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 26, 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 26, 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

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. Následně jsou popsané 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 SQL syntaktických stromů. Další velká část implementace popisuje makra v jazyce Scala a jejich využití pro validaci SQL dotazů během kompilace. Následně 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 the development of the Scala library capable of creating statically typed queries, together with research of Scala libraries that deal with constructing SQL queries.

First, the 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

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 *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.

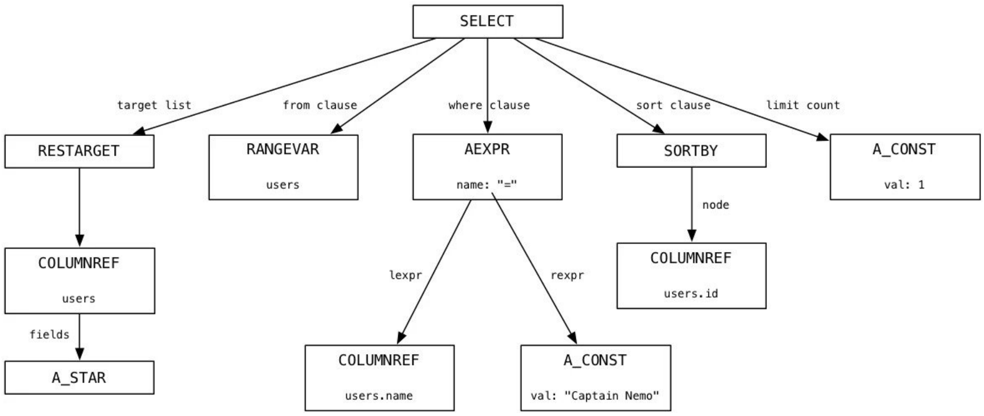

Figure 1.1: Visualisation of parse tree for `"SELECT * FROM users WHERE name = 'Captain Nemo' ORDER BY id ASC LIMIT 1"`

During the *Parse stage*, the parser checks the 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 Postgres 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 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.

- 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 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.[4] Statically typed languages validates the type during compile time and once you compile it, you can run the compiled program multiple times.

On the other hand, dynamic typing does all type checking during runtime, and everytime you want to run the program, it is compiled again. Examples of languages, which use dynamic typing, are Python, Ruby and PHP.

1.2.3 Strong vs. weak typing

Strongly typed languages enforce strict restrictions on intermixing values with different datatypes. Thanks to that, the behaviour is more predictable than it would be for weak typed language. 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 type for every statement.

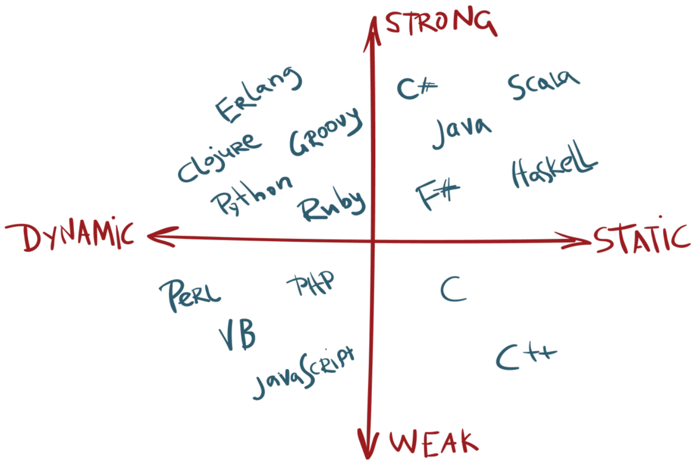

Figure 1.2: Languages divided into groups.[4]

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). [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 defined case classes, where each case class represents one table in the database. 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"*. [6] 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 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.

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.

2.2.2 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.

2.2.3 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].

```c
#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 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.

- `enum NodeTag` - defines all possible types of nodes

- `struct Node` - contains only one variable - the `NodeTag`

Thanks to that guarantee, any type of node can be cast to `Node` without losing the information about the type of the node. 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. A great example of that is the node of type `A_Expr`, where both left side expression and right side expression have `Node*` type.

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

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
```

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.[11] 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
}
```

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
}
```

The parameter list for the generated function contains a `JNIEnv` pointer, a `jobject` pointer, and any Java arguments declared by the Java method.[12]
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.

3.2.3 sbt-jni

Fortunately, `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`. 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 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.

3.3.1 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.

3.3.2 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.

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]`.

3.3.3 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.

3.3.4 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.

3.4 Scala custom interpolators

3.4.1 What are interpolators?

Since version 2.10, Scala offers a new possibility of string interpolation. [13] 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"
PgQueryParser.parse(query)
```

String interpolation

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

3.4.2 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.

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 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 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.* [14] 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.*[15]

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.*[16]

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") }
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))
```

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. [17]

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`.

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"
```

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"
```

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

3.6.2 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.[18] During the validation we also extract the `List[String]` from the `StringContext`.

3.6.3 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.

3.6.4 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`.

3.6.5 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.*[19]

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.

3.6.6 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.

3.6.7 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.*[20] 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: [21]

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 Scalatest

3.7.2 Parser testing

3.7.3 Core testing

3.8 Summary

Since the library is meant as an open-source project, the whole source code is available on *github.com* in a public repository, under the name *pgquery4s*. The project is separated into multiple submodules.

- Native
  This module contains everything related to handling the native `libpg_query` library. From Scala side there is `PgQueryWrapper` class, which implements single method `pgQueryParse` with `@native` annotation. Then there is 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.

3.9 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. [22].

Chapter 4

Conclusion

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] 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

[4] 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

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

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

[7] 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

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

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

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

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

[12] 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/

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

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

[15] 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

[16] 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

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

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

[19] 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

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

[21] 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

[22] 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

Appendix A

Acronyms

API Application programming interface

AST Abstract syntax tree

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
```