COMP2212 - Programming Language Concepts
Coursework Report

CSVQL

THE SIMPLE CSV QUERY LANGUAGE

Group Members:

Alberto Tamajo, at2n19@soton.ac.uk
Giovanni Arcudi, ga1g19@soton.ac.uk
James Channon, jc19g16@soton.ac.uk

University of Southampton

May 7, 2021

Introduction to CSVQL

CSVQL is a domain-specific programming language for querying CSV documents. While the widely used SQL query language has inspired the design of CSVQL, the latter substantially differs from the former for its syntax and rigid type checking capabilities. Thanks to its dynamic variable environment, CSVQL provides a neater syntax than SQL, allowing expressions to be bound to variables. Each variable has a global scope, and it is not final (it can be bound to several different expressions at run-time).

As regards CSVQL query functionalities, these are not limited to the relational algebra's operators as it is possible to manipulate values and construct new ones thanks to the support for typed expressions. The types supported so far are `Bool`, `Int`, `Float` and `String`.

CSVQL also supports `NULL` values. Any operation involving `NULL` values returns a `NULL` value. Thus, the logic system of this programming language is three-valued.

The CSVQL query language allows the users to comment their programs by making use of a commenting style similar to SQL's one: users can comment out a line by using ”- -”.

CSVQL Compiler and Interpreter

CSVQL comes bundled with both a *compiler* and an *interpreter*. The CSVQL compiler runs through the line command and takes as an argument the file path of a CSVQL program. The program is compiled, ran and its output is redirected to `stdout`. The CSVQL interpreter instead runs on a shell allowing the user to write and run CSVQL programs interactively.

CSVQL Logic Units

This programming language consists of two different logic units: CSVQL's **compile-time unit** and CSQL's **run-time unit**.

The CSVQL's **compile-time unit** comprises the CSVQL's *lexer unit*, CSVQL's *parser unit* and CSVQL's *static type checker unit*. While the functionalities of the lexer and the parser are obvious, it is important to mention that if a CSVQL's program is not well-formed, a `ParseException` is thrown, and the user is shown the following statement:

```
Parse error at <Line> :  <Column> => <Corresponding unparsable element>
```

As regards the CSVQL's static type checker, it is concerned with type checking a CSVQL program at compile time. How this task is performed by the static type checker is explained in a more detailed manner in the next section. Suppose a program is considered well-typed by the static type checker. In that case, no type errors can occur at run-time apart from the `NotSingletonTableMultipleRow-sException`, `NotSingletonTableNoRowException` and `ColumnTypeException` (described in the successive section).

The CSVQL's **run time unit** comprises two inter-operable units: CSVQL's *run time executor* and CSVQL's *dynamic type checker*. The CSVQL's run time executor is concerned with evaluating and executing the program. The dynamic type checker is concerned with analysing the tables imported at run-time to enforce correct entry types in each column and the provision of singleton tables (tables having only one column and one entry) when they are expected.

CSVQL's Programs

A CSVQL's program consists of a sequence of sentences separated by a semicolon. Three different types of sentences can be formed are: *Import sentence*, *Variable assignment sentence* and *Query sentence*. In what follows, we are going to briefly describe the semantic of each sentence, how it is type-checked by the CSVQL's static type checker and how it is evaluated and dynamically type-checked by the CSVQL's run time executor and dynamic type checker, respectively.

`IMPORT` sentence semantic

To query a CSV file, the user needs to import it through the `IMPORT` sentence. The `IMPORT` sentence requires the user to specify the path of the desired CSV file, the type of each column and an alias name (whose first character must be an upper-case letter) for the table. Optionally, the user can assign names to the columns of the table being imported to facilitate the writing of the table's queries. If the first line of the CSV file contains a header, the user should use the keyword `WITH HEADER` so that to indicate the evaluator to skip it.

How the `IMPORT` sentence is statically type-checked

The run time evaluation of the `IMPORT` sentence consists of reading the indicated CSV file and converting the content into a list of lists. If the file does not exist or cannot be open, an `IOException` is thrown. Otherwise, the dynamic type checker checks whether each entry matches the type of its column. If this is not the case, a `ColumnTypeException` is thrown. Otherwise, the following pair is added to the run time table environment `(table_alias_name,(header,col_types, table_content))`.

Variable assignment sentence semantic

A variable assignment sentence allows the user to bound an expression to a variable. The variable's first character needs to be a lower-case letter. In CSVQL, an expression can be either of the following: *variable*, *query*, *string expression*, *boolean expression*, *arithmetic expression*, *if-then-else expression*. It is crucial to note that both string, boolean, and arithmetic expressions can contain queries inside as these can output a string, boolean or numeric value, respectively.

How the variable assignment sentence is statically type-checked

The static type checker assigns the type `Void` to the whole variable assignment sentence as it does not return a table value. The static type checker also adds to the static type environment the following pair `(var_name, type_expression)`, indicating that the variable `var_name` is bound to an expression whose type is `type_expression`.Thus, the static type checker evaluates the type of the expression being bound to the variable `var_name`.

It is important to notice that the queries inside a string, boolean or arithmetic expression need to output singleton tables (tables having only one column with a unique entry). The static type checker can only check whether the table returned by a query has only one column, as whether the column contains one or more entries depends on the nature of the tables imported from the CSV files. In case an expected singleton table contains more than one column, a `NonSingletonTableInExpressiomException` is thrown at run-time.

How the variable assignment sentence is evaluated and dynamically type-checked

The run time evaluation of a variable assignment sentence consists of adding the following pair to the run time variables environment `(var_name, expression)`. CSVQL does not adopt an eager evaluation strategy but rather a **lazy evaluation** strategy. Thus, an expression bound to a variable is evaluated only when its value is needed, which occurs when the variable is used inside a query sentence. When the expression bound to a variable is evaluated, and a query expected to output a singleton table does not do so, two kinds of exceptions can be thrown: `NotSingletonTableMultipleRowsException`, `NotSingletonTableNoRowException`.

Query sentence semantic

The CSVQL's query system provides a wide range of SQL functionalities to the user, though with a different syntax. The CSVQL query functionalities and the corresponding syntax are listed in the Appendix. The fundamental operator that allows a user to query a table is `GET`. The `GET` operator behaves like a function that takes several arguments and returns a table. Compulsory arguments of the `GET` operator are: (1) a table to be manipulated and (2) a sequence of columns to include in the output table.

The table to be manipulated argument can be either an `alias_name` (previously assigned through an `IMPORT` statement), a nested query or a variable holding a query expression. As regards the sequence of columns argument, each column can be either a column reference, an expression, an aggregate function or binary operations among the latter. Each column can be assigned a column alias name. It is crucial to understand that in CSVQL, a column reference can be either an integer index or a name indicating a column of the table to be manipulated. In case a user needs to extract all columns from a table, CSVQL provides a wildcard character `*`, which replaces all columns of the table.

Optional arguments of the `GET` operator are: `DISTINCT`, `WHERE`. The semantics of `DISTINCT` is the same as the homonymous operator in SQL. As regards the `WHERE` operator, it is followed by one or more filters. Such filters are divided into two categories: (1) Row filters, (2) GroupOrderLimit filters.
It is necessary that the GroupOrderLimit filters follow the Rowfilters, but there are no constraints in the order filters of the same category need to have. The several filters that can be used in each category are listed in the appendix's table.

Unlike SQL, CSVQL offers the user the capability to transform a table into a single value through aggregation functions. In other words, it is possible to give as input to an aggregate function a query that must return a table containing a single column.

How the query sentence is statically type checked

The static type checker assigns a type to a query according to the type of the returned table. In other words, if a query returns a table with header `x` and column types `y` then the type of the query is `Table x y`. In case a query is not matched with a type, a type error is present inside the query, and a static type exception is thrown as a consequence. An exhaustive list of static type exceptions is located in the Appendix.

CSVQL compiler mode of execution

The mode of execution of the CSVQL compiler is the following:

1. The file program provided as argument is imported.
2. The CSVQL's lexer lexes the source code.
3. The sequence of lexemes is parsed by the CSVQL's parser. If a parse error is present in the sequence of lexemes, a parse error is output to `stderr`.
4. The CSVQL's static type checker checks whether the parsed content is well-typed. If this is not the case, a static type checker exception is thrown and a message is output to `stderr`.
5. The CSVQL's run time executor runs the parsed content in a top-down left-to-right fashion. Any run-time output is redirected to `stdout`. If a run-time type checker exception occurs, an error is output to `stderr`.

Appendix

Type Checker Exceptions and corresponding `stderr` errors

HeaderException `tName head types` The declared header for table `tName` is `head`. The declared types for table `tName` are `types`. The length of the declared header for table `tName` is `length head` while the length of the declared types is `length types`. The length of the header and the types should be equivalent
NonExistingTableException `tName` "The table `tName` has not been defined prior to its use."
NonSingletonTableInAggregation `aggreFun tType` "The argument of the aggregate function `aggreFun` is not a singleton table as it contains more than one column. Indeed, its type is `tType`."
UnionException `tType1 tType2` "The arguments of the UNION function do not have the same type. The first argument has type `tType1`. The second argument has type `tType2`."
MergeException `tType1 tType2` "The arguments of the MERGE function do not have the same type. The first argument has type `tType1`. The second argument has type `tType2`."
NonQueryVariable `varName t` "The variable `varName` is supposed to be a query. However, its type is `t`."
NonExistingQueryVariable `varName` "The variable `varName` is supposed to be a query. However, it has not been defined prior to its use."
NonExistingColumnInt `n` "The column `n` does not exist in the queried table."
NonExistingColumnName `xs` "The column `xs` does not exist in the queried table."
DuplicatedColumnInGetException `xs` "The column `xs` is instantiated twice inside the GET function. However, it is possible to instantiate a column only once."
NonSingletonTableInExpression `tType` "Tables used inside expressions need to be singleton. However, a table having type `tType` is used."
NonExistingVariable `xs` "The variable `xs` has not been defined prior to its use."
WrongColumnTypeInAggregation `aggreFun t` "The `aggreFun` aggregate function can be performed only with a column having type Int or Float. However, it is used for a column having type `t`."
IncorrectArgsStringOperatorException `strOp t1 t2` "The first argument of the string operator `strOp` has type `t1` . The second argument has type `t2`. However, the string operator `strOp` takes two arguments of type String."
IncorrectArgsBiLogicOpException `biLogOp t1 t2` "The first argument of the binary logic operator `biLogOp` has type `t1`. The second argument has type `t2`. However, the binary logic operator`biLogOp` takes two arguments of type String."
IncorrectArgsCompOpException `compOp t1 t2` "The first argument of the comparison operator `compOp` has type `t1`. The second argument has type `t2`. However, the comparison operator `compOp` takes two arguments of the same type."
IncorrectArgsArithOpException `arithOp t1 t2` "The first argument of the arithmetic operator `arithOp` has type`t1`. The second argument has type `t2`. However, the arithmetic operator `arithOp` takes two arguments of type Int or Float."
IncorrectArgUnaryLogicOpException `unLogicOp t1` "The argument of the unary logic operator `unLogicOp` has type `t1`. However, the unary logic operator `unLogicOp` takes an argument of type Bool."
IncorrectArgsIfThenElseCol `type1 type2` "The first expression in the If-Then-Else statement has type `type1`. The second expression has type `type2`. However, the expressions in the If-Then-Else statement should have the same type."

SQL Syntax vs CSVQL Syntax

SQL Syntax	CSVQL Syntax
SELECT column1, column2 FROM table_name WHERE condition;	GET table_name('column1', 'column2') WHERE condition
SELECT * FROM table_name	GET table_name(*)
SELECT DISTINCT column1, column2 FROM table_name WHERE condition;	GET DISTINCT table_name('column1', 'column2') WHERE condition
SELECT column1, column2 FROM table_name WHERE condition1 AND condition2 AND condition3;	GET table_name('column1', 'column2') WHERE condition1 && condition2 && condition3
SELECT column1, column2 FROM table_name WHERE condition1 OR condition2 OR condition3;	GET table_name('column1', 'column2') WHERE condition1 || condition2 || condition3
SELECT column1, column2 FROM table_name WHERE NOT condition;	GET table_name('column1', 'column2') WHERE NOT condition
SELECT column1, column2 FROM table_name ORDER BY column1, column2, [ASC|DESC];	GET table_name('column1', 'column2') ORDERBY([ASC|DESC], 'column1', 'column2')
SELECT column1, column2 FROM table_name WHERE condition ORDER BY column1, column2, [ASC|DESC];	GET table_name('column1', 'column2') WHERE condition, ORDERBY([ASC|DESC], 'column1', 'column2')
SELECT column1, column2 FROM table_name WHERE column1 IS NULL;	GET table_name('column1', 'column2') WHERE EMPTY('column1')
SELECT column1, column2 FROM table_name WHEREcolumn1 IS NOT NULL;	GET table_name('column1', 'column2') WHERE NOTEMPTY('column1')
SELECT column1, column2 FROM table_name WHERE condition LIMIT number;	GET table_name('column1', 'column2') WHERE condition, FIRST(number)
SELECT MIN(column1) FROM table_name WHERE condition;	GET table_name(MIN('column1')) WHERE condition
SELECT MAX(column1) FROM table_name WHERE condition;	GET table_name(MAX('column1')) WHERE condition
SELECT COUNT(column1) FROM table_name WHERE condition;	GET table_name(COUNT('column1')) WHERE condition
SELECT AVG(column1) FROM table_name WHERE condition;	GET table_name(AVG('column1')) WHERE condition
SELECT SUM(column1) FROM table_name WHERE condition;	GET table_name(SUM('column1')) WHERE condition
SELECT column1, column2 FROM table_name WHERE column1 IN (value1, value2);	GET table_name('column1', 'column2') WHERE column1 SATISFIES <comparisonOperator> (expression1, expression2)
SELECT column1, column2 FROM table_name WHERE column1 IN (<SELECT STATEMENT>);	
SELECT column1, column2 FROM table_name WHERE column1 BETWEEN value1 AND value2;	GET table_name('column1', 'column2') WHERE BETWEEN('column1', value1, value2)
SELECT column1 AS alias_name FROM table_name;	GET table_name('column1' AS alias_name)
SELECT column1, column2 FROM table_name AS alias_name;	alias_name := GET table_name('column1', 'column2')

SELECT Orders.C1, Customers.C2 FROM Orders **INNER JOIN** Customers **ON** Orders.C1=Customers.C2;	GET Orders('C1') **INNER PRODUCT** GET Customers('C2') **GIVEN** 'C1'='C2'
SELECT column1, column2 FROM table1 **UNION** SELECT column3, column4 FROM table2;	GET table1('column1', 'column2') **UNION** GET table2('column3', 'column4')
SELECT column1, column2 FROM table1 **UNION ALL** SELECT column3, column4 FROM table2;	GET table1('column1', 'column2') **MERGE** GET table2('column3', 'column4')
SELECT column1, column2 FROM table_name WHERE condition **GROUP BY** column1 **ORDER BY** column2;	GET table_name('column1', 'column2') WHERE condition, **GROUPBY**('column1'), **ORDERBY**([ASC|DESC], column2)
CREATE PROCEDURE SelectAllCustomers **AS** SELECT * FROM Customers GO; **EXEC SelectAllCustomers**;	selectAllCustomers **:=** GET Customers (*); **selectAllCustomers**
– – Commenting in SQL.	**– –** Commenting in CSVQL.
CREATE TABLE table_name (column1 datatype1, column2 datatype2, column3 datatype3);	**IMPORT** '<filepath>/file.csv' :: (datatype1, datatype2, datatype3) ('column1', 'column2', 'column3') **AS** table_name
SELECT CONCAT("SQL ", "is ", "fun!") **AS** ConcatenatedString;	concatenatedString **:=** "CSVQL" **CONCAT** "is" **CONCAT** "better!"; concatenatedString
SELECT IF(500<1000, "YES", "NO");	**x := IF** 500<1000 **THEN** "YES" **ELSE** "NO"; x

CSVQL Interpreter Screenshot

Note that in the screenshot below "SQL-LIKE" is the previous name that we had given to our language.