FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Scala library for constructing statically typed PostgreSQL queries
Student:	Petr Hron
Supervisor:	Ing. Vojtěch Létal
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

- Get familiar with the implementation of query parser in PostgreSQL
- Do a research of the existing libraries used to construct SQL queries in Scala programming language. Examine their pros and cons and see how your solution could fit into the existing library ecosystem.
- Design and implement a library which would attempt to support construction and composition of statically typed PostgreSQL queries in Scala programming language.
- To test the library create a unit test suite, as well as implement an example application using the library.
- Make sure that your library is available in a form of a public repository with a working CI pipeline, contribution guidelines, etc.
- Discuss your results.

Electronically approved by Ing. Michal Valenta, Ph.D. on 2 March 2021 in Prague.

Bachelor's thesis

Scala library for constructing statically typed PostgreSQL queries

Petr Hron

Department of software engineering
Supervisor: Ing. Vojtěch Létal

June 20, 2021

Acknowledgements

THANKS (remove entirely in case you do not with to thank anyone)

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 20, 2021 ....

Citation of this thesis

Hron, Petr. *Scala library for constructing statically typed PostgreSQL queries.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Scala, PostgreSQL, syntaktický strom, open source, validace během kompilace

Abstract

Summarize the contents and contribution of your work in a few sentences in English language.

Keywords Scala, PostgreSQL, parse tree, open source, compile time validation

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation and goals

Main goal of my work is to create Scala specific library to support constructing statically typed queries for Scala. This includes validation of queries and accessing internal PostgreSQL parse tree, which can be helpful for multiple different reasons....

Chapter 2

Technologies used

2.1 Scala

2.1.1 Introduction

Scala belongs to the group of programming languages that can be compiled into Java byte code and run on a Java virtual machine (JVM). The major part, which makes it different from well-known Java, is the combination of applying a functional approach with an object-oriented paradigm. Together with the fact that Scala is similar to Java language itself, having the object-oriented style still present can ease up transition for programmers who are unfamiliar with the functional world.

2.1.2 Static typing

Besides the functional fundamentals, Scala is part of the family of statically typed languages, together with languages like C, C++, Java, or Haskell. Therefore, every single object in Scala has a type.

To make a job easier for the programmer, Scala uses a system known as type inference - automatic type detection. That allows faster coding, thanks to the fact you don't have to worry about specifying every object's type. However, providing types is always considered good practice, especially when we're writing public API. Thanks to that, people who choose to use our code know what types of objects to expect.

Since Scala compiler knows the types of the objects, it reveals many bugs during compilation. This is a great thing, because the sooner we can identify a bug, the easier it should be to fix it.

2.2 PostgreSQL

PostgreSQL is an ORDBMS - abbreviation for open source object-relational database management system.[1] Origins date back to the year 1986, where the project then known as POSTGRES started as a reference to the older INGRES database. After 10 years it got renamed to PostgreSQL to clearly show its ability to work with SQL.

Nowadays, it's used in many different projects. PostgreSQL popularity has been steadily rising in the last few years. Based on "Stack Overflow Annual Developer Survey" [2], PostgreSQL currently sits at second place for the 'Most popular technology in the database category', right after the MySQL.

2.2.1 Parse tree

Postgres internally uses parse trees for processing SQL Queries. The whole parsing comprises multiple stages. First, the passed query, which is in the form of plain text, is transformed to tokens, using Unix tool "Flex". Next up the parser generator called "Bison" is called, which consists of multiple grammar rules and actions. Each action is executed whenever any of the rules are applied and together they are used to build the final parse tree.

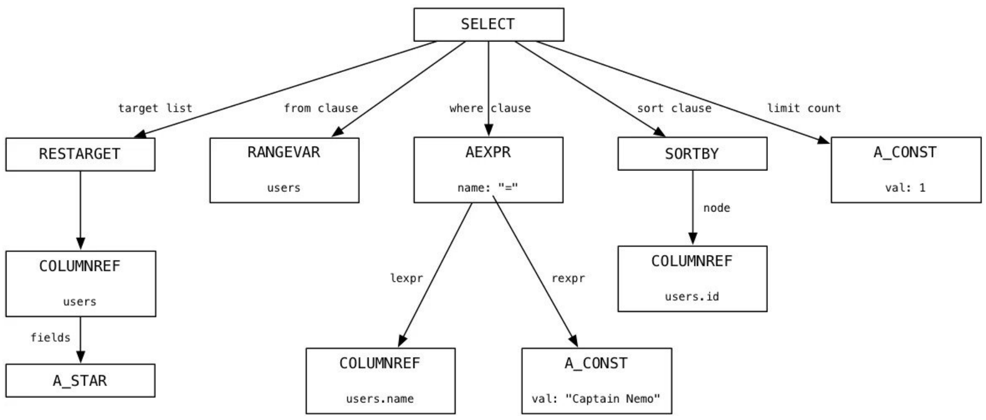


Figure 2.1: Visualisation of parse tree for "SELECT * FROM users WHERE name = 'Captain Nemo' ORDER BY id ASC LIMIT 1'

Chapter **3**

Existing options

3.1 Database libraries for Scala

When we are working with databases in Java, we are most likely using JDBC, either directly or by wrappers like JPA or Hibernate. JDBC is available in Scala as well, by simply importing the `java.sql` API, you can create connections similarly as you would do in Java. But there are multiple existing libraries made for Scala, that ensure an easier way for the programmer to work with databases. Below I describe few selected libraries that were created for that specific reason.

3.1.1 Quill

Quill provides a Quoted Domain Specific Language (QDSL). [3] Its primary usage is to generate SQL queries, using only Scala code. Doing it this way Quill also provides type-safe queries, based on validation against defined database structure. The disadvantage of this approach is that the user can't validate generic queries without creating the case class database structure beforehand.

3.1.2 Doobie

Next up there is have Doobie, which is presented as *"Doobie is a pure functional JDBC layer for Scala"*. [4] In this library you can actually create pure SQL queries. The disadvantage is you have to have existing database and establish connection with it. That only allows run-time validation of queries.

Chapter 4

Realisation

4.1 Getting the SQL parse tree

Since we want our library to work with parse trees of the SQL queries, first question we have to ask is how to get the internal parse tree itself. According to the official documentation[5] the parse tree gets created during the ”Parser stage”. Fortunately there already exists project, which extracts the parse tree called `Libpg_query`.

4.1.1 Libpg_query

`Libpg_query` is an open-source C library created by Lukas Frittl. It uses the PostgreSQL server to access the internal `raw_parse` function, which returns the internal parse tree. A minor disadvantage of this approach is that it uses the server code directly and it has to be compiled before it can be used. Then it accesses internal functions of the server which allows the library to get the parse tree for each valid query.

The main purpose of `libpg_query` is to be used as a base library for implementations in other languages. There already exist multiple wrappers, for example `pg_query` for Ruby or `pglast` for Python. However, at the moment of writing this thesis, there is no existing wrapper for it written for Scala. From `libpg_query`, I am using the `pg_query_parse` function. Thanks to this function we can get the resulting parse tree in JSON format, which will be further processed by our Scala library.

Here we have a simple code snippet describing simple usage taken from the GitHub README[6].

```
#include <pg_query.h>
#include <stdio.h>

int main() {
  PgQueryParseResult result;
  result = pg_query_parse("SELECT 1");
  printf("%s\n", result.parse_tree);
  pg_query_free_parse_result(result);
}
```

4.2 Parse tree representation in Scala

Before we can start our work with the parse tree, it will prove useful to create our own structure, to represent that data in a form we can easily work with.

The C library has its own struct representation for each type of possible Nodes that can be found in the internal PostgreSQL parse tree. That makes our job easier, because we just have to transform C structs to Scala case classes.

Original C struct in `libpg_query`

```
typedef struct A_Expr
{
    NodeTag     type;
    A_Expr_Kind kind;
    List       *name;
    Node       *lexpr;
    Node       *rexpr;
    int         location;
} A_Expr;
```

Scala case class

```
case class A_Expr(
    kind:      A_Expr_Kind.Value,
    name:      List[Node],
    lexpr:     Option[Node],
    rexpr:     Option[Node],
    location:  Option[Int]
) extends Node
```

4.3 Using native library

Now we have a seemingly simple task to do. Just call the function from the `libpg_query` library, which returns the parse tree in the form of JSON. Then we will convert the JSON into our case class structure. That is not as simple as it looks, and the reason for that is the conflict between native code and java byte code.

4.3.1 Native code and byte code

Native code is compiled to run on a specific processor. Examples of languages that produce native code after compilation are C, C++. That means, every time we want to run our C program, it has to be recompiled for that specific operating system or processor.

Java byte code, on the other hand, is compiled source code from i.e. Java, Scala. Byte code is then translated to machine code using JVM. Any system that has JVM can run the byte code, does not matter which operating system it uses. That is why Java and Scala as well, are platform-independent.

Now because of this difference, we can't directly "import" the C library into our Scala code, but we have to use workaround, that enables this.

4.3.2 Java native interface

JNI is programming interface for writing Java native methods.[7] It is used to enable Java code to use native applications and libraries.

My earlier version of the project used JNI directly. I used `javah` command to generate C header file for `PgQueryWrapper`. The JNI C header file works as bridge between native code and Scala program. Now we can call libpg_query inside the new C file, then compile it and create our shared library. Next step is loading the .so file into the Scala program. That was achieved by `System.loadLibrary` function. When that was done succesfully, we could finally receive the JSON containing the parse tree of our queries.

4.3.3 sbt-jni

Unfortunately, the preceding approach came with some problems. You had to modify the `java.library.path`, to add lib folder to it. When the project got separated into several subprojects, there was a problem with the system trying to use the shared library at the same time from different sources, which failed with `java.lang.UnsatisfiedLinkError` exception, since the native library was already loaded in different classloader.

For thses reasons, I decided to use existing JNI wrapper for Scala called `sbt-jni`. It contains plugins to help working with native libraries, to name a few, JniJavah works as wrapper around the `javah` command to generate headers for classes with `@native methods`. Next one I used is JniLoad, which enables easy loading of shared libraries through `@nativeLoader` annotation.

4.4 Parsing JSON result from libpg_query

There are few different libraries that can help with parsing JSONs. From those I decided to use *circe*. Circe is fork of the pure functional library called Argonaut. It is great for parsing, traversing JSON, but the main functionality I used are the Encoders and Decoders.

We have to generate Encoders and Decoders for every single case class, representing one of the nodes of the parse tree.

4.4.1 JSON structure

Each node is in the JSON defined as key-value pair. Key is always the name of the node and value is dictionary where keys are names of the parameters with their corresponding values.

4.4.2 How decoding works

Basic decoder for specific case class in *circe* works as follows. JSON is parsed as key-value pairs and it attempts to map each parameter in case class to corresponding key from JSON. Return value from parsing is `Decoder.Result[T]`, which translates to `Either[DecodingFailure, T]`. As the name suggests, you get either `Left(DecodingFailure)` in case any invalid operation happens during the parsing, or `Right(T)`, where `T` is the required object that is supposed to be parsed.

If the key is not found in the case class parameter list, it either sets the parameter to `None` (if the parameter is of `Option[T]` type), or returns `DecodingFailure`. If the value we are trying to further parse is not one of the built-in types, we have to implement Decoder for it. That means each of our case classes is required to have implementation of Decoder for everything to work smoothly.

4.4.3 Using `circe`

Input for parsing is always plain string representation of the query. `Libpg_query` is then used to get the JSON representation of parse tree. Then I parse the JSON using *circe* Decoder as Node type, which is an abstract class for all possible Nodes representing nodes of the SQL parse tree. In Node apply method correct Node subtype is chosen and Decoder for that subtype is used. Following the approach *circe* uses, the parsing returns `Either[PgQueryError, Node]`.

4.4.4 Parse expressions

For parsing expressions, I use similar approach. The difference is that before the expression is sent to `libpg_query`, the prefix ”SELECT ” is added. That way valid query should be created (if expression is valid) and following that the process is the same as for query. However, when we receive Node result, we have to get the expression only. That is done using pattern matching, since we expect `SelectStmt` node and we know its structure. Extracted expression is then returned as result.

4.4.5 Prettify

Prettify goes one step beyond the parsing of the query. In case the parse tree is built succesfully, it uses `Node.query` method. Depending on the structure of each Node, the query method is implemented to recursively build the whole parse tree back to SQL query in the string form.

4.5 Scala custom interpolators

4.5.1 What are interpolators?

Since version 2.10, Scala offers a new possibility of string interpolation. [8] This allows me to create generic queries with variables instead of direct values. That way we can define and reuse queries, without unnecessary copying and pasting of code. The idea behind Scala interpolation is the processing of string literals. For example, this code `id"Interpolated text"` is transformed into the call of method ”id” on instance of `StringContext` class. By extending this existing class we can introduce custom interpolators, which allows for a clear definition of these generic query definitions.

String concatenation

```
val query: String =
 "SELECT " + columnName + " FROM students WHERE " + expression
PgQueryParser.parse(query)
```

String interpolation

```
query"SELECT $columnName FROM students WHERE $expression"
```

4.5.2 Implementation

The first version of the library used only runtime validation. I defined my custom interpolator called *query*

4.5.3 'query' and 'expr'

There are currently two custom interpolators, one is meant for full, valid SQL queries, i.e. `SELECT * FROM students`. The second one is used for expressions, that can be inserted into the queries as parameters. Expression can be many different things, column name, constant, expression, function call etc.

4.6 Scala macros

Since we want to achieve compile time validation, I had to find a way to explicitly tell the Scala compiler. If the query would be defined as function, taking parameters, it would wait for runtime, when the parameters will be known (not just the types, as it is when compiling). And then each call to the function would be evaluated separately.

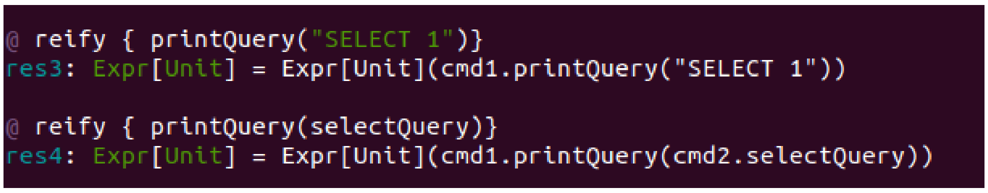
What we want to have is to validate query at compilation, so it creates at the parameter positions "placeholders". These will know the expected type, so every value passed to the function with correct type will be correct. That will not require further validation. And in case the query is not valid at compile time, we will get compile time error right away, making it easier for us to debug the code and fix it.

That is where Scala macros are useful. They have same signature as functions, but their body consists of `macro` keyword and name of the macro function. *It will expand that application by invoking the corresponding macro implementation method, with the abstract-syntax trees of the argument expressions args as arguments.* [9] I think that little description of what abstract syntax trees are is required here. In context of Scala, the AST is used as internal representation of the executed program.

4.6.1 Scala AST and Reflection library

Macros are part of the Scala reflection library. We will specifically talk about the "Compile-time reflection". *Scala reflection enables a form of metaprogramming which makes it possible for programs to modify themselves at compile time.*[10]

When we enter execution of macro, we have the context and the function arguments. Everything is in the form of AST, so programming macros is slightly different from the usual programming in Scala. In simple terms context tells us where the macro was called from, which class, method name etc.



```
@ reify { printQuery("SELECT 1")}
res3: Expr[Unit] = Expr[Unit](cmd1.printQuery("SELECT 1"))

@ reify { printQuery(selectQuery)}
res4: Expr[Unit] = Expr[Unit](cmd1.printQuery(cmd2.selectQuery))
```

Figure 4.1: Differences in AST between parsing string directly and parsing variable

4.6.2 Implementation

4.6.3 Lifting

4.7 Combining interpolators and macros

4.7.1 Parameterized queries in PostgreSQL

Before we can get to the part, where our custom interpolator is simple call to the macro, which does the validation, we have to talk about implementation of placeholders in PostgreSQL. There is existing support for something called *Prepared statements*. These allow for placeholders inside the query, in form of **$n** where **n** must be positive integer.

During compile time each variable in our interpolated string is during known by name only. In macro first thing we have to do is build the string itself from the *StringContext* and the arguments. To keep the final query valid, each of the arguments has to be replaced with *$n*. Let's say we have the following example.

```
query"SELECT $columnName FROM students WHERE $expression"
```

If we tried to pass this string directly to the libpg_query, we would get empty JSON result, because this is not a valid query. That means it is neccessary for us to transform it into this form.

```
query"SELECT $1 FROM students WHERE $2"
```

This returns correct parse tree, where each of the placeholders contains node of ParamRef type.

4.7.2 Transforming syntax tree

4.8 Testing

4.9 Summary

4.10 Publishing library

Chapter 5
Conclusion

Bibliography

[1] *What Is PostgreSQL?* [online]. [cit. 2021-06-14]. Available from: https://www.postgresql.org/docs/13/intro-whatis.html

[2] *Stack Overflow Annual Developer Survey* [online]. [cit. 2021-06-13]. Available from: https://insights.stackoverflow.com/survey/2020#technology-databases-all-respondents4

[3] *What is Quill?* [online]. [cit. 2021-04-25]. Available from: https://github.com/getquill/quill/

[4] NORRIS, Rob. *Doobie documentation* [online]. [cit. 2021-04-25]. Available from: https://tpolecat.github.io/doobie/

[5] The PostgreSQL Global Development Group *The Parser Stage* [online]. [cit. 2021-06-20]. Available from: https://www.postgresql.org/docs/10/parser-stage.html

[6] FITTL, Lukas. *libpg_query* [online]. [cit. 2021-04-25]. Available from: https://github.com/pganalyze/libpg_query

[7] Oracle *Java Native Interface* [online]. [cit. 2021-06-20]. Available from: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[8] SUERETH, Josh. *String interpolation* [online]. [cit. 2021-04-25]. Available from: https://docs.scala-lang.org/overviews/core/string-interpolation.html

[9] BURMAKO, Eugene. *Def macros* [online]. [cit. 2021-04-25]. Available from: https://docs.scala-lang.org/overviews/macros/overview.html

[10] Heather Miller, Eugene Burmako, Philipp Haller *Compile-time reflection* [online]. [cit. 2021-06-19]. Available from: https://docs.scala-lang.org/overviews/reflection/overview.html#compile-time-reflection

Appendix A

Acronyms

API	Application programming interface
JVM	Java virtual machine
JSON	JavaScript Object Notation

Appendix **B**

Contents of enclosed CD

```
readme.txt ........ the file with CD contents description
exe ............... the directory with executables
src ............... the directory of source codes
  wbdcm ........... implementation sources
  thesis .......... the directory of LaTeX source codes of the thesis
text .............. the thesis text directory
  thesis.pdf ...... the thesis text in PDF format
  thesis.ps ....... the thesis text in PS format
```