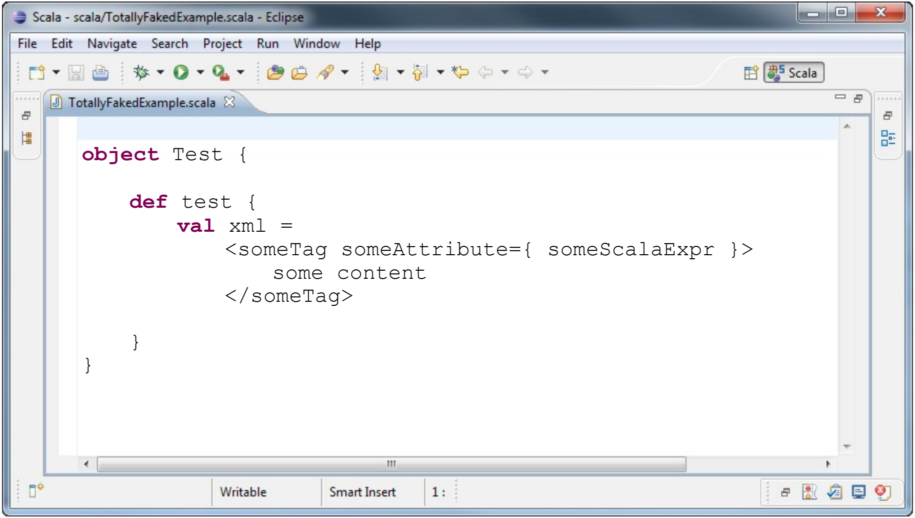
Type-safe SQL embedded in Scala

Christoph Wulf

University of Kiel

cwu@informatik.uni-kiel.de

Embedded XML in Scala



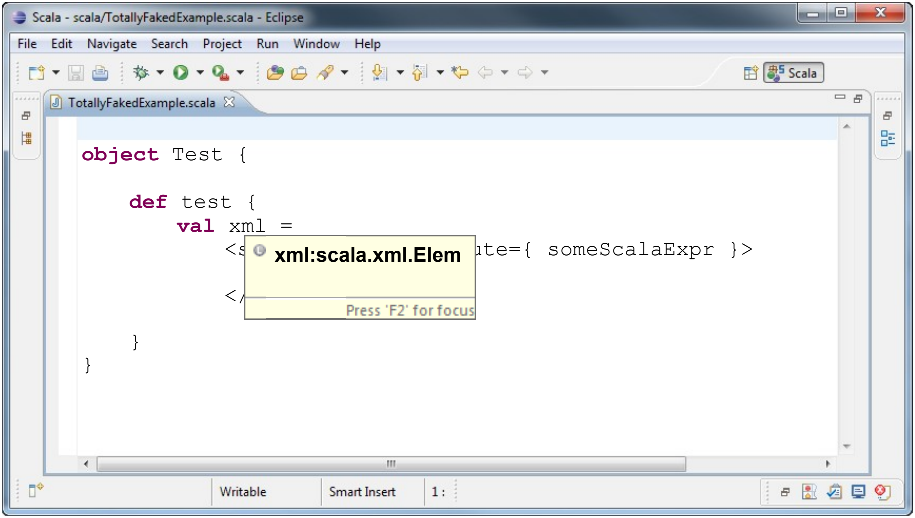
Scala - scala/TotallyFakedExample.scala - Eclipse
File Edit Navigate Search Project Run Window Help
Scala
TotallyFakedExample.scala
Writable
Smart Insert
1 :

```
object Test {

    def test {
        val xml =
            <someTag someAttribute={ someScalaExpr }>
                some content
            </someTag>

    }
}
```

Embedded XML in Scala



Scala - scala/TotallyFakedExample.scala - Eclipse
File Edit Navigate Search Project Run Window Help
Scala
TotallyFakedExample.scala
Writable
Smart Insert
1 :

```
object Test {

    def test {
        val xml =
            <s                     ute={ someScalaExpr }>

            </

    }
}
```

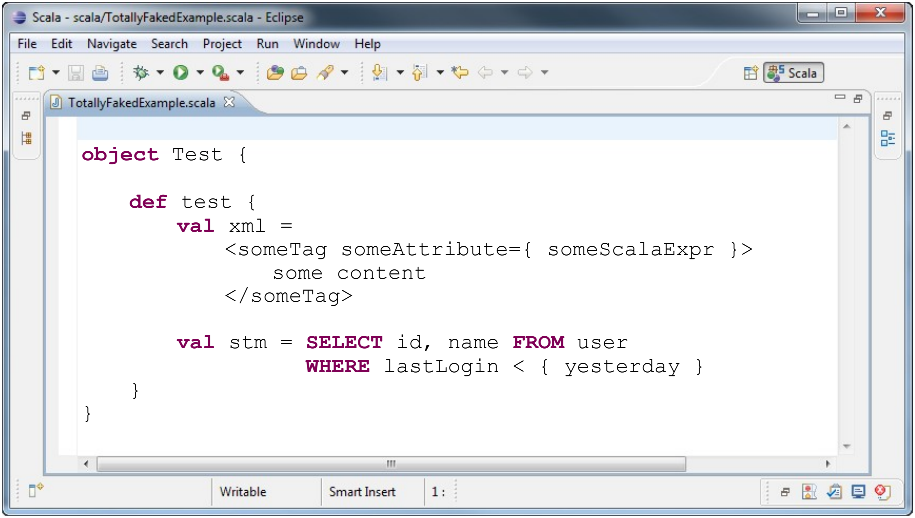
xml:scala.xml.Elem

Press 'F2' for focus

Embedded SQL in Scala

If XML qualifies to be embedded into Scala,
why not also SQL?

Embedded SQL in Scala



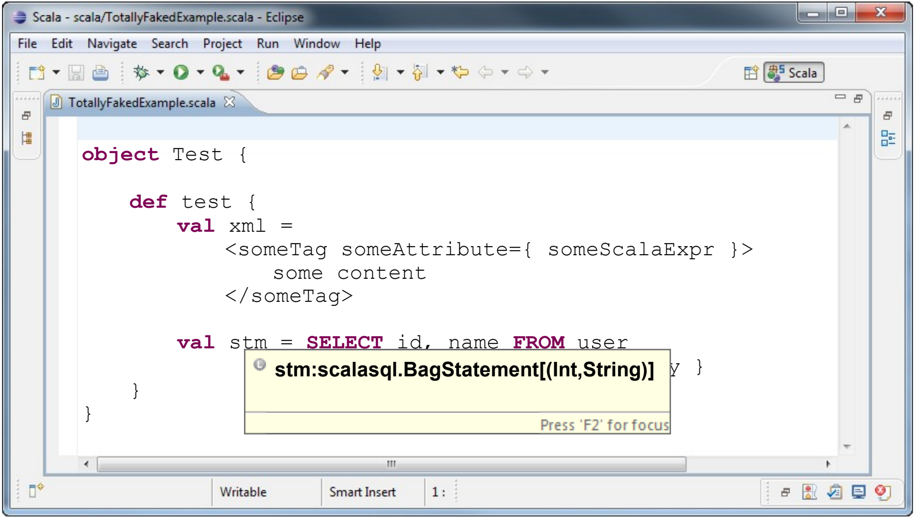
Scala - scala/TotallyFakedExample.scala - Eclipse
File Edit Navigate Search Project Run Window Help
Scala
TotallyFakedExample.scala
Writable
Smart Insert
1 :

```
object Test {

    def test {
        val xml =
            <someTag someAttribute={ someScalaExpr }>
                some content
            </someTag>

        val stm = SELECT id, name FROM user
                  WHERE lastLogin < { yesterday }
    }
}
```

Embedded SQL in Scala


Scala - scala/TotallyFakedExample.scala - Eclipse
File
Edit
Navigate
Search
Project
Run
Window
Help
Scala
TotallyFakedExample.scala
Writable
Smart Insert
1 :

```
object Test {

    def test {
        val xml =
            <someTag someAttribute={ someScalaExpr }>
                some content
            </someTag>

        val stm = SELECT id, name FROM user
                                          y }
    }
}
```

stm:scalasql.BagStatement[(Int,String)]

Press 'F2' for focus

Motivation

```
val sql = "SELEC id, name FROM user WHERE lastLogin < ?"
val stm = conn.prepareStatement(sql)
stm.setDate(1, someDate)
val rs = stm.executeQuery
while (rs.next) {
  handleUser(rs.getInt(1), rs.getString(2))
}
```

- For the compiler the query is just a `String`
- Compile time of the query is the runtime of the program
  - Parse errors are not detected
  - No type-safety for the query, parameters and the result set processing

Motivation

```
val sql = "SELEC id, name FROM user WHERE lastLogin < ?"
val stm = conn.prepareStatement(sql)
stm.setDate(1, someDate)
val rs = stm.executeQuery
while (rs.next) {
  handleUser(rs.getInt(1), rs.getString(2))
}
```

- For the compiler the query is just a `String`
- Compile time of the query is the runtime of the program
  - Parse errors are not detected
  - No type-safety for the query, parameters and the result set processing

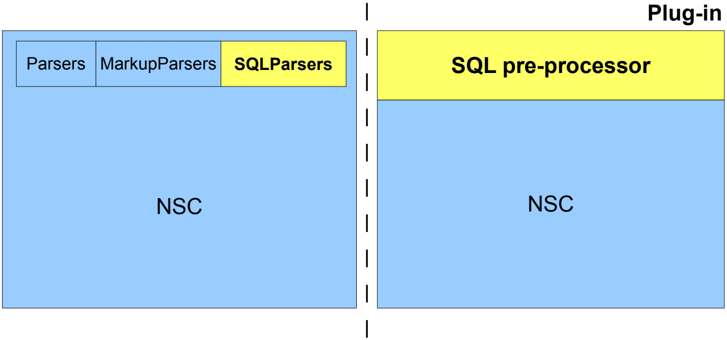
Related Work

Persistence Frameworks in Scala

- JPA / Hibernate and other tools inherited from Java
- Lift Persistence Framework
- ScalaQL
- Several type-safe DSLs for JDBC in Scala

Design

- Similar to embedded XML: embedded statements without special delimiters
- Processing of embedded SQL during the lexical phase (or a pre-processor)
- Minor changes to the NSC (plug-in if possible)



Type-safety in two ways

Correct application of SQL functions and operators

```
val stm = SELECT 1 + name, someDate < { yesterday }
          FROM user
          WHERE CONCAT(5)
```

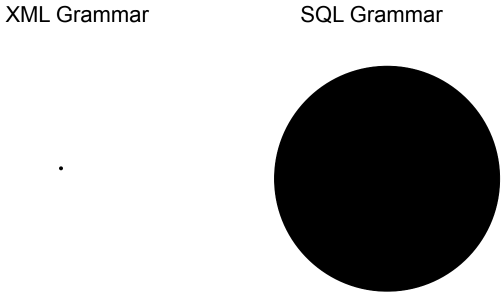
Result set type-inference ⇒ result processing without casting

```
val stm = SELECT id, name FROM user
// Type inference => stm : BagStatement[(Int,String)]

val iter = stm.execute(sqlConnection)
// inferred to:
// iter:Iterator[(Int,String)]
```

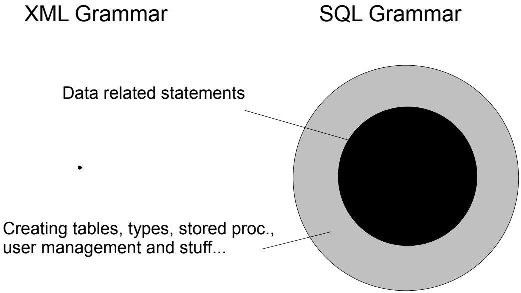
Parsing embedded statements

- Extending Scala's grammar by SQL statements



Parsing embedded statements

- Extending Scala's grammar by SQL statements



XML Grammar

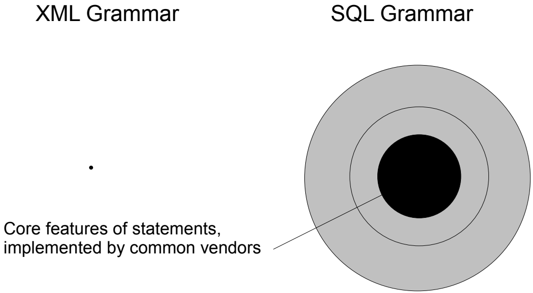
SQL Grammar

Data related statements

Creating tables, types, stored proc.,
user management and stuff...

Parsing embedded statements

- Extending Scala's grammar by SQL statements



Parsing embedded statements

- Extending Scala's grammar by SQL statements
- Should every SQL keyword be a Scala keyword, too?
  ⇝ `select`, `insert`, `delete`, `Call`, `as`, `by`, **`WITH`** not allowed
- SQL keywords only in upper case letters
- SQL keywords as Scala identifiers, only as keywords in the SQL parser
- Statement leading keywords as *weak keywords*

Parsing embedded statements

- Extending Scala's grammar by SQL statements
- Should every SQL keyword be a Scala keyword, too?
  ⇝ `select`, `insert`, `delete`, `Call`, `as`, `by`, **`WITH`** not allowed
- SQL keywords only in upper case letters
- SQL keywords as Scala identifiers, only as keywords in the SQL parser
- Statement leading keywords as *weak keywords*
- Try to parse SQL, fall back in case of parse error and parse as Scala
  - **`SELECT`**
  - **`WITH`** (for recursive queries)
  - **`INSERT`**
  - **`UPDATE`**
  - **`DELETE`**

Parsing embedded statements

- Extending Scala's grammar by SQL statements
- Should every SQL keyword be a Scala keyword, too?

  ⇝ `select`, `insert`, `delete`, `Call`, `as`, `by`, **`WITH`** not allowed
- SQL keywords only in upper case letters
- SQL keywords as Scala identifiers, only as keywords in the SQL parser
- Statement leading keywords as *weak keywords*
- Try to parse SQL, fall back in case of parse error and parse as Scala
  - **`SELECT`**
  - **`WITH`** (for recursive queries)
  - **`INSERT`**
  - **`UPDATE`**
  - **`DELETE`**

Parsing embedded statements

- Extending Scala's grammar by SQL statements
- Should every SQL keyword be a Scala keyword, too?

  ⇝ `select`, `insert`, `delete`, `Call`, `as`, `by`, **WITH** not allowed
- SQL keywords only in upper case letters
- SQL keywords as Scala identifiers, only as keywords in the SQL parser
- Statement leading keywords as *weak keywords*
- Try to parse SQL, fall back in case of parse error and parse as Scala
  - **SELECT** … **FROM** …
  - **WITH** … **AS** …
  - **INSERT** … **INTO** …
  - **UPDATE** … **SET** …
  - **DELETE** … **FROM** …

Result processing

- UpdateStatement
  - **INSERT**, **UPDATE** and **DELETE**
- BagStatement
  - **SELECT** and **WITH**
  - RowStatement (for queries returning at most one row)

Result processing

- `UpdateStatement`
  - **INSERT**, **UPDATE** and **DELETE**
- `BagStatement`
  - **SELECT** and **WITH**
  - `RowStatement` (for queries returning at most one row)

Result processing

- UpdateStatement **extends** Statement[Int]
    - **INSERT**, **UPDATE** and **DELETE**
- BagStatement[T] **extends** Statement[Iterator[T]]
    - **SELECT** and **WITH**
    - RowStatement[T] **extends** Statement[Option[T]]

Result processing

- UpdateStatement **extends** Statement[Int]
  - **INSERT**, **UPDATE** and **DELETE**
- BagStatement[T] **extends** Statement[Iterator[T]]
  - **SELECT** and **WITH**
  - RowStatement[T] **extends** Statement[Option[T]]

```
abstract class Statement[T] {
  def execute(connection : java.sql.Connection) : T
  def >>(implicit conn : java.sql.Connection) = execute(conn)
}
```

Result processing

- UpdateStatement **extends** Statement[Int]
  - **INSERT**, **UPDATE** and **DELETE**
- BagStatement[T] **extends** Statement[Iterator[T]]
  - **SELECT** and **WITH**
  - RowStatement[T] **extends** Statement[Option[T]]

```
abstract class Statement[T] {
  def execute(connection : java.sql.Connection) : T
  def >>(implicit conn : java.sql.Connection) = execute(conn)
}
```

- Operator >> to execute the statement for an (implicit) connection

```
val iter = SELECT id, name FROM user >> sqlConnection

val iter = SELECT id, name FROM user >>
```

Result processing

- Operator >>> for direct result processing
- Defined in each statement class to take an action and an implicit connection
  - UpdateStatement: partial action is applied to result number
  - BagStatement: partial action is applied to each row value
  - RowStatement: partial action is applied to result row value (if exists)

```
SELECT name, lastLogin FROM user >>> {
  case (name,lastLogin) => out write <tr>
                                       <td>{ name }</td>
                                       <td>{ lastLogin.toLocaleString }</td>
                                     </tr>
}
```

Result processing

- Operator >>> for direct result processing
- Defined in each statement class to take an action and an implicit connection
    - UpdateStatement: partial action is applied to result number
    - BagStatement: partial action is applied to each row value
    - RowStatement: partial action is applied to result row value (if exists)

```
SELECT name, lastLogin FROM user >>> {
  case (name,lastLogin) => out write <tr>
                                       <td>{ name }</td>
                                       <td>{ lastLogin.toLocaleString }</td>
                                     </tr>
}
```

Implementation

Processing of embedded statements is done in the lexical phase (or before)

⇒ Compiled statement has to be independent from its environment (especially from the database schema)

⇒ Type-safety has to be ensured by later phases

Implementation

Processing of embedded statements is done in the lexical phase (or before)

⇒ Compiled statement has to be independent from its environment (especially from the database schema)

⇒ Type-safety has to be ensured by later phases

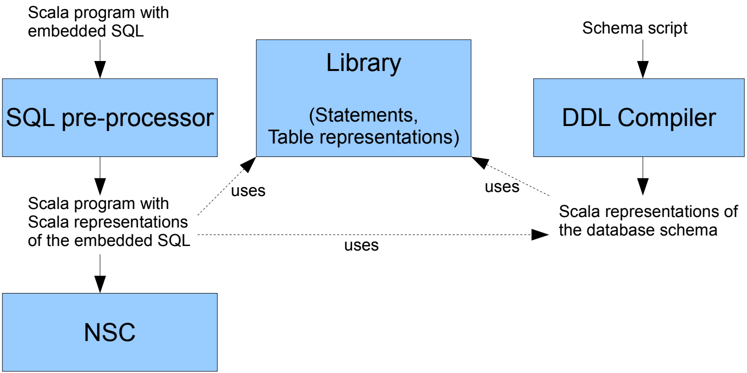

Scala program with embedded SQL
SQL pre-processor
Scala program with Scala representations of the embedded SQL
NSC
Library
(Statements, Table representations)
uses
uses
uses
Schema script
DDL Compiler
Scala representations of the database schema

Table representation

For each SQL type a Scala trait

- character (varying) ⇒ `StringExpr`
- bit / bool ⇒ `BoolExpr`
- integer ⇒ `IntExpr`

Table representation

For each SQL type a Scala trait

- character (varying) ⇒ `StringExpr`
- bit / bool ⇒ `BoolExpr`
- integer ⇒ `IntExpr`

```
trait IntExpr extends Expr[Int] {
  def <(that:IntExpr) = new BinOp(this,"<",that) with BoolExpr
  def >(that:IntExpr) = new BinOp(this,">",that) with BoolExpr
  ...
  def +(that:IntExpr) = new BinOp(this,"+",that) with IntExpr
  ...
  def /(that:IntExpr) = new BinOp(this,"/",that) with FloatExpr
  def ===(that:IntExpr)= new BinOp(this,"=",that) with BoolExpr
  def <>(that:IntExpr) = new BinOp(this,"<>",that) with BoolExpr
  ...
  def extract = i => rs => rs.getInt(i)
  def AS(alias:String) = new AliasColumn(this, alias) with IntColumn
}
```

Table representation

```
CREATE TABLE user(
  id INTEGER NOT NULL,
  name VARCHAR(50) NOT NULL,
  lastLogin DATETIME NOT NULL,
  ...
  PRIMARY KEY (id)
)
```

⬇

```
class user(qualifier:String) extends Table(qualifier,"user") {
  def id = new Column(qualifier,"id") with IntExpr
  def name = new Column(qualifier,"name") with StringExpr
  def lastLogin = new Column(qualifier,"lastLogin") with DateExpr
  ...
  def * = id ~ login ~ lastLogin ~ ...
  ...
}
```

Compiled query

```
val stm = SELECT u.id + 1, CONCAT('Mr.', name)
          FROM user u
          WHERE u.male AND lastLogin < { yesterday }
```

⬇

```
val stm = new QueryExpression {
  val u = new user("u")
  import u
  import scalasql.functions._
  def select = u.id + c(1) ~ CONCAT(c('Mr.'), name)
  def from = u
  override def where = u.male AND lastLogin < c(yesterday)
}
```

Compiled query

```
val stm = SELECT u.id + 1, CONCAT('Mr.', name)
          FROM user u
          WHERE u.male AND lastLogin < { yesterday }
```

⬇

```
val stm = new QueryExpression {
  val u = new user("u")
  import u
  import scalasql.functions._
  def select = u.id + c(1) ~ CONCAT(c('Mr.'), name)
  def from = u
  override def where = u.male AND lastLogin < c(yesterday)
}

trait QueryExpression {
  def select : Projection[_]
  def from : TableReference
  def where : BoolExpr = c(true)
  ...
}
```

Result set type inference

Until now everything could be realized in Java, too!
(except: operators, tuples, traits, local imports, ...)

Scalas type inference systems takes affect at the way from `QueryExpression` to a generic `BagStatement[(Int,String)]`

```
val optionalInt = Some[Int](26)
val optionalInt = Some(26)

final case class Some[+A](x: A) extends Option[A] {
  def isEmpty = false
  def get = x
}
```

Result set type inference

Until now everything could be realized in Java, too!
(except: operators, tuples, traits, local imports, ...)

Scalas type inference systems takes affect at the way from `QueryExpression` to a generic `BagStatement[(Int,String)]`

```
val optionalInt = Some[Int](26)
val optionalInt = Some(26)

final case class Some[+A](x: A) extends Option[A] {
  def isEmpty = false
  def get = x
}
```

Result set type inference

```
trait QueryExpression[A] {
  def select : Projection[A]
  def from : TableReference
  def where : BoolExpr = c(true)
  ...
}
```

Result set type inference

```
trait QueryExpression[A] {
  def select : Projection[A]
  def from : TableReference
  def where : BoolExpr = c(true)
  ...
}
```

Getting a generic result set requires a small hack:

```
class BagStatement[A](q:QueryExpression, p:Projection[A]) { ... }
```

Result set type inference

```
trait QueryExpression[A] {
  def select : Projection[A]
  def from : TableReference
  def where : BoolExpr = c(true)
  ...
}
```

Getting a generic result set requires a small hack:

```
class BagStatement[A](q:QueryExpression, p:Projection[A]) { ... }

val stm = {
  new QueryExpression {
    val u = new user("u")
    import u
    import scalasql.functions._
    def select = u.id + c(1) ~ CONCAT(c(’Mr.’), name)
    def from = u
    override def where = u.male AND lastLogin < c(yesterday)
  }
  new BagStatement(query,query.select)
}
```

Further re-writings

The paper covers compilation of further, more complex statements including

- Joins
- Aggregation
- Recursive queries
- Null values
- Set operations (**UNION**, **INTERSECT**, **EXCEPT**, IN, **ALL**)
- Modifying statements (**INSERT**, **UPDATE**, **DELETE**)

Object-relational features

- Tuples are best representations for bulk queries selecting only parts of table columns, but objects are often required
- Usage of $*$ operator returns objects instead of tuples
- **data** classes as derivation of **case** classes
- No need for another ORM tool
  $\Rightarrow$ Possible integration with Lift persistence
  - Queries select from Lift records/mappers
  - $*$ projection returns Lift objects
  - Embedded queries as an alternative option for bulk queries in Lift applications

Conclusion

- SQL statements directly embedded in Scala
- Type-safe in queries and result processing
- No additional compiler for programs using embedded SQL
- First implementation for simple queries almost developed
- I am looking forward to provide a full implementation with my master thesis!