based on Rep types.

NEW INTERFACE

```
class Query[T] {
  def length: Int = ???
  def map[S](projection: T => S): Query[S] = ???
  def filter(projection: T => Boolean): Query[T] = ???
  def withFilter(projection: T => Boolean): Query[T] = ???
  def flatMap[S](projection: T => Query[S]): Query[S] = ???
  // ...
  def groupBy[S](f: T => S): Query[(S, Query[T])] = ???
  def innerJoin[S](q2: Query[S]): JoinQuery[T, S] = ???
  def leftJoin[S](q2: Query[S]): JoinQuery[T, S] = ???
  def rightJoin[S](q2: Query[S]): JoinQuery[T, S] = ???
  def outerJoin[S](q2: Query[S]): JoinQuery[T, S] = ???
}
```

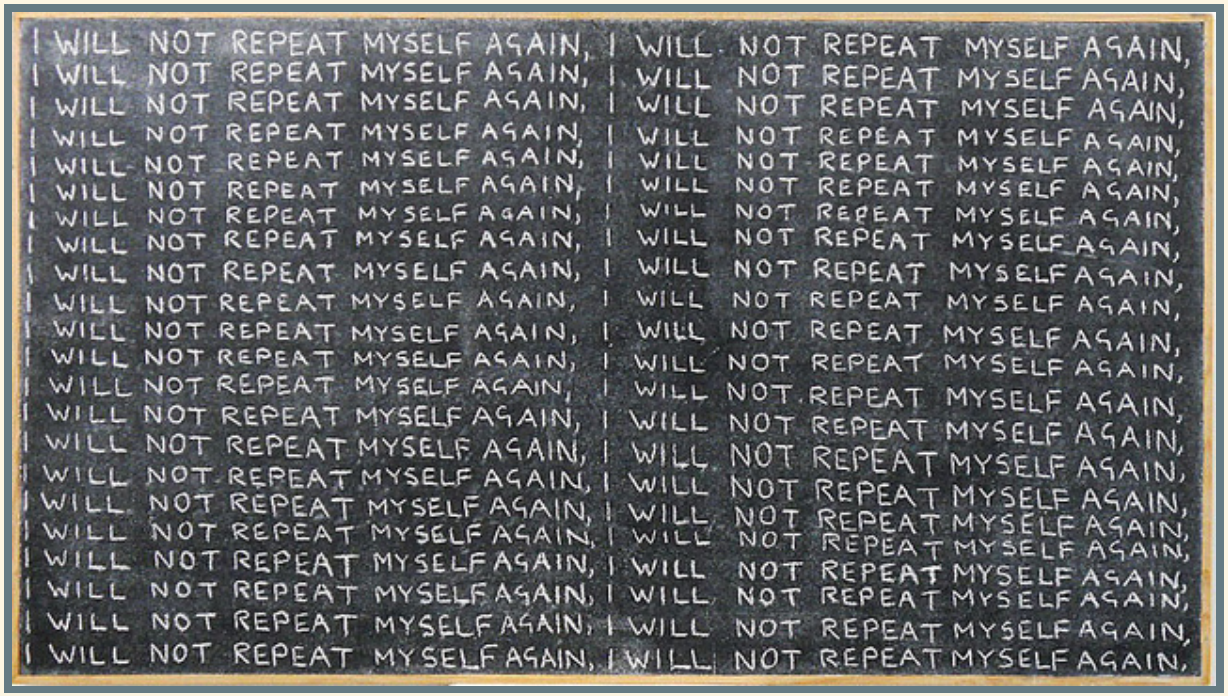
WRAPPER

```
trait QueryOps[T] { self: YYQuery[T] =>
  def withFilter(p: YYRep[T] => YYRep[Boolean]): YYQuery[T] =
    filter(p)
  def flatMap[S](p: YYRep[T] => YYQuery[S]): YYQuery[S] =
    YYQuery.fromQuery(query flatMap { (x: Rep[T]) =>
      p(YYValue[T, E](x)).query
  })
}
```

IMPLEMENTATION

- 2 months of development
- ~1000 lines of code
- 50+ tests are passing

WHAT ABOUT CONSTRUCTING A NEW DSL?



DEEP EMBEDDING GENERATION

```
object BigDecimal {
  def apply(s: String): BigDecimal =
    new BigDecimal(new JBigDecimal(s))
}

class BigDecimal(val v: JBigDecimal) {
  def +(lhs: BigDecimal): BigDecimal =
    new BigDecimal(v.add(lhs.v))
  def *(lhs: BigDecimal): BigDecimal =
    new BigDecimal(v.multiply(lhs.v))
}
```

GENERATING AN INTERFACE

```
trait BigDecimalOps extends Base with OverloadHack {
  object BigDecimal {
    def apply(s: Rep[String]): Rep[BigDecimal] = // ...
  }
  implicit class BigDecimalRep(self: Rep[BigDecimal]) {
    def +(lhs: Rep[BigDecimal]): Rep[BigDecimal] = // ...
    def *(lhs: Rep[BigDecimal]): Rep[BigDecimal] = // ...
  }
}
```

GENERATE AN INTERMEDIATE REPRESENTATION

```
trait BigDecimalExp extends BigDecimalOps with BaseExp {
  // case classes
  case class BigDecimalNew
    (v: Rep[BigDecimal]) extends Def[BigDecimal]
  case class BigDecimal$plus
    (self: Rep[BigDecimal], lhs: Rep[BigDecimal]) extends Def[BigDecimal]
  case class BigDecimal$times
    (self: Rep[BigDecimal], lhs: Rep[BigDecimal]) extends Def[BigDecimal]
  case class Apply(s: Rep[String]) extends Def[BigDecimal]
  // ...
}
```

GENERATE CODE GENERATION

```
override def emitNode(sym: Sym[Any], node: Def[Any]): Unit = node match {
  case BigDecimalNew(v) =>
    stream.print("val " + quote(sym) + " = new BigDecimal")
    stream.print("(")
    stream.print(quote(v))
    stream.print(")")
    stream.println("")
    // ...
}
```

AND FINALLY

```
trait BigDecimalExpOpt extends BigDecimalExp {
  override def bigDecimal$plus
    (self: Rep[BigDecimal], lhs: Rep[BigDecimal]): Rep[BigDecimal] = {
    /* please add optimizations here */
    super.bigDecimal$plus(self, lhs)
  }
  override def bigDecimal$times
    (self: Rep[BigDecimal], lhs: Rep[BigDecimal]): Rep[BigDecimal] = {
    /* please add optimizations here */
    super.bigDecimal$times(self, lhs)
  }
  // ...
}
```

DEMO

TAKEAWAYS

Easier to program DSLs for both DSL users and DSL authors.

ARE WE THERE YET?

THE TEAM BEHIND YIN-YANG:

- Vojin Jovanovic
- Sandro Stucki
- Amir Shaikhha
- Vladimir Nikolaev
- Vera Salvisberg

THE END

Yin-Yang / http://github.com/vjovanov/yin-yang

By Vojin Jovanovic / @vojjov