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.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Scala library for constructing statically typed PostgreSQL queries

Petr Hron

Department of software engineering
Supervisor: Ing. Vojtěch Létal

June 27, 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 27, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Petr Hron. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

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

Hlavní téma této práce je vývoj a implementace knihovny pro vytváření staticky typovaných SQL dotazů v jazyce Scala, společně s průzkumem existujících Scala knihoven, které se zabývají vytvářením SQL dotazů.

Nejprve jsou představeny použité technologie a popsány existující knihovny pro práci s PostgreSQL. Implementační část následně popisuje kroky potřebné k vytvoření knihovny. Popsáno je propojení Scaly a knihovny v jazyce C, použití *circe* knihovny, která slouží pro parsování JSON výsledků a vytvoření *case class* struktury pro reprezentaci syntaktických stromů SQL výrazů. Další velká část implementace popisuje makra v jazyce Scala a jejich využití pro validaci SQL dotazů během kompilace. Nakonec je popsán nynější stav knihovny společně s plány pro budoucí vylepšení.

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

Abstract

The focus of this thesis is development of the Scala library capable of creating statically typed queries, together with research of Scala libraries that deal with constructing SQL queries.

First, technologies used for this project are introduced, followed by research of existing Scala libraries for working with PostgreSQL. The implementation part then follows the steps that were required to create the library. It covers the connection of Scala with C library, use of *circe* library for parsing JSON results, and creating case class structure to represent SQL parse trees. Another big part of implementation covers macros in Scala and their usage for compile time validation of queries. Then the current state of the library is described, together with plans for future improvements.

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

Contents

List of Figures

List of Listings

Introduction

Scala is a programming language that combines object-oriented programming with the support of functional programming. The source code of Scala is intended to be compiled into Java bytecode and run on JVM. That makes it a great starting point for programmers who want to get their first experience with functional programming.

Then we have PostgreSQL, one of the most popular relational database management systems currently available. It has wide support for working with different programming languages, regular updates, improvements, and plenty of documentation and tutorial available everywhere. The fact that the whole project is open source and free allows anyone to dive right into it.

In the world of Scala, there are already few libraries made to work with the PostgreSQL database, create queries, and more. Most of them are used with a direct connection to the database. Because of that, the queries used are validated only when they are executed. However, Scala is a statically typed language, which means that type checking is done at compile time, which eliminates few categories of possible bugs before the code is run. The goal is to apply the same approach to validation of the SQL queries, so we can, to some extent, do that during compilation. By using SQL parse trees, we can also create and update statically typed queries.

In the theoretical part, we will talk about technologies used in this project like Scala, PostgreSQL and SQL parse trees. Then we will show few examples of existing Scala libraries for working with databases, their pros, cons, and how does our library fit into the whole ecosystem. We will also describe the C library, which is used to access the internal parse function of the PostgreSQL server, to get the parse tree.

Then in the realization part, we will go through the implementation process. We will start with the representation of the parse tree in Scala and accessing the C library from our Scala code. Then we will talk about macros and how to use them. In the end, we combine the macros and custom interpolators to introduce type-checked queries.

Chapter 1

Technologies used

1.1 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.

1.1.1 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 *Bison* 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.

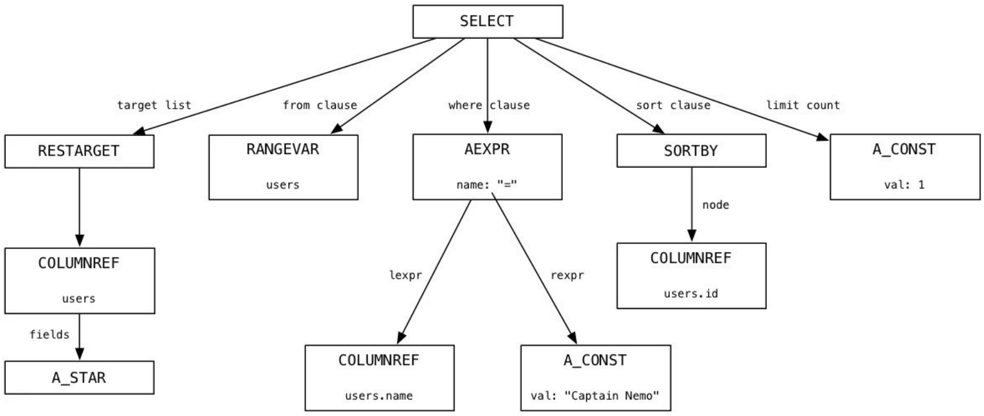

Figure 1.1: Visualisation of parse tree for `"SELECT * FROM users WHERE name = 'Captain Nemo' ORDER BY id ASC LIMIT 1"`[3]

During the *Parse stage*, the parser checks the syntax of a query string. It does not do any lookups in the system catalogs, and for that reason, it is independent of the database schema. In the following chapters you will see how our library seamlessly exposes the internal PostgreSQL parse trees to the higher level language and how this can be used to validate queries during compilation.

1.1.2 Reasons to use parse trees

Having the option to work with parse trees can prove useful in multiple cases.[4] However, working with the tree directly can be tedious work, especially for big nested queries. That's why we will use high-level Scala features to make working with them easier while still keeping the option to access and work with the parse tree on the lower level.

- 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.

- Determining 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.

1.2 Scala

1.2.1 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.

1.2.2 Functional error handling

Scala avoids the usual `try catch` error handling, which is used in Java. The only occasion where it might be used is when we are calling some Java API or unsafe library. Instead, Scala is using monads, for example, `Option[T]`, `Try[T]`, and `Either[A, B]`. Monad is in simple terms container around a certain type, for which there is a `flatMap` function, which allows us to compose their individual instances.

- **Option[T]**
  `Option` is definition of nullable type. It contains either `None` or `Some(T)` object. For example, we're trying to find a certain number in the `List[Int]`. We define a function that will traverse the list, and if the value is present, it returns the number, wrapped like this `Some(Int)`. If the value is not found, it returns `None`.

- **Either[A, B]**
  `Either` is similar to `Option`, but instead of returning simple `None`, which does not tell us what went wrong, it returns `Left(A)` or `Right(B)` object. `Right` is just like `Some`, it's returned when everything went right and we got the expected result. On the other hand, we return `Left` whenever something did not go as planned. It can contain info about the problem.

- **Try[T]**
  `Try` is more specific `Either`. It is the same as `Either[Throwable, B]`. The difference is that instead of `Right` there is `Success` and instead of `Left` there is `Failure`. However, `Failure` can be only exception. It is mostly used as replacement in situations, where we would use `try catch` block.

1.2.3 Static vs. dynamic 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.[5] Statically typed languages validate the type during compile time and once it is compiled, it can be run multiple times.

On the other hand, dynamic typing does all type checking during runtime, and every time we want to run the program, it has to be compiled again. Examples of languages, which use dynamic typing, are Python, Ruby, and PHP.

1.2.4 Strong vs. weak typing

Strongly typed languages enforce strict restrictions on intermixing values with different data types. Thanks to that, the behavior is more predictable than it would be for weakly typed language. The majority of strongly typed languages require explicit declaration of type for each variable. However, for Scala, that's not entirely true. It is strongly typed language, but it 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 the type for every statement.

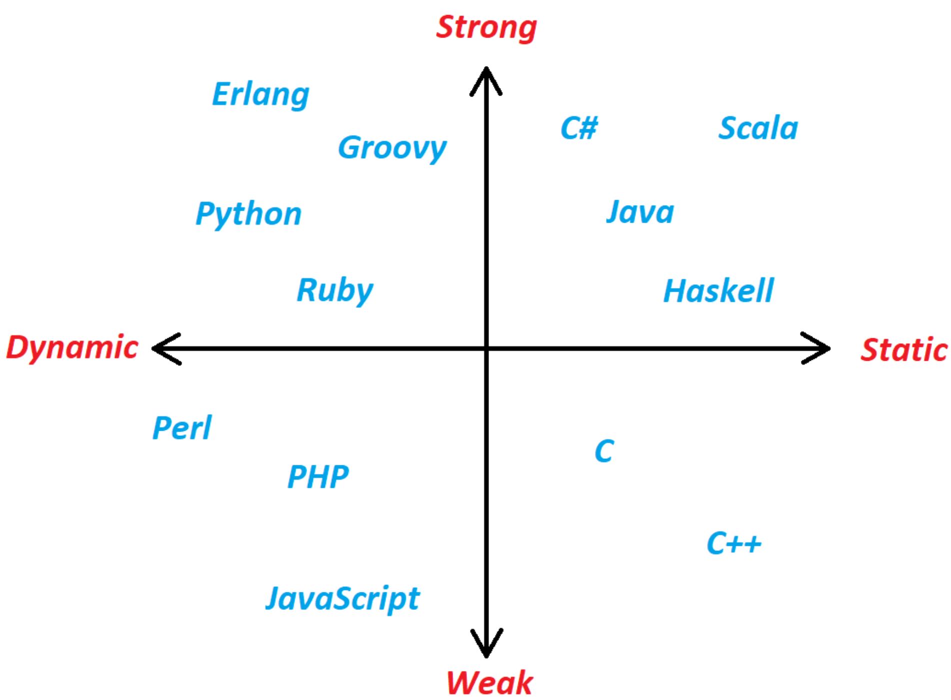

Figure 1.2: Languages divided into groups.

Chapter **2**

Existing solutions

2.1 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.

2.1.1 Quill

Quill provides a Quoted Domain Specific Language (QDSL). [6] 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. The query generation requires defined case classes, where each case class represents one table in the database. *Quill* supports generating queries for two languages - SQL and CQL. The queries are generated at compile time by translating the AST to the target language. *Quill* also provides compile-time validation of the queries by checking against an existing database connection.

2.1.2 Doobie

Next up there is *Doobie*, which is presented as *"Doobie is pure functional JDBC layer for Scala"*. [7] In this library we can create pure SQL queries in plain text form. Thanks to the low level access to the `java.sql`, we can create a connection to the database in functional style.

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

2.2 Difference in approach

2.2.1 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. This project isn't meant as a competitor to those mentioned libraries. Instead, it is recommended to use them together. In the example project that was implemented to showcase the usage of our library, we are using *Doobie*. The queries are created and validated using our implementation, and then they are executed on a specific database using a connection created by *Doobie* to show that the queries are, in fact, valid.

2.2.2 Implementation goal

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.

All that will be done during compilation. We will create an interpolator, which will generate the parse tree structure. As arguments the interpolator will accept either `Nodes` directly, or even primitive types like `String` or `Int`, which will be transformed into `Node` thanks to the implicit conversion.

For example we will be able to create function, that will create different filtering queries, based on passed expression.

```
def filterStudents(expr: ResTarget) : Node =
    query"SELECT * FROM students WHERE $expr"
```

Listing 2.1: Example of `query` interpolator usage

2.2.3 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[9] wiki points us in the direction of *pg_query, a Ruby gem which can generate query trees in JSON representation.* Internally it uses *libpg_query*, which is standalone C library used to parse PostgreSQL queries.

2.2.4 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.

```
#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);
}
```

Listing 2.2: `Libpg_query usage example` [10]

Chapter 3

Realisation

3.1 Parse tree representation

3.1.1 C

If we look at internal representation of the tree directly in the PostgreSQL parser, it defines each possible node of the parsetree in form of `struct`. Any type of node is guaranteed to have `NodeTag` as the first field.

```
typedef struct Node
{
   NodeTag  type;
} Node;
```

Listing 3.1: `Node` representation in PostgreSQL parser

`NodeTag` is an `enum`, which contains all types of possible nodes. This is used to achieve polymorphism in C. Thanks to that guarantee, any type of node can be cast to `Node` without losing the information about the type. That allows casting the `Node` back to the original type when needed.

This fact is also used when a node contains another node as a leaf. Most of the time, the type of the required node isn't specified directly instead, `Node` pointer is used. This provides flexibility for the parameters of the nodes.

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

Listing 3.2: `A_Expr` representation in PostgreSQL parser

However, in many places, the `Node` reference could be replaced by a smaller subset of possible types. Having `Node` everywhere creates a flat structure, which could be improved, especially for purposes of type checking.

3.1.2 Scala

Since we already have existing parse tree representation in C we can mirror it in Scala. Scala is an object-oriented language, therefore each type of node should be defined by its own class. However, using case classes offers several advantages over classes in Scala:

- Case classes can be pattern matched
- Automatic definition of equals and hashcode methods
- Automatic definition of getters

First, we convert the `struct Node` into `abstract class Node`. It will be used as base class for every node of the parse tree.
Each `struct` can be then converted into case class in Scala.

- Each `Node*` is converted to `Option[Node]` - same for other variables, which are pointers to specific `Node` type
- Each `List*` is converted to `List[Node]`
- Primitive data types are converted to their Scala equivalents (e.g. `int` to `Int`, `char*` to `String`)
- The `NodeTag` parameter can be ommited, because in Scala we can use pattern matching to check the type of the `Node`.

Each `enum` is converted to object extending abstract class `Enumeration`.
Converted `A_Expr` case class

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

Listing 3.3: `A_Expr` representation in Scala

3.2 Using native library

Libpg_query provides `pg_query_parse` function, which takes `const char*` parameter (the SQL query) and returns parse tree in form of JSON. However, because of difference between native code and java byte code, we can't directly import the C library into our Scala code.

3.2.1 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.

3.2.2 Java native interface

JNI is programming interface for writing Java native methods.[12] It is used to enable Java code to use native applications and libraries. The native functions are implemented in separate generated `.c` or `.cpp` file. Let's say we defined our class with native method like this:

```
package com.pgquery

class Wrapper {
  @native def parse(query: String): String
}
```

Listing 3.4: Scala class with native method

Then we compile the file with the Scala source code. From the compiled code we generate the JNI header using `javah` command. The definition of the native function then looks like this:

```
JNIEXPORT void JNICALL Java_com_pgquery_Wrapper_parse
  (JNIEnv *env, jobjectobj, jstring string)
{
  // Method native implementation
}
```

Listing 3.5: Generated JNI header

The parameter list for the generated function contains a `JNIEnv` pointer, a `jobject` pointer, and any Java arguments declared by the Java method.[13] The `JNIEnv` pointer is used as an interface to the JVM. Thanks to that we can

for example use function the convert native `const char*` to and from Scala string.
The `jobject` pointer is used to access class variables of the object the method was called from.

The JNI header is then compiled, with included JNI headers from local Java JDK. The extension of the final shared library depends on system - `.so` for Linux, `.dylib` for MacOS and `.dll` for Windows. The created native library is then loaded using `System.loadLibrary`.

3.2.3 sbt-jni

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

3.3 Parsing JSON result from libpg_query

There are few different libraries that can help with parsing JSON. From those, we are using *circe*. *Circe* is fork of a pure functional library called *Argonaut*. It is great for parsing, traversing JSON, but the main functionality we are using is the auto derivation of `Encoder` and `Decoder` instances for a given algebraic data type.

3.3.1 How decoding works in circe

Basic decoder for case class in *circe* iterates over all parameters of the case class and matches the name of the paramater with the key in JSON. Then it attempts to parse the value as the type of the parameter. Let's say we have case class representing person together with implicit definitions of `Decoder`.

```
@JsonCodec case class Person(age: Int,
                             name: Option[String])
```

Listing 3.6: Scala case class with `@JsonCodec` annotation

The `@JsonCodec` annotation simplifies the process of generating the `Decoder` and `Encoder` using semi-automatic derivation. [14]
Next we have simple JSON, that we want to parse as `Person` object.

```
parse("{ "age" : 5 }").as[Person]
```

Listing 3.7: Deparsing JSON using *circe*

First the string is converted to `Json` - `circe`-specific representation of JSON. Decoding starts with the `age` parameter and successfully finds the key in JSON. Then it type checks the value, if it is `Int`, or if it can be converted to `Int` using any implicit conversion. Then it continues with the `name` parameter. The JSON doesn't contain key `name`, but the type of name is `Option[String]`, which represents nullable type. The decoder then sets `Name` as `None` and the decoding is finished. *Circe* then returns `Right(Person(5, None))`.
However, if the `name` was `String` instead, the decoding would stop and return:

```
Left(
    DecodingFailure(
        Attempt to decode value on failed cursor,
        List(DownField(name))
        )
    )
```

Listing 3.8: Return value on decoding failure

3.3.2 Query parsing

Parsing of the SQL queries is covered by `PgQueryParser` object. The main `parse` function takes SQL query as a plain string on input. The query is parsed using *libpg_query*. The string representing JSON is then converted to *circe Json* type and parsed. The result is then decided based on pattern matching of the output of *circe*.

To keep code consistent, we are following the monadic approach *circe* uses. The parsing method then returns `Either[PgQueryError, Node]`.

- When everything goes well, we get the `Node` and return it as `Right(node)`.
- If the result is an empty array, it suggests that the query was not valid and `libpg_query` returned JSON with empty array. `Left(EmptyParsingResult)`
- If the parsing of the JSON fails, we get `DecodingFailure` object from `circe`. Return value is then `Left(FailureWhileParsing(DecodingFailure))`

3.3.3 Parse expressions

Besides parsing full queries, we also support the parsing of expressions. Having access to parse trees of expression will be useful for the interpolation of queries. Just like `parse` method used for parsing queries, the `parseExpression` takes expression as string on input. However, *libpg_query* only supports parsing of full valid queries. For that reason, we are using a small trick, where we add the prefix "SELECT " in front of the expression. This works, because by definition, `targetList` of `SelectStmt` can contain arbitrary expression.

The created query is then parsed using the original `parse` method. Since the structure of the `SelectStmt` node is known, we can use pattern matching to extract precisely only the expression. The error handling works in similar fashion as in the `parse` function and the return type is `Either[PgQueryError, ResTarget]`.

3.3.4 Prettify

Prettify goes one step beyond the parsing of the query. In case the parse tree is built successfully, 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 the SQL query in the string form.

3.4 Scala custom interpolators

3.4.1 What are interpolators?

Since version 2.10, Scala offers a new possibility of string interpolation. [15] 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.

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

Listing 3.9: Example of `String` concatenation

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

Listing 3.10: Example of `String` interpolation

3.4.2 Runtime implementation

Although the goal of this project is to validate queries during compilation and transform the interpolated string to `Node` at compile time, the runtime validation is important as well. Parse trees of queries are accessible at runtime.

Simply use the built-in string interpolator to create the query. The parse tree can be generated by the `parse` method of `PgQueryParser`.

3.5 Scala macros

Since we want to achieve compile time validation, we have to explicitly tell the Scala compiler. If the query is defined as a function with parameters, it waits for runtime, when the values of parameters are 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 the query at compilation and create placeholders at the parameter positions. These will have the expected type, so every value passed to the function with the matching type will result in a valid query. If 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 the 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.* [16] I think that little description of what abstract syntax trees are is required here. *Trees are the basis of Scala's abstract syntax which is used to represent programs. They are also called abstract syntax trees and commonly abbreviated as ASTs.*[17]

3.5.1 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.*[18]

When we enter the execution of the 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.

Another tool, often used when working with macros, is called `reify`. It is a method, which takes expression and returns its AST. In the snippet below we can see difference in the AST, when we call the method directly with the `String` vs. passing the `String` as variable.

```
reify { printQuery("SELECT 1") }
res1: Expr[Unit] =
    Expr[Unit](cmd1.printQuery("SELECT 1"))
```

```
val selectQuery: String = "SELECT 1"
reify { printQuery(selectQuery) }
res1: Expr[Unit] =
    Expr[Unit](cmd1.printQuery(cmd2.selectQuery))
```

Listing 3.11: Comparison of ASTs of function calls with `String` and with variable

Here we can see that if our function calls a macro, the compiler does not know the value of parameters of the function, only the type, and name. This will be important when we are going to implement our interpolators using macros.

3.5.2 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. [19]

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

3.6 Combining interpolators and macros

3.6.1 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"
```

Listing 3.12: Query with variable name.

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"
```

Listing 3.13: Query with placeholder.

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

3.6.2 Validation of the query

We are able to build the query string from the `StringContext` and passed arguments. To create a valid query, we have to enumerate the arguments. Each argument is replaced by placeholder starting from `$1` up to `$n`, where `n` is the total count of passed arguments. After we intersperse the placeholders into `List[String]` we can use the `parse` method from `PgQueryParser` for validation. If the query is valid, we get the parse tree representation in the form of a `Node`.

3.6.3 Transforming syntax tree

The parse tree now contains one `ParamRef` node for each argument. The next step is to replace these placeholder nodes with their corresponding arguments. Macros are required to return results in the form of `Tree`. The easiest solution is to `lift` the `Node` we got from *libpg_query* to AST representation and then replace the placeholders with the original arguments.

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.*[21]

The `ParamRefTransformer` overrides the `transform` method, which takes one argument - the parse tree in form of a `Tree`. The method then iterates over each node of the `Tree` and matches the following pattern.

```
q"ParamRef(${Literal(Constant(constant:Int))}, ${_})"
```

Listing 3.14: Pattern of the AST of `ParamRef` node.

Whenever the pattern matches the current `Tree`, the whole `ParamRef` is replaced by the argument AST with same index as the `constant`. 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.

3.6.4 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.

3.6.5 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.*[22] 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: [23]

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.

3.7 Testing

3.7.1 Unit testing

Unit tests are used to test small parts of the code base. They are intended to be run often, so they have to be simple and quick. For testing we are using the most popular option for Scala - *scalatest*.

3.7.2 Parser and core testing

The tests in *parser* submodule cover parsing and deparsing of the SQL queries. The goal is to have every `Node` properly tested. The tests are currently separated into three groups - tests for `DatabaseStmt`, `InsertStmt`, and `SelectStmt`.

Tests in *core* submodule focus on the interpolators and implicit conversions. We are testing both `query` and `expr` interpolator using `Matchers` DSL from scalatest.

```
test("Func call expression test") {
    val expr = expr"MIN(columnName)"
    expr should matchPattern {
      case ResTarget(_, _, Some(_: FuncCall), _) =>
    }
  }
```

Listing 3.15: Test for expression with `FuncCall`

3.7.3 Continuous integration

Our library is intended as open source project, which means we can expect contributions from other developers. *Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently.*[25] For our project, we are using free *TravisCI* service, which provides quick integration with public *GitHub* repositories.

With *TravisCI* we can create building and testing pipeline, that will be run for each new contribution. The pipeline is divided into few steps:

- Before the testing part of the pipeline is run, *TravisCI* sets up local PostgreSQL 10 database and creates `pgquery_example` database.
- The script navigates inside the *libpg_query* folder and compiles the C library.
- Then all tests are run and the artifacts are locally published.
- In the last step of the pipeline, the example project is built and run, using locally published artifacts.

Chapter **4**

Conclusion

4.1 Summary

Since the library is meant as an open-source project, the whole source code is available on `https://github.com/Ivellien/pgQuery4s/` as a public repository. The project is separated into multiple submodules.

- Native

  This module contains everything related to native code. There is the `PgQueryWrapper` class, which implements single method `pgQueryParse` with `@native` annotation. Then there is the *libpg_query* library itself, and the JNI implementation of the native method, which directly calls the C library.

- Parser

  Possibly the most important part of the library. Parser submodule contains the whole existing case class structure representation of the parse tree in *node* and *enums* packages. The `PgQueryParser` object then defines part of our usable public API. Each one of these methods takes string representing SQL query or expression:

  - `json` - Returns JSON representation of the passed query as received from `libpg_query`
  - `prettify` - Creates the `Node` representation of the passed SQL query and then deparses it back to string again.
  - `parse` - Attempts to parse the whole SQL query. Returns result as `PgQueryResult[Node]`.
  - `parseExpression` - Same as `parse`, but instead of parsing the whole input as query it prepends *"SELECT "* to the expression. The expression is then extracted from the parse tree using pattern matching. Return result as `PgQueryResult[ResTarget]`.

- Macros

  The macros submodule is further split into two other subprojects - *liftable* and *macros*. The macros were one of the reasons for splitting up the project because the macro has to always be compiled before it can be used elsewhere.

  The macros subproject currently contains macro implementations for parsing queries, expressions and for implicit conversion from `String` to `ResTarget`.

  The *liftable* subproject contains generators of `Liftable` objects, as explained in section 4.5.2.

- Core

  The core uses the macros package and contains definitions of the custom interpolators for queries, expressions, and implicit conversions.

So far, we have a library, which can validate queries using a 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 a functional library for working with JSON, *circe*. Then we implemented our interpolators, one for expressions and another one for 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 type-checked and throws compilation errors whenever the types don't match.

4.2 Future work

The library can be, for now, considered a prototype. It covers the majority of generally used SQL keywords and queries. However, the list of SQL keywords is long, and together with all the possible combinations, it leaves room for improvement. The library can be further expanded to eventually cover the whole SQL node structure.

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. [24].

Bibliography

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

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

[3] SHAUGHNESSY, Pat. *Following a Select Statement Through Postgres Internals* [online]. 15 Jun 2015. [Accessed 27 June 2015] Available from: https://www.cloudbees.com/blog/following-a-select-statement-through-postgres-internals

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

[5] BHATNAGAR, Mayank. *Magic lies here - Statically vs Dynamically Typed Languages* [online]. Sep 9, 2018. [Accessed 21 June 2021]. Available from: https://medium.com/android-news/magic-lies-here-statically-typed-vs-dynamically-typed-languages-d151c7f95e2b

[6] IOFFE, Alexander. *What is Quill?* [online]. 2021. [Accessed 25 April 2021]. Available from: https://github.com/getquill/quill/

[7] NORRIS, Rob. *Doobie documentation* [online]. 2021. [Accessed 25 April 2021]. Available from: https://tpolecat.github.io/doobie/

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

[9] RENNER, Michael. *Query Parsing* [online]. 2014. [Accessed 25 June 2021]. Available from: https://wiki.postgresql.org/wiki/Query_Parsing

[10] FITTL, Lukas. *libpg_query* [online]. 2021. [Accessed 25 June 2021]. Available from: https://github.com/pganalyze/libpg_query

[11] sbt Documentation. *Forking* [online]. [Accessed 22 June 2021]. Available from: https://www.scala-sbt.org/0.12.3/docs/Detailed-Topics/Forking.html

[12] Oracle. *Java Native Interface* [online]. [Accessed 20 June 2021]. Available from: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[13] MIKHALENKO, Peter. *Discover how the Java Native Interface works* [online]. 6 Sep 2006 [Accessed 26 June 2021]. Available from: https://www.techrepublic.com/article/discover-how-the-java-native-interface-works/

[14] Circe documentation. *Semi-automatic Derivation* [online]. [Accessed 27 June 2021]. Available from: https://circe.github.io/circe/codecs/semiauto-derivation.html

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

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

[17] Scala Documentation. Symbols, Trees, and Types. Scala Documentation [online]. [Accessed 26 June 2021]. Available from: https://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html

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

[19] SHABALIN, Denys. *Quasiquotes lifting* [online]. [Accessed 22 June 2021]. Available from: https://docs.scala-lang.org/overviews/quasiquotes/lifting.html

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

[21] Scala Documentation. *Transformer* [online]. [Accessed 23 June 2021]. Available from: https://www.scala-lang.org/api/current/scala-reflect/scala/reflect/api/Trees$Transformer.html

[22] Scala Documentation. *Implicit conversions* [online]. [Accessed 24 June 2021]. Available from: https://docs.scala-lang.org/tour/implicit-conversions.html

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

[24] FITTL, Lukas. *Release 13-2.0.0* [online]. 18 Mar, 2021. [Accessed 24 June 2021]. Available from: https://github.com/pganalyze/libpg_query/releases/tag/13-2.0.0

[25] Could Bees. *What is Continuous Integration?* [online]. 2021. [Accessed 27 June 2021]. Available from: https://www.cloudbees.com/continuous-delivery/continuous-integration

Appendix A

Acronyms

API Application programming interface

AST Abstract syntax tree

CI Continuous integration

CQL Cassandra Query Language

DSL Domain-specific language

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
```