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 25, 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 25, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

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 PostgreSQL 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 PostgreSQL

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

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

2.1.1 Parse tree

PostgreSQL internally uses parse trees to process SQL Queries. The whole parsing comprises multiple stages. First, a query passed in form of plain text is transformed to tokens using a tool *Flex*. Next up the parser generator called *Flex* is used. It 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.

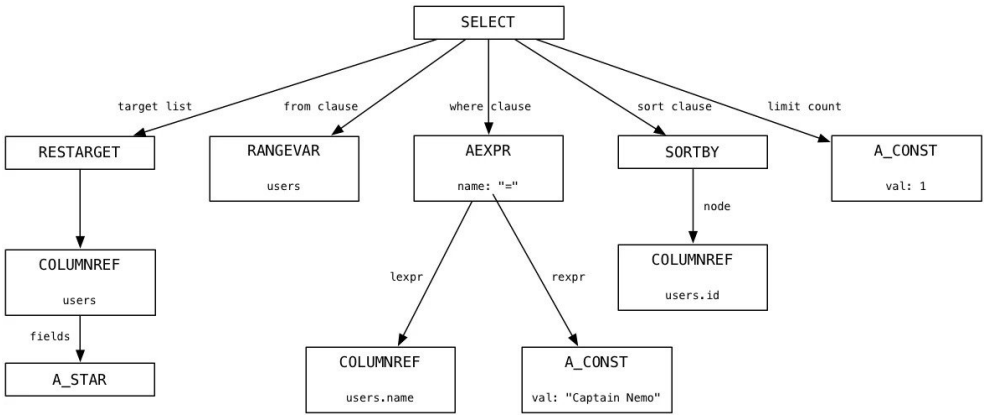

Figure 2.1: Visualisation of parse tree for `"SELECT * FROM users WHERE name = 'Captain Nemo' ORDER BY id ASC LIMIT 1"`

During the *Parse stage*, the parser checks the query string for valid syntax. It does not lookup in the system catalogs, and for that reason, it is independent of the database structure.

2.1.2 Reasons to use parse trees

Having the option to work with parse tree directly, can prove useful in multiple cases. [3]

- Extracting specific part of query
  Using parse tree, we will be able to easily extract parts like column names from the `SELECT` target list, expression from the `WHERE` statement or nested statement from some complicated SQL query.

- Modifying part of the query string
  In a similar fashion to extracting, we can also replace parts in the query. We can for example change the `sort_clause` in Select statement or change the target columns of the query.

- Determine type of query
  It can be also used to accomplish load balancing in applications, by deciding whether the query is read only, or it writes something into the database.

2.2 Scala

2.2.1 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.

2.2.2 Static typing

Besides the functional fundamentals, Scala belongs to the family of statically typed languages. This family also includes languages like C, C++, Java, or Haskell. Therefore, every single statement in Scala has a type.[4]

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 that we don't have to worry about specifying every object's type.

Since Scala compiler knows the types of every statement, it is able to reveal bugs during compilation. That is a great thing, because the sooner we can identify a bug, the easier it should be to fix it.

Chapter 3

Existing solutions

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. The connection to the database can be established similarly as it would be done in Java. But there are multiple existing libraries made for Scala, that ensure an easier way for the programmer to work with databases. Below there are few examples of libraries that were created for that specific reason.

3.1.1 Quill

Quill provides a Quoted Domain Specific Language (QDSL). [5] Its primary usage is to generate SQL queries, using only Scala code which resembles collection-like operations using combinator methods, such as a filter or map. Doing it this way, Quill also provides type-safe queries, based on validation against defined database structure. The query generation requires a defined case class database structure. Quill also provides compile-time validation of the queries by checking against an existing database connection.

3.1.2 Doobie

Next up there is Doobie, which is presented as *"Doobie is pure functional JDBC layer for Scala"*. [6] In this library we can create pure SQL queries in plain text form.

Just like in Quill, validation is possible only with an existing database. Additionally, it is only possible to validate during runtime.

3.2 Difference in approach

3.2.1 Database-independent validation

As we can see, working with the database has been already done by multiple existing libraries. However, sometimes we might not have the option to use the database to validate the queries. And if we do and the query is not valid, we might have to backtrack to find the source of the problem.

The goal is to use the parse tree generated during the *Parser stage* of the PostgreSQL parser. As mentioned before, the parsing is independent of the existing database. Thanks to that, we can check whether the syntax of the SQL query is valid.

3.2.2 Getting parse tree

To get the parse tree we have to access the internal functions of the PostgreSQL parser. These internal functions are not accessible directly, fortunately, the PostgreSQL[8] wiki points us in the direction of *pg_query, a Ruby gem which can generate query trees in JSON representation.* This then further leads to `Libpg_query`, which will help us get the parse tree.

3.2.3 Libpg_query

`Libpg_query` is an open-source C library created by Lukas Frittl. It uses parts of the PostgreSQL server to access the internal `raw_parse` function, which returns the internal parse tree. It accesses internal functions of the server, which allows the library to get the parse tree for each valid query. 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.

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. The important function from `libpg_query` is the `pg_query_parse` function.

The `pg_query_parse` takes the plain text SQL query in form of `const char*`. Then it calls the extracted parts of the PostgreSQL server and returns the parse tree as JSON. Once we have that, we can decode the JSON and map it onto the created case class structure in Scala.

Here we have a simple code snippet describing simple usage taken from the GitHub README[9].

```
#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);
}
```

Chapter 4

Realisation

4.1 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 simply 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
```

4.2 Using native library

Now we have to use 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.

4.2.1 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.

4.2.2 Java native interface

JNI is programming interface for writing Java native methods.[11] 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 a bridge between native code and the Scala program. Now we can call `libpg_query` inside the new C file, then compile it and create our shared library. The next step is loading the .so file into the Scala program. That was achieved by `System.loadLibrary` function. The system uses the local `java.library.path`, so it was necessary to add the path of the file with the shared library to the path variable.

4.2.3 Issues

The raw JNI approach worked fine for a single module. However, the project is now separated into several submodules (native, parser, macros, core). By default, the sbt task runs in the same JVM as sbt.[10] Using for example `sbt test` command runs tests for each submodule at the same time and since multiple submodules try to access the native library. Tests for each submodule are run on the same JVM, but different ClassLoader. However, the created shared library can be linked only once, so the execution fails with `UnsatisfiedLinkError`.

4.2.4 sbt-jni

To fix these issues, I used an existing JNI wrapper for Scala called `sbt-jni`. It is a suite of sbt plugins for simplifying the creation and distribution of JNI programs. To name the ones I used, `JniJavah` works as a wrapper around the `javah` command to generate headers for classes with `@native methods`. Next one I used is `JniLoad`, which enables correct loading of shared libraries through `@nativeLoader` annotation.

Another applied fix is the `fork := true` setting in `build.sbt`. This causes the task to run in different process and JVM. This gets rid of the `UnsatisfiedLinkError`.

4.3 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 a pure functional library called Argonaut. It is great for parsing, traversing JSON, but the main functionality I use is autoderivation of `Encoder` and `Decoder` instances for a given algebraic data type.

For each case class that represents one node of the parse tree, we have to generate `Encoder` and `Decoder` instances.

4.3.1 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.

4.3.2 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.

4.3.3 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]`.

4.3.4 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.

4.3.5 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.

4.4 Scala custom interpolators

4.4.1 What are interpolators?

Since version 2.10, Scala offers a new possibility of string interpolation. [12] 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"
```

4.4.2 Runtime implementation

The first version of the library used only runtime validation. I defined my custom interpolator called *query*. Inside the interpolator, the arguments were merged into `StringContext`. Following that, the finished string got parsed using `PgQueryParse.parse` method.

4.5 Scala macros

Since we want to achieve compile time validation, we have 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 do 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 matching type will result in valid query. In case the query is not valid, 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.* [13] 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.

4.5.1 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.*[14]

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

4.5.2 Liftable

Scala uses trait `Liftable[T]` to specify conversion of type to tree. It has only single abstract method - `def apply(value: T): Tree`. Since the goal of using macros is to validate queries at compile time, we will use `parse` method from `PgQueryParse`, which returns the parse tree in form of a `Node`. We will have to 'lift' the result, so we can return the correct `Tree` representation. [15]

Therefore, we have to define Liftable[Node]. We are using three macros, that generate Liftable object from the original.

- `LiftableCaseClass`
- `LiftableCaseObject`
- `LiftableEnumeration`

Each one of them provides implementation of creating implicit object, which extends Liftable[T] and implements the logic of creating corresponding `Tree`.

4.6 Combining interpolators and macros

4.6.1 Parameterized queries in PostgreSQL

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

During compile time each variable in our interpolated string is known by name only. In macro, the 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 the placeholder *$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 an empty JSON result, because this is not a valid query. That means we have to transform it into this form.

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

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

4.6.2 Implementation of macro validation

First of all, the context prefix is validated. The prefix contains info about the expression the macro was called on. We expect that the macro is always called using a custom interpolator from the `CompileTimeInterpolator` object. That is validated using pattern matching of the context prefix tree against the quasiquote representing the expected tree. Quasiquotes are another example of an interpolator. They are used to convert a snippet of code into its tree representation.[16] During the validation we also extract the `List[String]` from the `StringContext`.

4.6.3 Handling arguments

Once we validate the prefix and extract the string from `StringContext`, we have to transform the arguments. Each argument is represented by its AST, which we need to retain. Therefore, we create an indexed map from the arguments with type `Map[Int, Tree]`. This structure is used in the Transforming step later on.

4.6.4 Validation of query with placeholders

For each argument we generate placeholder starting from `$1` up to `$n`. The placeholders are then interspersed into the extracted `List[String]` we got from `StringContext`. This finished string is then parsed using our `parse` method from `PgQueryParser`, which gets us the parse tree representation in the form of a `Node`.

4.6.5 Transforming syntax tree

Now we have the result parse tree, which contains `ParamRef` nodes in places where the arguments are supposed to be. As I described in section 4.6.2, we have to lift the Node structure to the AST representation before we can return the result. However, before we do it, we have to insert the arguments back into the structure in their corresponding places. For that purpose, we are going to use our custom class `ParamRefTransformer`. It extends the abstract class `Transformer`, which *implements a default tree transformation strategy: breadth-first component-wise cloning.*[17]

The `ParamRefTransformer` class takes the `Map` we created in section 4.7.2 as input parameter. Then it overrides the `transform` method, which takes one argument - the `Tree` object. `Tree` is the representation of the parse tree. The method then iterates over each node of the `Tree` and matches the `q"ParamRef(${Literal(Constant(constant:Int))}, ${_})"` pattern.

Whenever the pattern matches the current `Tree`, the whole `ParamRef` is replaced by the `Tree` value from the `Map` with corresponding index. If the pattern doesn't match, the original method from the superclass is called. The original transform then applies the transform function again on each leaf of the current node. This way, every node of the AST is traversed, and we replace each `ParamRef` node with the original argument.

4.6.6 Type checking

In the end, we have the finished SQL parse tree in the form of AST. The parsing in `PgQueryParser` ensures that the query is valid. Within the tree, each placeholder is replaced with the original argument. The compiler then compares the type of the argument with the expected type in the context of the parse tree structure. If the type of the argument isn't correct, it throws the *type mismatch* error.

4.6.7 Implicit conversions

Since we introduced the validation and type checking using the macro, we could only use interpolator, which uses the macro with arguments that are Node objects or a more specific type of `Node`, depending on where we try to insert it. That means if we wanted to define a function, which takes `String` as an argument, we couldn't use it in the interpolator. Instead, we had to parse it as an expression and only then pass it to the interpolator.

Fortunately, Scala provides *implicit* keyword that can be used to create the implicit conversion from one type to another. *An implicit conversion from type S to type T is defined by an implicit value which has function type* $S \rightarrow T$*, or by an implicit method convertible to a value of that type.*[18] Whenever the type of an expression does not conform to the expected type, compile attempts to find an implicit conversion function, which can be used to get the correct type. The order in which the compiler looks for the implicit conversion is as follows: [19]

1. Implicits defined in the current scope
2. Explicit imports (i.e. `import ImplicitConversions.int2string`)
3. Wildcard imports (i.e. `import ImplicitConversions._`)
4. Same scope in other files

Currently the library supports implicit conversions from `String` and `Int` to `ResTarget` and `A_Const` nodes. These nodes cover majority of possible expressions that can be used. The conversion from `String` to `ResTarget` uses another macro, which validates the expression. The rest creates the desired objects directly.

4.7 Testing

4.7.1 Scalatest

4.7.2 Parser testing

4.7.3 Core testing

4.8 Summary

So far we have library, which can validate queries using C library called `libpg_query`. To connect our Scala code with the native code we are using sbt-jni plugins. The JSON containing the parse tree representation is then parsed to our custom case class structure using functional library for working with JSON, `circe`. Then we implemented our own interpolators, one for expressions and another one for full fledged queries. To achieve compile time validation we used macros, where we are working with abstract syntax trees of the program itself. The final query is then typechecked and throws compilation errors, whenever the types don't match.

4.9 Future work

The library can be for now considered a prototype. It covers majority of generally used SQL keywords and queries. The list of SQL keywords and possible combinations is still vastly broader, so the library can be further expanded to eventually cover all of the possible cases.

At the end of May, 2021 the newest version of `libpg_query` was also released. It contains plenty of changes, support for the PostgreSQL 13 version, changes to JSON output format, new Protobuf parse tree output format, added deparsing functionality from parse tree back to SQL and more. [20].

4.10 Publishing library

The whole library is currently publically accessible in repository on github.com under the name *pgquery4s*.

Chapter 5

Conclusion

Bibliography

[1] *What Is PostgreSQL?* [online]. [cit. 2021-06-21]. Available from: https://www.postgresqltutorial.com/what-is-postgresql/

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

[3] PENG, Bo. *Introducing PostgreSQL SQL Parser* [online]. [cit. 2021-06-24]. Available from: https://www.pgcon.org/2019/schedule/attachments/556_PostgreSQL_SQL_parser.pdf

[4] Mayank Bhatnagar *Magic lies here - Statically vs Dynamically Typed Languages* [online]. [cit. 2021-06-21]. Available from: https://medium.com/android-news/magic-lies-here-statically-typed-vs-dynamically-typed-languages-d151c7f95e2b

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

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

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

[8] *Query Parsing* [online]. [cit. 2021-06-25]. Available from: https://wiki.postgresql.org/wiki/Query_Parsing

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

[10] *Forking* [online]. [cit. 2021-06-22]. Available from: https://www.scala-sbt.org/0.12.3/docs/Detailed-Topics/Forking.html

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

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

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

[14] MILLER, Heather, BURMAKO Eugene and HALLER Philipp *Compile-time reflection* [online]. [cit. 2021-06-19]. Available from: https://docs.scala-lang.org/overviews/reflection/overview.html#compile-time-reflection

[15] SHABALIN, Denys. *Quasiquotes introduction* [online]. [cit. 2021-06-22]. Available from: https://docs.scala-lang.org/overviews/quasiquotes/lifting.html

[16] SHABALIN, Denys. *Quasiquotes introduction* [online]. [cit. 2021-06-22]. Available from: https://docs.scala-lang.org/overviews/quasiquotes/intro.html

[17] *Transformer* [online]. [cit. 2021-06-23]. Available from: https://www.scala-lang.org/api/2.12.9/scala-reflect/scala/reflect/api/Trees$Transformer.html

[18] *Implicit conversions* [online]. [cit. 2021-06-24]. Available from: https://docs.scala-lang.org/tour/implicit-conversions.html

[19] SUERETH, Josh. *Implicits without the import tax* [online]. [cit. 2021-06-24]. Available from: http://jsuereth.com/scala/2011/02/18/2011-implicits-without-tax.html

[20] FITTL, Lukas. *Release 13-2.0.0* [online]. [cit. 2021-06-24]. Available from: https://github.com/pganalyze/libpg_query/releases/tag/13-2.0.0

Appendix A

Acronyms

API Application programming interface

JDBC Java Database Connectivity

JPA Jakarta Persistence

JSON JavaScript Object Notation

JVM Java virtual machine

SQL Structured Query Language

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