# Type Checking SQL Queries

CAMILLE BOSSUT
JOSEPH KOMSKIS
ISAAC WEINTRAUB

The ability to statically type-check SQL queries can have both security and ease of development benefits. In order to statically type-check a language, one must formalize its syntax and typing rules. This paper defines an abstract syntax and typing rules for a subset of SQL, including integer, boolean, and string expressions, CREATE TABLE statements, SELECT-FROM-WHERE queries, GROUP BY and HAVING clauses, joins, unions, and intersections. We then use this syntax and these rules to implement a parser and type-checker in Python to demonstrate the soundness of our design. Finally, we sketch a proof of the progress and preservation theorems for part of our SQL subset.

## 1 INTRODUCTION

Static type checking provides several benefits to a language. It allows one to avoid classes of bugs in a program without having to execute it. This is not only useful in compiled languages; there is considerable practical interest in introducing static type systems to dynamic languages such as Python type annotations [4] and TypeScript [3]. These type systems help developers catch mistakes, such as accessing the incorrect field of a variable, and strengthens the capabilities of IDEs and other tooling [3, 4].

When writing SQL queries, most development environments will not inform the developer that a query fails to type-check. Instead, the developer will only learn of the error when they try to run the query on the DBMS. Adding static type checking would give developers the ability to type-check their queries before submitting them to a DBMS. Caldwell and Roan [5] also list security as a benefit of type-checking SQL queries. For example, intruders can submit queries to a DBMS and use any returned errors to infer what tables and fields may exist. If a type-checker was deployed in front of a DBMS, it could reject these types of queries without leaking information about the database catalog and report them to administrators.

Our syntax and typing rules are based on those laid out in [5]. However, we make several extensions to their syntax and rules. First, we allow expressions in the SELECT clause of a query rather than only allowing field names. Second, we differentiate between statements and queries in our syntax, which prevents certain invalid programs such as having a CREATE TABLE statement in the FROM clause of a query. Finally, we add support for the GROUP BY and HAVING clauses, aggregation operators, UNIONs, and INTERSECTs.

The rest of the paper is laid out as follows: Section 2 defines the abstract syntax for our subset of SQL. Section 3 describes the typing rules for our subset of SQL. In section 4 we document the implementation status, our implementation approach, and our experiences implementing the syntax and typing rules. Section 5 lays out a proof of the progress and preservation theorems for part of our SQL subset. Section 6 describes future opportunities for the project. Finally, section 7 concludes the report.

## 2 ABSTRACT SYNTAX

Before we can lay out typing rules, we first define an abstract syntax for our subset of the SQL language. This syntax is largely based on [6] and [5], making simplifications where it is reasonable to do so. As in [5], we use $A^*$ to denote 0 or more occurrence of a class $A$ and $A^+$ to denote 1 or

Authors' addresses: Camille Bossut, cbossut21@gatech.edu; Joseph Komskis, jkomskis3@gatech.edu; Isaac Weintraub, weintraub@gatech.edu.

more occurrences of a class $A$. To simplify the typing rules, we assume all tables will have distinct names, unlike regular SQL which allows nested queries to produce anonymous tables. We also assume all names, such as column names, will be fully qualified, unlike regular SQL which can infer the table name of a unique column name.

| | | |
|---|---|---|
| Identifier $I$ | ::= $I$ | *Base identifier* |
| Int $i$ | ::= $i$ | *Numbers* |
| String $s$ | ::= $s$ | *Strings* |
| Bool $b$ | ::= true \| false | *Boolean values* |
| Type $\theta$ | ::= BOOL \| INT \| VARCHAR | *Base datatypes* |
| TName $TN$ | ::= $I$ | *Table name* |
| CName $CN$ | ::= $TN.I$ | *Field name* |
| TElement $TE$ | ::= $I\ \theta$ | *Table element* |
| Aggr $AG$ | ::= MIN \| MAX \| AVG \| COUNT | *Aggregation operators* |
| Expr $E$ | ::= $CN$ | *Expressions* |
| | \| AG(E) | |
| | \| $b$ | |
| | \| $i$ | |
| | \| $s$ | |
| | \| $E_1$ AND $E_2$ | |
| | \| NOT $E$ | |
| | \| $E_1 = E_2$ | |
| | \| $E_1 < E_2$ | |
| | \| $E_1 + E_2$ | |
| | \| $E_1 \times E_2$ | |
| | \| CONCAT$(E_1, E_2)$ | |
| | \| SUBSTR$(E_1, E_2, E_3)$ | |
| SExpr $SE$ | ::= $E$ [AS $I$] | *Select expressions* |
| Query $Q$ | ::= $TN_1$ [AS $TN_2$] | *Query table contents* |
| | \| $Q_1$ JOIN $Q_2$ ON $BE$ AS $TN$ | *Query joined tables* |
| | \| SELECT $SE^+$ FROM $Q$ [WHERE $BE$] [GROUP BY $E_g^+$ [HAVING $E_h$]] | *Select query* |
| | \| $Q_1$ UNION $Q_2$ | *Union query* |
| | \| $Q_1$ INTERSECT $Q_2$ | *Intersection query* |
| Stmt $S$ | ::= CREATE TABLE $TN$ ( $TE^+$ ) | *Create table statement* |
| | \| $Q$ | *Query statement* |
| | \| $S_1$ ; $S_2$ | *Sequence of statements* |

## 3 TYPING RULES

Just as statements in a functional programming language work in the context of a set of variables and their types, statements in SQL work in the context of a set of tables and their schemas. Let $\Gamma = \{I_1 : \sigma_1\ ,\ I_2 : \sigma_2\ ,\ \ldots\ ,\ I_n : \sigma_n\}$ be a list of typing assumptions. Each assumption $I_j : \sigma_j$ in the list denotes that the table with unique identifier $I_j$ has the schema $\sigma_j$.

Each schema, in turn, is a tuple of the columns and their types that make up a table. Let $\sigma = \langle I_1 : \theta_1\ ,\ I_2 : \theta_2\ ,\ \ldots\ ,\ I_k : \theta_k \rangle$ be a schema. Each column or field $I_j : \theta_j$ in the tuple denotes the $j$th column has the unique identifier $I_j$ (within its schema) and type $\theta_j$. For some typing rules, such as typing expressions, we will need the ability to concatenate schemas. We use the notation $\sigma @ \sigma'$ to denote the concatenation of schemas $\sigma$ and $\sigma'$.

*3.0.1 Types.* Types are typed as follows:

$$\frac{}{\Gamma \vdash \text{INT} : \text{Type}}\ \text{ty-INT} \qquad \frac{}{\Gamma \vdash \text{BOOL} : \text{Type}}\ \text{ty-Bool}$$

$$\frac{}{\Gamma \vdash \text{VARCHAR} : \text{Type}}\ \text{ty-VARCHAR}$$

*3.0.2 Tables.* A table that appears in the typing assumptions $\Gamma$ can be easily typed by the schema bound to it in $\Gamma$:

$$\frac{}{\Gamma_1,\ I : \sigma,\ \Gamma_2 \vdash I : \sigma}\ \text{ty-table}$$

*3.0.3 Columns.* A column name type checks if it's table $TN$ appears in the typing assumptions and the column's name appears in $TN$'s schema:

$$\frac{\Gamma \vdash I_1 : \sigma \qquad I_2 : \theta \in \sigma}{\Gamma \vdash I_1.I_2 : \theta}\ \text{ty-column}$$

*3.0.4 Expressions.* Since expressions in SQL are written in the context of a schema, the most natural to way to model them is as functions that take input tuples of some schema and return values of some type. When an expression is executed, values representing each tuple are provided for the schema and a single value is returned.

Using a column name as an expression is interesting, as it expects input values of the same type as the column, and returns the values as output:

$$\frac{\Gamma \vdash I : \theta}{\Gamma \vdash I : \langle I : \theta \rangle \rightarrow \theta}\ \text{ty-column-expr}$$

The rules for the other expressions are as follows:

$$\frac{}{\Gamma \vdash b : \langle\rangle \rightarrow \text{BOOL}}\ \text{ty-bool-expr}$$

$$\frac{\Gamma \vdash E_1 : \sigma_1 \to \mathsf{BOOL} \qquad \Gamma \vdash E_2 : \sigma_2 \to \mathsf{BOOL}}{\Gamma \vdash E_1 \ \mathsf{AND}\ E_2 : \sigma_1 @ \sigma_2 \to \mathsf{BOOL}}\ \text{ty-and}$$

$$\frac{\Gamma \vdash E : \sigma \to \mathsf{BOOL}}{\Gamma \vdash \mathsf{NOT}\ E : \sigma \to \mathsf{BOOL}}\ \text{ty-not}$$

$$\frac{\Gamma \vdash E_1 : \sigma_1 \to \theta \qquad \Gamma \vdash E_2 : \sigma_2 \to \theta \qquad \theta \text{ is INT, BOOL, or VARCHAR}}{\Gamma \vdash E_1 = E_2 : \sigma_1 @ \sigma_2 \to \mathsf{BOOL}}\ \text{ty-equal}$$

$$\frac{\Gamma \vdash E_1 : \sigma_1 \to \theta \qquad \Gamma \vdash E_2 : \sigma_2 \to \theta \qquad \theta \text{ is INT or VARCHAR}}{\Gamma \vdash E_1 < E_2 : \sigma_1 @ \sigma_2 \to \mathsf{BOOL}}\ \text{ty-lt}$$

$$\frac{}{\Gamma \vdash i : \langle\rangle \to \mathsf{INT}}\ \text{ty-int-expr}$$

$$\frac{\Gamma \vdash E_1 : \sigma_1 \to \ \mathsf{INT} \qquad \Gamma \vdash E_2 : \sigma_2 \to \ \mathsf{INT}}{\Gamma \vdash E_1 + E_2 : \sigma_1 @ \sigma_2 \to \mathsf{INT}}\ \text{ty-sum}$$

$$\frac{\Gamma \vdash E_1 : \sigma_1 \to \ \mathsf{INT} \qquad \Gamma \vdash E_2 : \sigma_2 \to \ \mathsf{INT}}{\Gamma \vdash E_1 \times E_2 : \sigma_1 @ \sigma_2 \to \ \mathsf{INT}}\ \text{ty-product}$$

$$\frac{}{\Gamma \vdash s : \langle\rangle \to \mathsf{VARCHAR}}\ \text{ty-varchar-expr}$$

$$\frac{\Gamma \vdash E_1 : \sigma_1 \to \ \mathsf{VARCHAR} \qquad \Gamma \vdash E_2 : \sigma_2 \to \ \mathsf{VARCHAR}}{\Gamma \vdash \mathsf{CONCAT}(E_1, E_2) : \sigma_1 @ \sigma_2 \to \mathsf{VARCHAR}}\ \text{ty-concat}$$

$$\frac{\Gamma \vdash E_1 : \sigma_1 \to \ \mathsf{VARCHAR} \qquad \Gamma \vdash E_1 : \sigma_2 \to \ \mathsf{INT} \qquad \Gamma \vdash E_2 : \sigma_3 \to \ \mathsf{INT}}{\Gamma \vdash \mathsf{SUBSTR}(E_1, E_2, E_3) : \sigma_1 @ \sigma_2 @ \sigma_3 \to \mathsf{VARCHAR}}\ \text{ty-substr}$$

*3.0.5 Typing Queries.* The simplest form of a query is a single table name:

$$\frac{\Gamma \vdash TN : \sigma}{\Gamma \vdash TN : \{TN : \sigma\}}\ \text{ty-table-query}$$

Optionally, we can also rename the table used in a query:

$$\frac{\Gamma \vdash TN_1 : \sigma \qquad TN_2 \notin \Gamma}{\Gamma \vdash TN_1\ \mathsf{AS}\ TN_2 : \{TN_2 : \sigma\}}\ \text{ty-table-rename-query}$$

To join two tables, we require that the boolean expression used to join them works on the concatenation of their schemas, and results in a single table whose schema is the concatenation of its component tables:

$$\frac{\Gamma \vdash Q_1 : \sigma_1 \qquad \Gamma \vdash Q_2 : \sigma_2 \qquad \Gamma \vdash E : \sigma_1 @ \sigma_2 \to \mathsf{BOOL}}{\Gamma \vdash Q_1\ \mathsf{JOIN}\ Q_2\ \mathsf{ON}\ E\ \mathsf{AS}\ TN : \{TN : \sigma_1 @ \sigma_2\}}\ \text{ty-join}$$

For select queries, we require from our abstract syntax that the source query $Q$ have a output table name. This differs from standard SQL, which allow anonymous tables. In a select query, each expression, which results in a field in the output table, must use the schema of the source query:

$$\frac{\Gamma \vdash Q : \{TN : \sigma\} \qquad \Gamma \vdash SE_i : \sigma \to \theta_i}{\Gamma \vdash \mathsf{SELECT}\ SE^+\ \mathsf{FROM}\ Q : \{TN : \langle SE_0 : \theta_0, SE_1 : \theta_1, \cdots \rangle\}}\ \text{ty-select}$$

If a where clause is provided, it can reference columns from the original query or from the new columns created in the select portion of the query:

$$\frac{\begin{array}{c}\Gamma \vdash Q : \{TN : \sigma_1\} \\ \Gamma \vdash \mathsf{SELECT}\ SE^+\ \mathsf{FROM}\ Q : \{TN : \sigma_2\} \qquad \Gamma \vdash E : \sigma_1 @ \sigma_2 \to \mathsf{BOOL}\end{array}}{\Gamma \vdash \mathsf{SELECT}\ SE^+\ \mathsf{FROM}\ Q\ \mathsf{WHERE}\ E : \{TN : \sigma_2\}}\ \text{ty-select-where}$$

When unioning or intersecting two queries, the queries must have the same number of columns and the same data type in corresponding columns, but the field names may differ. The resulting query will inherit the table names of both queries and field names of the left subquery:

$$\frac{\Gamma \vdash Q_1 : \{TN_1 : \sigma_1\} \qquad \Gamma \vdash Q_2 : \{TN_2 : \sigma_2\} \qquad \sigma_1.\theta_0 = \sigma_2.\theta_0,\ \sigma_1.\theta_1 = \sigma_2.\theta_1,\ \cdots}{\Gamma \vdash Q_1\ \mathsf{UNION}\ Q_2 : \{TN_1\_TN_2 : \sigma_1\}}\ \text{ty-union}$$

$$\frac{\Gamma \vdash Q_1 : \{TN_1 : \sigma_1\} \qquad \Gamma \vdash Q_2 : \{TN_2 : \sigma_2\} \qquad \sigma_1.\theta_0 = \sigma_2.\theta_0,\ \sigma_1.\theta_1 = \sigma_2.\theta_1,\ \cdots}{\Gamma \vdash Q_1\ \mathsf{INTERSECT}\ Q_2 : \{TN_1\_TN_2 : \sigma_1\}}\ \text{ty-intersect}$$

*3.0.6 Aggregation.* One may provide a group by clause containing expressions on the original schema. In this case, the output may only contain the grouped-by expressions or aggregation functions. The Agg $\sigma$ notation formalizes this by declaring an aggregation to have a different type than the original schema.

$$\frac{\Gamma \vdash Q : \{TN : \sigma\} \qquad \sigma <: \sigma_g \qquad \Gamma \vdash SE_i : \text{Agg } \sigma \to \theta_i \qquad \Gamma \vdash E : \sigma \to \text{BOOL} \qquad \Gamma \vdash E_{g_i} : \sigma_g \to \theta_i}{\Gamma \vdash \textsc{Select } SE^+ \textsc{ From } Q \textsc{ Where } E \textsc{ Group By } E_g^+ : \{TN : \langle SE_0 : \theta_0, SE_1 : \theta_1, \cdots \rangle\}} \text{ ty-group-by}$$

If a having clause is provided, it also may only contain the grouped-by expressions or aggregation functions. As a simplification, new columns created in the select portion of the query cannot be referenced subsequently; standard SQL would allow the new columns to be referenced by the group by and having clauses.

$$\frac{\Gamma \vdash Q : \{TN : \sigma\} \qquad \sigma <: \sigma_g \qquad \Gamma \vdash SE_i : \text{Agg } \sigma \to \theta_i \qquad \Gamma \vdash E : \sigma \to \text{BOOL} \qquad \Gamma \vdash E_{g_i} : \sigma_g \to \theta_i \qquad \Gamma \vdash E_h : \text{Agg } \sigma \to \text{BOOL}}{\Gamma \vdash \textsc{Select } SE^+ \textsc{ From } Q \textsc{ Where } E \textsc{ Group By } E_g^+ \textsc{ Having } E_h : \{TN : \langle SE_0 : \theta_0, \cdots \rangle\}} \text{ ty-having}$$

Likewise, aggregation functions take Agg $\sigma$ instead of $\sigma$:

$$\frac{\Gamma \vdash E : \sigma \to \theta \qquad \theta \text{ comparable with } <}{\Gamma \vdash \text{MIN}(E) : \text{Agg } \sigma \to \theta} \text{ ty-min}$$

$$\frac{\Gamma \vdash E : \sigma \to \theta \qquad \theta \text{ comparable with } <}{\Gamma \vdash \text{MAX}(E) : \text{Agg } \sigma \to \theta} \text{ ty-max}$$

$$\frac{\Gamma \vdash E : \sigma \to \text{ INT}}{\Gamma \vdash \text{AVG}(E) : \text{Agg } \sigma \to \text{ INT}} \text{ ty-avg}$$

$$\frac{\Gamma \vdash E : \sigma \to \theta \qquad \theta \text{ comparable with } =}{\Gamma \vdash \text{COUNT}(E) : \text{Agg } \sigma \to \text{INT}} \text{ ty-count}$$

Operators are also overloaded so that aggregation functions can be used in computation, e.g.:

$$\frac{\Gamma \vdash E_1 : \text{Agg } \sigma_1 \to \theta \qquad \Gamma \vdash E_2 : \text{Agg } \sigma_2 \to \theta \qquad \theta \text{ is INT or VARCHAR}}{\Gamma \vdash E_1 < E_2 : \text{Agg } \sigma_1 @ \sigma_2 \to \text{BOOL}} \text{ ty-agg-lt}$$

$$\frac{\Gamma \vdash E_1 : \text{Agg } \sigma_1 \to \text{ INT} \qquad \Gamma \vdash E_2 : \text{Agg } \sigma_2 \to \text{ INT}}{\Gamma \vdash E_1 + E_2 : \text{Agg } \sigma_1 @ \sigma_2 \to \text{INT}} \text{ ty-agg-sum}$$

Grouped-by expressions may be selected without an aggregation function.

$$\frac{\Gamma \vdash E : \sigma \to \theta \qquad \sigma_g <: \sigma}{\Gamma \vdash E : \text{Agg } \sigma \to \theta} \text{ ty-grouped-expr}$$

*3.0.7 Typing statements.* When a query is used as a statement, it follows the same typing rules as in the previous section. A CREATE TABLE statement type checks as long as each field/type pair is valid:

$$\frac{\Gamma \vdash I_i : \theta_i}{\Gamma \vdash \text{CREATE TABLE } TN \text{ ( } TE^+ \text{ )} : \{TN : \langle I_0 : \theta_0, I_1 : \theta_1, \cdots \rangle\}} \text{ ty-create-table}$$

When type checking a sequence of statements, the first statement in the sequence always starts from an empty list of assumptions, and returns the type of the last statement:

$$\frac{\vdash S_1 : \{TN_1 : \sigma_1\} \qquad \Gamma \vdash S_n : \{TN_n : \sigma_n\}}{\vdash S_1 \text{ ; } \cdots \text{ ; } S_n : \{TN_n : \sigma_n\}} \text{ ty-sequence}$$

The statements after the first assume typing assumptions $\Gamma$ based on the statements that preceded them. Most statements do not change the typing assumptions used in later statements. However, when a create table statement $CT$ which creates a table $TN$ with schema $\sigma$ is type-checked in a statement, later statements can make use of it (denoted by $\cup$) in their typing assumptions:

$$\frac{\Gamma \vdash CT : \{TN : \sigma\} \qquad \Gamma \cup \{TN : \sigma\} \vdash S_i : \{TN_i : \sigma_i\}}{\vdash \cdots \text{ ; } CT \text{ ; } S_i : \{TN_i : \sigma_i\}} \text{ ty-seq-create}$$

$$\frac{\Gamma \vdash Q : \{TN : \sigma\} \qquad \Gamma \vdash S_i : \{TN_i : \sigma_i\}}{\vdash \cdots \text{ ; } Q \text{ ; } S_i : \{TN_i : \sigma_i\}} \text{ ty-seq-query}$$

*3.0.8 Subtyping.* Expressions such as those used in select, where or join-on clauses must be well-typed with respect to the entire query's schema, but it is valid for an expression to only use a subset of the schema available to it. In general, a schema $\sigma'$ can be substituted for another schema $\sigma$ as long as $\sigma \subseteq \sigma'$. This can be formalized as a subtyping relation on schemas:

$$\frac{}{\{I_0 : \theta_0, \cdots, I_n : \theta_n, I_{n+1} : \theta_{n+1}\} <: \{I_0 : \theta_0, \cdots, I_n : \theta_n\}} \text{ ty-sub}$$

Since expressions are modeled as functions, they are contravariant with respect to their input schemas:

$$\frac{\sigma_1 <: \sigma_2}{\sigma_2 \to \theta <: \sigma_1 \to \theta} \text{ ty-expr-sub}$$

## 4 IMPLEMENTATION

We implemented the syntax and typing rules using Python, as we were all familiar with it, and it is well-suited for efficiently implementing the syntax and rules.

Our approach was to use a parsing library to parse SQL programs into Abstract Syntax Trees. Each node of the AST is a user-defined object with 0 or more children. We then added methods to each of these user defined objects, so that when we type check on the root of the tree, our type-checking algorithm will proceed down the tree. Each node of the AST is responsible for ensuring its children type checking and returning intermediate information to the parent as needed.

We won't describe the type checking behavior for every type of node in the AST, since most follow from the typing rules, but here we will describe the type-checking algorithm for `SELECT-FROM-WHERE` queries. First, we type check the source query specified in the `FROM` clause, getting its name and schema. Then, for each expression in the `SELECT` clause, we ensure each one type checks in the context of the schema of the source query. Then, we build a new schema from the concatenation of the source query and the select expressions. Next, we check the condition in the `WHERE` clause type checks and has a boolean output in the context of the concatenated schema. If a `GROUP BY` clause is specified, we check each expression provided type checks under the concatenated schema. We then go back to the expressions in the `SELECT` clause and ensure all of them are aggregated or based on the columns on which we are grouping. If a `HAVING` clause is specified, we also type check it and ensure it is an aggregated expression. If a `WHERE` clause is specified, we ensure it is not an aggregated expression. Finally, once all of our checks pass, we can return the output schema of the query. For a `SELECT-FROM-WHERE` query, the output schema is the field names and output types for each of the `SELECT`ed expressions.

When a program fails to type check, our implementation throws one of several errors to describe the nature of the problem. A `ParseError` is thrown when a program is not syntactically valid. A `KeyError` is thrown when a program tries to reference a table or field which it is not permitted to access. For example, a `KeyError` is thrown when a `SELECT` or `WHERE` clause tries to access a field from a table not being selected. A `TypeMismatchError` is thrown when an expression has a different type than expected. For example, this error is thrown when the expression of a `WHERE` clause returns an integer or string rather than a boolean. A `RedefinedNameError` is thrown when one tries to create a table with a name that has already been used. An `AggregationMismatchError` is thrown when an expression is aggregated in a context where only a non-aggregated expression is allowed or vice versa. For example, this error is thrown when an aggregation operator is used in a `WHERE` clause or a non-aggregated column is used in a `HAVING` clause.

### 4.1 Implementation Status

Our type checker is available on GitHub Enterprise at https://github.gatech.edu/iweintraub3/sql-typeck. At this time we have fully implemented the syntax and typing rules described above. We wrote unit tests to ensure our implementation behaved as expected and to prevent regressions. As a starting point, we recommend looking at the files in `test/e2e`, which provide high-level unit tests that demonstrate the types of queries we support, the types of errors we can catch, and the output of well-typed programs.

### 4.2 Parsing with Parsy

We used Parsy [2], a parser-combinator library for Python, to encode our abstract syntax. Our primary motivation for using a parser-combinator over a parser-generator was that we were less familiar with parser-combinators and wanted to gain experience using one. We chose Parsy in particular because it seemed to have the easiest and most natural API. Parsy worked quite well for

our purposes. Using Parsy's decorators and Python's generators made it easy to construct objects and build our AST while parsing. However, using a parser-combinator did result in some challenges during development. First, left-recursive grammars cause Parsy to recurse until a stack overflow occurs. Therefore, we had to take care when writing our expression and query parsing rules to avoid left-recursion. Additionally, we originally had separate syntax classes for integer, boolean, varchar, and generic expressions. This worked well in most cases, but failed when trying to parse a column name, since the Parser could not predict what type a column would have. This required us to refactor these different types of expressions into a single syntax class.

## 5 PROGRESS AND PRESERVATION

We will sketch proofs of progress and preservation for a subset of our subset of SQL. While the proofs are intentionally limited in scope and rigor, they demonstrate at a high level how to prove progress and preservation for our type system as a whole. We will focus on a subset of the language with the following syntax:

| | | |
|---|---|---|
| Identifier $I$ | $::= I$ | *Base identifier* |
| Int $i$ | $::= i$ | *Numbers* |
| String $s$ | $::= s$ | *Strings* |
| Bool $b$ | $::=$ true \| false | *Boolean values* |
| Val $v$ | $::= b \mid i \mid s$ | *Values* |
| Type $\theta$ | $::=$ BOOL \| INT \| VARCHAR | *Base datatypes* |
| TElement $TE$ | $::= I\ \theta$ | *Table element* |
| MiniExpr $E$ | $::= I$ | *Expressions* |
| | $\mid\ v$ | |
| | $\mid\ E_1$ AND $E_2$ | |
| | $\mid\ E_1 = E_2$ | |
| | $\mid\ E_1 < E_2$ | |
| | $\mid\ E_1 + E_2$ | |
| | $\mid$ CONCAT$(E_1, E_2)$ | |
| MiniStmt $S$ | $::=$ SELECT $E_1$ FROM $I$ WHERE $E_2$ | *Select query* |
| | $\mid$ CREATE TABLE $I$ ( $TE^+$ ) IN $S$ | *Create table* |
| MiniTerm $T$ | $::= E \mid S$ | |

Note the modification to create table. To avoid the need to reason about imperative state across statements, we modify create table to behave analogously to the `let ... in` expressions of functional languages. Next, we must define steps-to ($\Rightarrow$) semantics for this subset of the language .

$$\frac{}{b \text{ val}} \qquad \frac{}{i \text{ val}} \qquad \frac{}{s \text{ val}}$$

Val terms are fully-reduced values. Since SQL queries produce result sets, a Val represents a *column* of boolean, integer, or string values. Therefore, a Val conceptualizes a column containing a fixed set of elements. However, the exact set of elements is not necessarily specified since this subset of SQL lacks a means to insert values into a table and the exact result of a query depends on

the contents of the queried tables. This is reflected in the steps-to rules defined below. For example, a table column steps to a Val corresponding to the column's type, the < operator produces a column of boolean values from two columns of integer or string values, the + operator produces a column of integer values that is, in general, distinct from its two operands, and a where clause filters a result set into another result set.

$$\begin{aligned}
\text{CREATE TABLE } I \text{ ( } TE^+ \text{ ) IN } S &\Rightarrow S[(\ TE^+\ )/I] \\
I \text{ BOOL} &\Rightarrow I : b \\
I \text{ INT} &\Rightarrow I : i \\
I \text{ VARCHAR} &\Rightarrow I : s \\
\text{SELECT } E_1 \text{ FROM } (I_0 : v_0,\ I_1 : v_1,\ \ldots) \text{ WHERE } E_2 &\Rightarrow \\
\text{SELECT } E_1[v_0/I_0] \text{ FROM } (I_1 : v_1,\ \ldots) &\text{ WHERE } E_2[v_0/I_0] \\
\text{SELECT } E_1 \text{ FROM } (I : v) \text{ WHERE } E_2 &\Rightarrow E_1[v/I] \text{ WHERE } E_2[v/I] \\
b_1 \text{ WHERE } b_2 &\Rightarrow b_3 \\
i_1 \text{ WHERE } b &\Rightarrow i_2 \\
s_1 \text{ WHERE } b &\Rightarrow s_2 \\
b_1 \text{ AND } b_2 &\Rightarrow b_3 \\
b_1 = b_2 &\Rightarrow b_3 \\
i_1 = i_2 &\Rightarrow b \\
s_1 = s_2 &\Rightarrow b \\
i_1 < i_2 &\Rightarrow b \\
s_1 < s_2 &\Rightarrow b \\
i_1 + i_2 &\Rightarrow i_3 \\
\text{CONCAT}(s_1,\ s_2) &\Rightarrow s_3
\end{aligned}$$

There is also a contextual search rule. The syntax of the context $K$ is defined in Appendix A.1. It enforces left-to-right evaluation of operands within expressions and that the where clause evaluates before the selected expressions. This matches the evaluation order of SQL [1].

$$\frac{t_1 \Rightarrow t_2}{K[t_1] \Rightarrow K[t_2]}$$

The rules for substitutions $S[(\ TE^+\ )/I]$ and $E[v/I]$ are defined in Appendix A.2. We can now state the progress and preservation theorems for the simplified language.

Theorem 5.1 (Progress). *Suppose $T$ is a well-typed MiniTerm. Then either $T$ val, or $T \Rightarrow T'$ for some $T'$.*

Proof. Induction on the syntax of MiniTerm.

There are two base cases to consider. First, suppose $T$ = Val $v$. Then $T$ val by the syntax Val $v ::= b \mid i \mid s$. Second, suppose $T$ = Identifier $I$. $T$ being a lone identifier contradicts the hypothesis that $T$ is well-typed, so this case can be ignored.

For the inductive case, consider the remaining possible derivations of MiniTerm.

(1) Suppose $T = E_1$ AND $E_2$. By the hypothesis that $T$ is well-typed, ty-and gives that $E_1 : \sigma_1 \to$ BOOL, $E_2 : \sigma_2 \to$ BOOL. Then $T$ steps:
   - to $b_3$ if $E_1 =$ Bool $b_1$ and $E_2 =$ Bool $b_2$.
   - on the right, $E_2 \Rightarrow E_2'$, if $E_1 =$ Bool $b_1$. $E_2 \Rightarrow E_2'$ is possible by the inductive hypothesis.
   - on the left, $E_1 \Rightarrow E_1'$, otherwise. $E_1 \Rightarrow E_1'$ is possible by the inductive hypothesis.

(2) Suppose $T = E_1 = E_2$, $T = E_1 < E_2$, $T = E_1 + E_2$, or $T =$ CONCAT$(E_1, E_2)$. Apply reasoning analogous to (1).

(3) Suppose $T =$ SELECT $E_1$ FROM $I$ WHERE $E_2$. By the hypothesis that $T$ is well-typed, ty-select-where gives that $I : \{TN : \sigma\}$, $E_1 : \sigma \to \theta$, $E_2 : \sigma \to$ BOOL. Then $T$ steps to $T' =$ SELECT $E_1$ FROM $(I_0 : v_0, \ldots)$ WHERE $E_2$. In turn, $T'$ steps:
   - to $T'' = E_1[v_0/I_0]$ WHERE $E_2[v_0/I_0]$ if $(I_0 : v_0)$ is the only column selected from. In turn, $T''$ steps according to reasoning analogous to (1).
   - to SELECT $E_1[v_0/I_0]$ FROM $(I_1 : v_1, \ldots)$ WHERE $E_2[v_0/I_0]$ if there are multiple columns selected from.

(4) Suppose $T =$ CREATE TABLE $I$ ( $TE^+$ ) IN $S$. Then $T \Rightarrow S[(\ TE^+\ )/I]$.

□

Theorem 5.2 (Preservation). *Let $T$ be a MiniTerm with type $\tau$. Suppose $T : \tau \Rightarrow T' : \tau'$. Then $\tau' <: \tau$.*

Proof. Induction on MiniTerms considering each steps-to rule.

(1) Suppose $T =$ CREATE TABLE $I$ ( $TE^+$ ) IN $S$. Then $T \Rightarrow S[(\ TE^+\ )/I]$. Since ty-create-table does not match the modified create-table-in statement, we would need to appeal to a typing rule and substitution rule that would allow concluding $S[(\ TE^+\ )/I] : \tau$.

(2) Suppose $T = I$ BOOL, so $\tau =$ BOOL by ty-column. Then $T \Rightarrow I : b$. Apply ty-bool-expr.

(3) Suppose $T = I$ INT or $T = I$ VARCHAR. Apply reasoning analogous to (2).

(4) Suppose $T =$ SELECT $E_1$ FROM $(I_0 : v_0,\ I_1 : v_1,\ \ldots)$ WHERE $E_2$. By the hypothesis that $T$ is well-typed, ty-select-where gives that $\tau = \{TN : \langle E_1 : \theta \rangle\}$. $T$ steps to $T' =$ SELECT $E_1[v_0/I_0]$ FROM $(I_1 : v_1, \ldots)$ WHERE $E_2[v_0/I_0]$. Again appealing to a substitution rule, we can apply ty-select-where to obtain $\tau' = \{TN : \langle E_1 : \theta \rangle\}$ and therefore $\tau = \tau'$ (so $\tau' <: \tau$).

(5) Suppose $T =$ SELECT $E_1$ FROM $(I : v)$ WHERE $E_2$. Apply reasoning analogous to (4).

(6) Suppose $T = E_1$ WHERE $E_2$. Since this syntax is not accepted if written by the user, it is not exactly matched by a typing rule, so the subsequent argument is imprecise. However, assuming ty-select-where can be used to give $E_1 : \sigma \to \theta$, $E_2 : \sigma \to$ BOOL, argue that $\tau = \sigma \to \theta$. $T$ steps:
   - to $b_3$ if $E_1 =$ Bool $b_1$ and $E_2 =$ Bool $b_2$. Apply ty-bool-expr to obtain $\tau = \tau' = \langle\rangle \to$ BOOL.
   - to $i_2$ if $E_1 =$ Int $i_1$ and $E_2 =$ Bool $b$, or to $s_2$ if $E_1 =$ String $s_1$ and $E_2 =$ Bool $b$. Apply analogous reasoning to above.
   - on the left, $E_1 \Rightarrow E_1'$, if $E_2 =$ Bool $b$. $E_1 <: \sigma \to \theta$ by the inductive hypothesis, so $\tau' <: \tau$.
   - on the right, $E_2 \Rightarrow E_2'$, otherwise. Apply analogous reasoning to above.

(7) Suppose $T = E_1$ AND $E_2$. ty-and gives that $E_1 : \sigma_1 \to$ BOOL, $E_2 : \sigma_2 \to$ BOOL, and $\tau = \sigma_1 @ \sigma_2 \to$ BOOL. $T$ steps:
   - to $b_3$ if $E_1 =$ Bool $b_1$ and $E_2 =$ Bool $b_2$. Apply ty-bool-expr to obtain $\tau' = \langle\rangle \to$ BOOL. Therefore $\tau' <: \tau$ by contravariance.

- on the right, $E_2 \Rightarrow E_2'$, if $E_1 = \text{Bool } b_1$. $E_2' <: \sigma_2 \rightarrow \text{BOOL}$ by the inductive hypothesis. Apply ty-and to obtain $\tau' <: \sigma_1 @ \sigma_2 \rightarrow \text{BOOL}$.
- on the left, $E_1 \Rightarrow E_1'$, otherwise. Apply the same reasoning as above.

(8) Suppose $T = E_1 = E_2$, $T = E_1 < E_2$, $T = E_1 + E_2$, or $T = \text{CONCAT}(E_1,\ E_2)$. Apply reasoning analogous to (7).

□

## 6 FUTURE WORK

There are several ways this project could be extended. One interesting feature would be to deploy this project as a layer in front of a real DBMS to automatically intercept and type-check incoming queries. Rather than users having to specify all created tables up front, the project could query the database catalog to discover tables and fields. One could also expand the subset of SQL supported to support features like not fully qualifying column names, supports wildcards in `SELECT` clauses, anonymous tables, insert statements, and so on. Additional state could be tracked during the type-checking algorithm to support more verbose and helpful error messages. This would help users find how and where exactly their query failed to type-check. Finally, our progress and preservation theorem proofs are just a sketch of a subset of our subset of SQL. They are intentionally limited in rigor, particularly since the steps-to semantics introduce states that were not explicitly described by the abstract syntax or type checking rules. Without insert statements or imperative state to describe the contents of tables, the execution of a query must be described very coarsely. There is great potential in this area for more precise formulations of steps-to semantics for SQL and the proofs that would follow from it.

## 7 CONCLUSION

We presented an abstract syntax for a subset of the SQL language and defined typing rules for this syntax. We implemented a parser and type-checker for SQL programs in Python and discussed our experiences of the development. Finally, we sketched a proof of the progress and preservation theorems for a subset of our syntax. While aspects of our project are preliminary, there is significant potential for subsequent work.

## REFERENCES

[1] Microsoft 2021. *SELECT (Transact-SQL).* Microsoft. https://docs.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql?redirectedfrom=MSDN&view=sql-server-ver15#logical-processing-order-of-the-select-statement

[2] 2022. *Parsy.* Retrieved March 13, 2022 from https://github.com/python-parsy/parsy

[3] Microsoft 2022. *TypeScript: JavaScript With Syntax For Types.* Microsoft. https://www.typescriptlang.org/

[4] Python Software Foundation 2022. *typing – Support for type hints.* Python Software Foundation. https://docs.python.org/3/library/typing.html

[5] James Caldwell and Ryan Roan. 2012. Type Checking SQL for Secure Database Access. In *2012 Symposium on Trends in Functional Programming (TFP '12).* University of St Andrews, UK.

[6] Jonathan Leffler. 2017. *BNF Grammar for ISO/IEC 9075:1999 - Database Language SQL (SQL-99).* Retrieved March 21, 2022 from https://ronsavage.github.io/SQL/sql-99.bnf.html

# A ADDITIONAL DEFINITIONS FOR PROGRESS AND PRESERVATION

## A.1 Steps-To Context

$$
\begin{aligned}
\mathit{Context}\ K ::= {} & \cdot \\
& \mid K \text{ AND } E \mid v \text{ AND } K \\
& \mid K = E \mid v = K \\
& \mid K < E \mid v < K \\
& \mid K + E \mid v + K \\
& \mid \text{CONCAT}(K,\ E) \mid \text{CONCAT}(v,\ K) \\
& \mid E \text{ WHERE } K \mid K \text{ WHERE } v \\
& \mid \text{SELECT } E_1 \text{ FROM } K \text{ WHERE } E_2
\end{aligned}
$$

## A.2 Substitution Rules

The substitution $S[(\ TE^{+}\ )/I]$ is defined using the following rules:

$$\frac{I' \text{ distinct from } I}{(\text{CREATE TABLE } I'\ (\ TE^{+}\ )\ \text{IN } S)[(\ TE^{+}\ )/I] \Rightarrow_{sub} \text{CREATE TABLE } I'\ (\ TE^{+}\ )\ \text{IN } S[(\ TE^{+}\ )/I]}$$

$$\frac{I' \text{ distinct from } I}{(\text{SELECT } E_1 \text{ FROM } I' \text{ WHERE } E_2)[(\ TE^{+}\ )/I] \Rightarrow_{sub} \text{SELECT } E_1 \text{ FROM } I' \text{ WHERE } E_2}$$

$$(\text{SELECT } E_1 \text{ FROM } I \text{ WHERE } E_2)[(\ TE^{+}\ )/I] \Rightarrow_{sub} \text{SELECT } E_1 \text{ FROM } (\ TE^{+}\ ) \text{ WHERE } E_2$$

The substitution $E[v/I]$ is defined using the following rules:

$$\frac{I' \text{ distinct from } I}{I'[v/I] \Rightarrow_{sub} I'} \qquad I[v/I] \Rightarrow_{sub} v \qquad v'[v/I] \Rightarrow_{sub} v' \qquad \frac{E_1 \Rightarrow_{sub} E_2}{K[E_1] \Rightarrow_{sub} K[E_2]}$$

The contextual search rule has the syntax below. No order of substitution is enforced.

$$
\begin{aligned}
\mathit{SubContext}\ K ::= {} & \cdot \\
& \mid K \text{ AND } E \mid E \text{ AND } K \\
& \mid K = E \mid E = K \\
& \mid K < E \mid E < K \\
& \mid K + E \mid E + K \\
& \mid \text{CONCAT}(K,\ E) \mid \text{CONCAT}(E,\ K)
\end{aligned}
$$