Polymorphic Record Types in a Lifted Embedding

Stefan Zeiger

Join the conversation #scaladays

Slick

Slick

- Write database queries in Scala (like using collections)

```
val q = users.filter(_.id < 42).map(_.first)
```

- Run them on a database

```
val result = db.run(q)
```

- Statically typed

```
val result: Future[Vector[String]] = db.run(q)
```

slick.lightbend.com

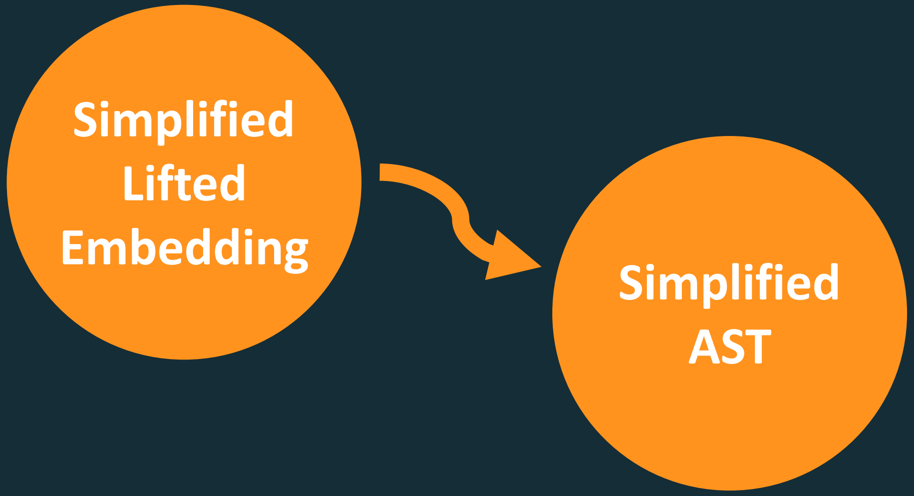
Slick

1. Write query in Slick's *Lifted Embedding* Scala DSL
   - Plain Scala – No macros, preprocessing, etc.
2. Lifted Embedding builds a Slick AST
   - Reify the computations
3. AST is compiled to SQL statement
4. Statement gets executed on a database (via JDBC)
5. Results delivered asynchronously as *Future* or *Reactive Stream*

slick.lightbend.com

Toy Slick

- No query execution
- No query compilation
- Simple, untyped AST
- No profiles
- Fewer operations
- No Option types
- No type constructors (always *Seq*)
- No ShapeLevels



Simplified Lifted Embedding

Simplified AST

https://github.com/szeiger/slick/tree/toy-slick-scaladays2016

Abstract Syntax Tree (AST)

Toy Slick AST

```scala
sealed trait Node

case class LiteralNode(value: Any) extends Node
case class ProductNode(children: Vector[Node]) extends Node
case class TableNode(name: String) extends Node

case class MapNode(sym: Symbol, from: Node, select: Node) extends Node
case class Filter(sym: Symbol, from: Node, where: Node) extends Node
case class Ref(sym: Symbol) extends Node
case class Select(in: Node, field: Symbol) extends Node
case class Apply(f: Symbol, children: Vector[Node]) extends Node

case class Symbol(name: String)
```

Toy Slick AST

```scala
val q = users.filter(_.id < 42).map(u => u.first)
```

```
Map
┣ from s2: Filter
┃ ┣ from s1: Table users
┃ ┗ where: Apply <
┃     ┣ 0: Select id
┃     ┃ ┗ in: Ref s1
┃     ┗ 1: Literal 42
┗ select: Select first
  ┗ in: Ref s2
```

Types T are "lifted" into Rep[T]

Lifted Embedding

Query language is "embedded" in Scala

Lifted Representation Rep[T]

```
/** Common base trait for all lifted values. */
trait Rep[T] {
  /** Get the Node for this Rep. */
  def toNode: Node

  /** Encode a reference into this Rep. */
  def encodeRef(path: Node): Rep[T]
}

object Rep {
  def apply[T](n: Node): Rep[T] = new Rep[T] {
    def toNode = n
    def encodeRef(path: Node): Rep[T] = apply(path)
  }
}
```

v.encodeRef(path)
.toNode == path

Lifted Representation Rep[T]

```
/** Common base trait for all lifted values. */
trait Rep[T, R <: Rep[T, R]] {
  /** Get the Node for this Rep. */
  def toNode: Node

  /** Encode a reference into this Rep. */
  def encodeRef(path: Node): R
}
```

Enforced in a different way to keep types simple

Literal Primitive Values

```scala
/** A lifted literal value. */
final case class LiteralRep[T : TypedType](value: T) extends Rep[T] {
  val toNode = LiteralNode(value)
  def encodeRef(n: Node) = Rep(n)
}

final class TypedType[T]

object TypedType {
  implicit val booleanType = new TypedType[Boolean]
  implicit val intType     = new TypedType[Int]
  implicit val stringType  = new TypedType[String]
}
```

Extension Methods

```
implicit class ColumnExtensionMethods[T : TypedType]
  (private val n: Rep[T]) {

  def < (e: Rep[T]) =
    Rep[Boolean](Apply(Symbol("<"), Vector(n.toNode, e.toNode)))

  def === (e: Rep[T]) =
    Rep[Boolean](Apply(Symbol("=="), Vector(n.toNode, e.toNode)))
}
```

Tables

```
abstract class Table[T](val tableTag: Tag, val tableName: String)
extends Rep[T] {
  def column[C : TypedType](n: String) = Rep[C](Select(toNode, Symbol(n)))
  // ...
}


// Example:
class Users(tag: Tag) extends Table[(Int, String, String)](tag, "users") {
  def id    = column[Int]("id")
  def first = column[String]("first")
  def last  = column[String]("last")
  def * = (id, first, last)
}
```

Building Concrete Table Instances

```
abstract class Table[T](val tableTag: Tag, val tableName: String)
extends Rep[T] {
  def column[C : TypedType](n: String) = Rep[C](Select(toNode, Symbol(n)))

  def toNode = tableTag.toNode(TableNode(tableName))
  def encodeRef(path: Node) = tableTag.encodeRef(path).asInstanceOf[Table[T]]
  // ...
}

class Tag(cons: Tag => Table[_]) {
  def encodeRef(path: Node): Table[_] = cons(new Tag(cons) {
    override def toNode(n: Node): Node = path
  })
  def toNode(n: Node): Node = n
}
```

Naive Tuple Encoding (ScalaQuery)

```
implicit class AnyRepExtensionMethods[T1 : TypedType]
  (private val v1: Rep[T1]) {

  def ~ [T2](v2: Rep[T2]) = RepTuple2[T1, T2](v1, v2)
}

case class RepTuple2[T1 : TypedType, T2 : TypedType]
  (v1: Rep[T1], v2: Rep[T2]) extends Rep[(T1, T2)] {

  def toNode: Node = ProductNode(Vector(v1.toNode, v2.toNode))
  def encodeRef(path: Node) = new RepTuple2(v1, v2) {
    override def toNode = path
  }

  def ~ [T3 : TypedType](v3: Rep[T3]) = RepTuple3[T1, T2, T3](v1, v2, v3)
}
```

Naive Tuple Encoding (ScalaQuery)

```
/* What we get: */
users.filter(_.id < 42).map(u => u.id ~ u.first ~ u.last)

/* What we want: */
users.filter(_.id < 42).map(u => (u.id, u.first, u.last))

users.filter(_.id < 42).map(u => (u, u.first, u.last))

users.filter(_.id < 42).map(u => (u.id, (u.first, u.last)))

users.filter(_.id < 42).map(u => u.id :: u.first :: u.last :: HNil)
```

Not a Rep[T]

Abstract over
element types

Polymorphic Record Types

Fixed number of
elements with
known type

Polymorphic Record Types

- Tuples

```
(Int, String, String)
(Rep[Int], Rep[String], Rep[String])
(Int, Rep[String], Users)
```

- Other Product-like Types (isomorphic to tuples)

```
class Pair[T1, T2](val v1: T1, val v2: T2)
```

- HList Types (isomorphic to nested tuples)

```
Int :: String :: HNil
Rep[Int] :: Rep[String] :: HNil
```

Functional Dependencies

between type parameters

Functional Dependencies: Example

```
class Convert[From, To](val f: From => To)
object Convert {
  implicit val intToLong    = new Convert[Int,    Long  ](_.toLong)
  implicit val longToString = new Convert[Long,   String](_.toString)
  implicit val stringToInt  = new Convert[String, Int   ](_.toInt)
}

def f[T1, T2](v: T1)(implicit conv: Convert[T1, T2]): T2 = conv.f(v)

val l: Long    = f(42)
val s: String  = f(l)
val i: Int     = f(s)
```

Functional Dependencies: Example

```
class Convert[From, To](val f: From => To)
object Convert {
  implicit val intToLong    = new Convert[Int,    Long  ](_.toLong)
  implicit val longToString = new Convert[Long,   String](_.toString)
  implicit val stringToInt  = new Convert[String, Int   ](_.toInt)
}

def f[T1, T2](v: T1)(implicit conv: Convert[T1, T2]): T2 = conv.f(v)

val l          = f(42)
val s          = f(l)
val i          = f(s)
```

Type-level function

CanBuildFrom

- Scala 2.8 collections redesign added CanBuildFrom

```
trait CanBuildFrom[-From, -Elem, +To]

trait TraversableLike[+A, +Repr] ... {

  def map[B, That](f: A => B)
    (implicit bf: CanBuildFrom[Repr, B, That]): That = ...
}
```

- Functional Dependencies were added in Scala 2.8 to enable this
- CanBuildFrom allows reuse of collection operation implementations

Shapes

Shapes

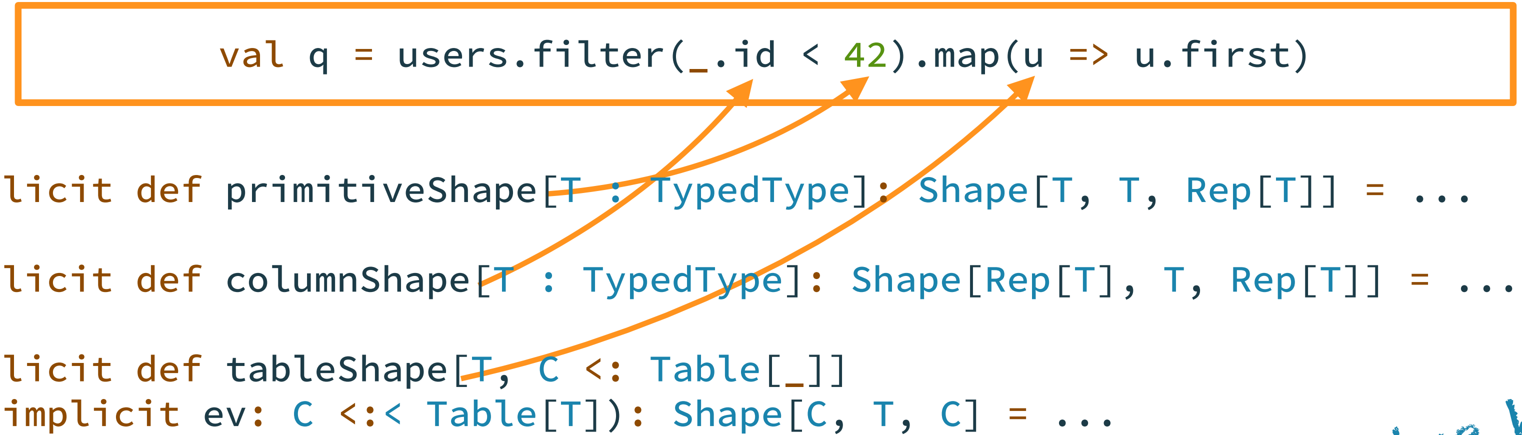
- Every value / expression in the Lifted Embedding has a Shape

```
trait Shape[-Mixed, Unpacked, Packed] { ... }
```

- Instead of hardcoding "Rep[T] produces a value of type T"
- Lookup is done by Mixed type
- Unpacked is the plain Scala type (e.g. for result values)
- The Packed type "has Reps everywhere"

Primitive Shapes

```
trait Shape[-Mixed, Unpacked, Packed] { ... }
```



```
val q = users.filter(_.id < 42).map(u => u.first)
```

```
implicit def primitiveShape[T : TypedType]: Shape[T, T, Rep[T]] = ...

implicit def columnShape[T : TypedType]: Shape[Rep[T], T, Rep[T]] = ...

implicit def tableShape[T, C <: Table[_]]
  (implicit ev: C <:< Table[T]): Shape[C, T, C] = ...
```

Every value has a shape!

Tuple Shapes

```
users.map(u => (u.first, 42) )

implicitly[Shape[(Rep[String], Int), _, _]]
```

```
implicit def tuple2Shape[M1,M2, U1,U2, P1,P2]
  (implicit u1: Shape[M1, U1, P1],
            u2: Shape[M2, U2, P2]):
    Shape[(M1,M2), (U1,U2), (P1,P2)] = ...
```

Generated for all arities

```
implicit def primitiveShape[T : TypedType]: Shape[T, T, Rep[T]] = ...

implicit def columnShape[T : TypedType]: Shape[Rep[T], T, Rep[T]] = ...
```

Nested Tuple Shapes

```
users.map(u => (u.first, (u.id, 42)) )

implicitly[Shape[(Rep[String], (Rep[Int], Int)), _, _]]
```

```
implicit def tuple2Shape[M1,M2, U1,U2, P1,P2]
  (implicit u1: Shape[M1, U1, P1],
            u2: Shape[M2, U2, P2]):
    Shape[(M1,M2), (U1,U2), (P1,P2)] = ...
```

Generated for all arities

```
implicit def primitiveShape[T : TypedType]: Shape[T, T, Rep[T]] = ...

implicit def columnShape[T : TypedType]: Shape[Rep[T], T, Rep[T]] = ...
```

Shape Implementations

```
trait Shape[-Mixed, Unpacked, Packed] {

  def toNode(value: Mixed): Node

  def encodeRef(value: Mixed, path: Node): Any

  def pack(value: Mixed): Packed

  def packedShape: Shape[Packed, Unpacked, Packed]
}
```

Shape Implementations

```
implicit def columnShape[T : TypedType] =
  repShape[Rep[T], T]

implicit def tableShape[T, C <: Table[_]](implicit ev: C <:< Table[T]) =
  repShape[C, T]

def repShape[MP <: Rep[_], U]: Shape[MP, U, MP] = new Shape[MP, U, MP] {
  def toNode(value: MP) = value.toNode
  def encodeRef(value: MP, path: Node) = value.encodeRef(path)
  def pack(value: MP) = value
  def packedShape = this
}
```

Shape Implementations

```
implicit def primitiveShape[T : TypedType]: Shape[T, T, Rep[T]] =
  new Shape[T, T, Rep[T]] {
    def pack(value: T) = LiteralRep(value)
    def packedShape = repShape[Rep[T], T]
    def toNode(value: T): Node = pack(value).toNode
    def encodeRef(value: T, path: Node) =
      throw new RuntimeException(
        "Shape does not have the same Mixed and Packed type")
  }
```

toNode(encodeRef(v, path)) == path

Queries

Queries

```
final class Query[+E, U](val toNode: Node,
                         val shaped: ShapedValue[_ <: E, U])
extends Rep[Seq[U]] {

  def encodeRef(path: Node): Query[E, U] = new Query[E, U](path, shaped)

  // ...
}

object TableQuery {
  def apply[C, E <: Table[_]](cons: Tag => E)
    (implicit ev: E <:< Table[C]): Query[E, C] = {
    val shaped = ShapedValue(cons(new Tag(cons)), Shape.repShape[E, C])
    new Query[E, C](shaped.toNode, shaped)
  }
}
```

Queries: Filter

```
final class Query[+E, U](val toNode: Node,
                         val shaped: ShapedValue[_ <: E, U])
extends Rep[Seq[U]] { // ...

  def filter(f: E => Rep[Boolean]): Query[E, U] = {
    val s = Symbol.fresh
    val fv = f(shaped.encodeRef(Ref(s)).value)
    new Query[E, U](Filter(s, toNode, fv.toNode),
                    shaped)
  }
}
```

```
users.filter(u => u.id < 42)
```

```
Filter
┣ from s9: Table users
┗ where: Apply <
  ┣ 0: Select id
  ┃ ┗ in: Ref s9
  ┗ 1: Literal 42
```

Queries: Naive Map

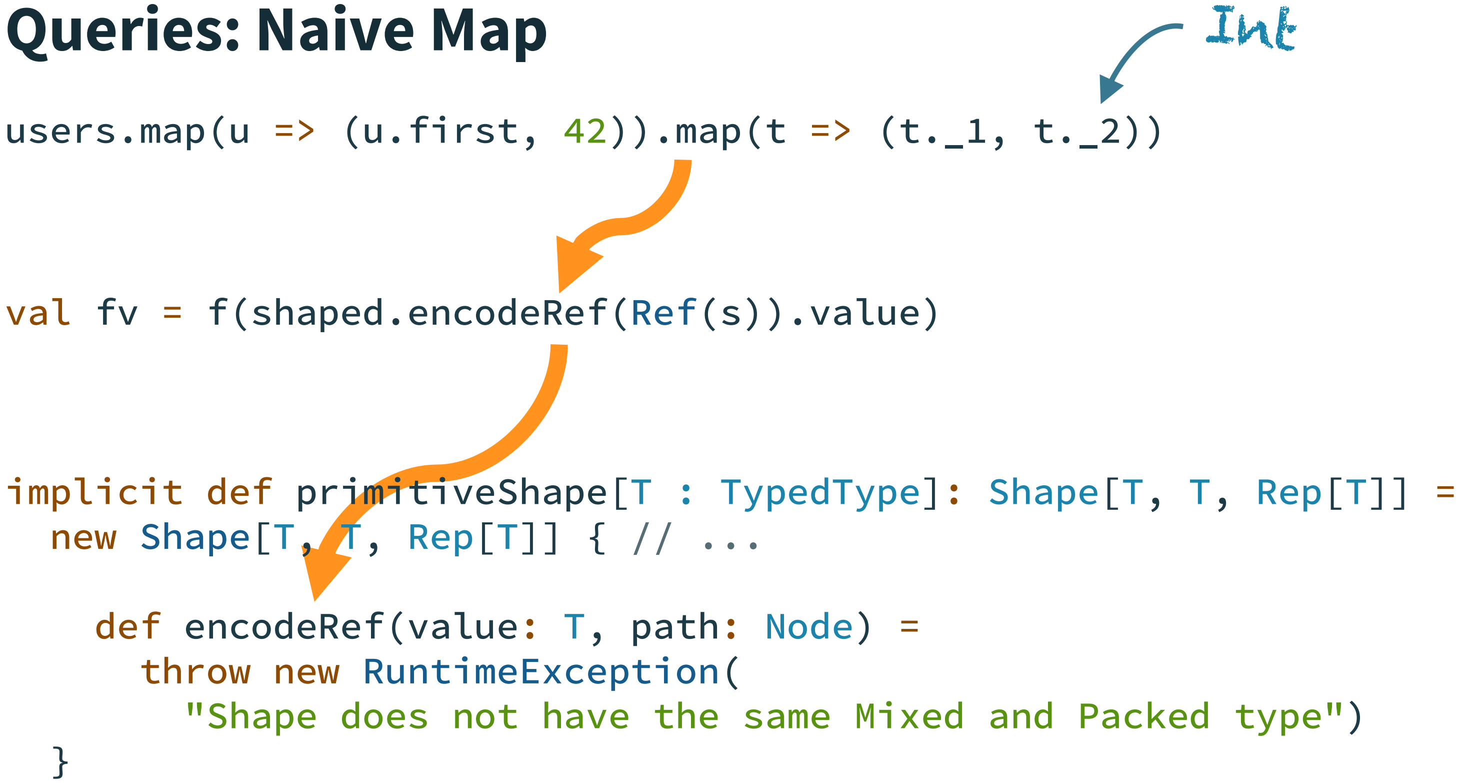
```scala
final class Query[+E, U](val toNode: Node,
                         val shaped: ShapedValue[_ <: E, U])
extends Rep[Seq[U]] { // ...

  def map[F, T](f: E => F)(implicit shape: Shape[F, T, _]): Query[F, T] = {
    val s = Symbol.fresh
    val fv = f(shaped.encodeRef(Ref(s)).value)
    val sv = ShapedValue(fv, shape)
    new Query[F, T](
      new MapNode(s, toNode, sv.toNode),
      sv)
  }
}
```

```scala
users.map(u => (u.first, 42))
```

```
Map
┣ from s8: Table users
┗ select: Product
  ┣ 1: Select first
  ┃ ┗ in: Ref s8
  ┗ 2: Literal 42
```

Queries: Naive Map



```
users.map(u => (u.first, 42)).map(t => (t._1, t._2))


val fv = f(shaped.encodeRef(Ref(s)).value)


implicit def primitiveShape[T : TypedType]: Shape[T, T, Rep[T]] =
  new Shape[T, T, Rep[T]] { // ...

    def encodeRef(value: T, path: Node) =
      throw new RuntimeException(
        "Shape does not have the same Mixed and Packed type")
  }
```

Queries: Map

```
final class Query[+E, U](val toNode: Node,
                         val shaped: ShapedValue[_ <: E, U])
extends Rep[Seq[U]] { // ...

  def map[F, G, T](f: E => F)(implicit shape: Shape[F, T, G]): Query[G, T] = {
    val s = Symbol.fresh
    val fv = f(shaped.encodeRef(Ref(s)).value)
    val packed = ShapedValue(fv, shape).packedValue
    new Query[G, T](
      new MapNode(s, toNode, packed.toNode),
      packed)
  }
}
```

```
users.map(u => (u.first, 42))
```

```
Map
┣ from s8: Table users
┗ select: Product
  ┣ 1: Select first
  ┃ ┗ in: Ref s8
  ┗ 2: Literal 42
```

Queries: Map

Rep[Int]

```
users.map(u => (u.first, 42)).map(t => (t._1, t._2))
```

Shape Implementations

```
abstract class ProductNodeShape[C, M <: C, U <: C, P <: C] extends
Shape[M, U, P] {
  // ...

  def toNode(value: M): Node =
    ProductNode(shapes.iterator.zip(getIterator(value)).map {
      case (p, f) => p.asInstanceOf[Shape[Any, Any, Any]].toNode(f)
    }.toVector)

  def encodeRef(value: M, path: Node) =
    buildValue(shapes.iterator.zip(getIterator(value)).zipWithIndex.map {
      case ((p, x), pos) =>
        p.asInstanceOf[Shape[Any, Any, Any]].encodeRef(x,
          Select(path, Symbol("_" + (pos+1))))
    }.toIndexedSeq)
}
```

Base class for
Tuple shapes

Heterogeneous Lists (HLists)

HLists

```
sealed abstract class HList

final object HNil extends HList

final class HCons[+H, +T <: HList](val head: H, val tail: T) extends HList
```

```
val l: HCons[Int, HCons[String, HNil.type]]
val l: Int :: String :: HNil
```

HList Shapes

```
final class HListShape[M <: HList, U <: HList, P <: HList]
  (val shapes: Seq[Shape[_, _, _]])
extends MappedScalaProductShape[HList, M, U, P] {
  def buildValue(elems: IndexedSeq[Any]) =
    elems.foldRight(HNil: HList)(_ :: _)
  def copy(shapes: Seq[Shape[_, _, _]]) = new HListShape(shapes)
}

implicit val hnilShape =
  new HListShape[HNil.type, HNil.type, HNil.type](Nil)

implicit def hconsShape
  [M1, M2 <: HList, U1, U2 <: HList, P1, P2 <: HList]
  (implicit s1: Shape[M1, U1, P1], s2: HListShape[M2, U2, P2]) =
    new HListShape[M1 :: M2, U1 :: U2, P1 :: P2](s1 +: s2.shapes)
```

HList Shapes

```
class Users(tag: Tag) extends Table[Int :: String :: String :: HNil]
  (tag, "users") {
  def id    = column[Int]("id")
  def first = column[String]("first")
  def last  = column[String]("last")
  def * = id :: first :: last :: HNil
}

lazy val users = TableQuery(new Users(_))

val q1 = users.map(u => u.id :: u.first :: HNil)
```

Works everywhere in the Lifted Embedding!

Links

- Slick: http://slick.lightbend.com
- Toy Slick: https://github.com/szeiger/slick/tree/toy-slick-scaladays2016

- Follow me: 🐦 @StefanZeiger

SCALA DAYS
NEW YORK