Type Checking SQL for Secure Database Access

James Caldwell[1] and Ryan Roan[2]

[1] Department of Computer Science, University of Wyoming, Laramie, WY 82071
[2] Handel Information Technologies, 200 South 3rd Street Laramie, WY 82070

Abstract. In client/server software systems clients often interface with database servers by dynamically generating strings which are to be interpreted as an SQL query. The practice of passing raw string data that may or may not be syntactically correct or well-typed admits the possibility of run-time errors and compromises the security of the database by opening them to injection attacks. It is possible to avoid a large class of these attacks by comparing the syntactic structure of a generated query to the structure of the query the programmer intended. In this paper we present a type system for a subset of SQL that provides stronger guarantees than syntax based approaches. A SQL query is well-typed with respect to a database instance if it only refers to table names and field names that exist in the target database and if those references are all well typed. Error messages from queries that are syntactically well-formed but which refer to nonexistent tables or field names or are otherwise intentionally non well-typed can be used to discoverer the structure of the database providing significant aid to intruders. The type system described here has been implemented using F# interfacing to a MySQL backend. This paper briefly introduces the ideas behind SQL injection attacks, presents both an abstract syntax for SQL and the typing rules, and provides some details of the implementation.

1 Introduction

In this paper we describe a type system for a subset of the SQL query language. A type checker based on this system has been implemented in F# and it serves as a firewall to an SQL database server. Dynamically generated queries are parsed and statically type checked against the state of the data-base server. Thus, queries accessing tables and columns in those tables are well-typed only if the names match valid tables and column names in the database and if the types of the data stored there are consistent with the usage of those fields in the query. The firewall rejects syntactically ill-formed queries, those that are not well-typed and can be used to detect SQL injection attacks.

Web applications are often written in some combination of Java, JavaScript, Perl, Python and PHP and they typically pass on SQL queries to the server as strings. Passing untyped strings leads to the possibility

of run-time errors [9] and compromises the security of the database by opening them to injection attacks [3]. An approach based on parsing dynamically generated SQL queries and comparing resulting shape of the parse tree to the expected shape protects against a large class of injection attacks [1]. By type-checking we provide even stronger guarantees.

We should mention that a number of efforts are underway to integrate syntax and type-checking into scripting languages and libraries allowing access to databases [2, 8, 4, 10, 5]. Our approach is different in that it provides type-checking support as a trusted layer front of the database backend. Syntactically ill-formed or non well-typed queries are rejected and never reach the server. The inputs to the type checker are query strings generated by an existing web application. The input is parsed and then type-checked before being passed to the backend database for processing.

In the rest of the paper we introduce the basics of SQL injection attacks, we present the type system and describe the implementation in F# which connects to the MySQL backend. The type system presented here is built using techniques well-known in the the programming languages community and are described in a number of text-books [7, 6].

2 SQL Injection Attacks

SQL injection attacks allow malicious users to make unauthorized access to a database. Injection attacks are mounted by users who enter text which, used to dynamically generate a query, is interpreted as a valid SQL but whose resulting effect is different from the one intended by the programmer. By altering the structure of the query, the user is able to force their input into being treated as code. With partial control of the code, a user is then able to perform tasks of their choosing.

A significant risk related to SQL-Injection is exposing the schemas in use on the target database. An attacker with knowledge about the database structure is far more likely to succeed in unauthorized access than an attacker without such information. The type checker only allows well-typed queries to be run against the database - this allows an application to control the visibility of errors that would generally be returned by the SQL engine. In an unprotected server environment, when a statement is run requesting information from a table that does not exist, an error will be generated by the SQL engine specifying the table does not exist. The error provides information to the attacker that a table does not exist in the database. This information would be valuable on its own, but additional information in the error tends to include the SQL database

runtime information. This information identifies what type of database is being used, and since each SQL engine provides built in functionality, an attacker would now know exactly what built in schemes may exist. The attacker will typically try other options until a table name is discovered. A query referencing a non-existent table is not well-typed and non-well-typed results in a null answer. A null answer provides some information to the attacker, but not as much as an error message which explicitly identifies a table by name.

Once a table has been identified, an injection can be created by determining the column names in a similar guess and check fashion. Similarly, a query containing references to non-existent columns will not be well typed, by checking if the statement is well-typed we control the information returned from a query. This limits the information an attacker can gain from error messages. With no information to go on, it is significantly more difficult to formulate a successful attack. Note that in a parser based approach, the shape of a dynamically generated query may match the expected shape and yet still refer to nonexistent table names and thereby inadvertently exposing information to an attacker via error messages.

Another type of injection attack is based on purposely mismatching column types in the hopes that a SQL error will yield schema information. When a user is asked to enter an integer, a user might purposely input a string value. This can cause the database engine to throw an error that not only informs the attacker of the table being selected from, but also the column name that had the incorrect value in it. By type checking all statements before execution, we are able to identify that a value provided is of the wrong type and can prevent these errors from being displayed as well.

Most applications taking information from the user and passing it through a business layer to perform a database query are susceptible to SQL-Injection. The applications with the highest risk though, are web-based applications and sites that are Internet facing. The nature of the application requires it to be available to both the target users, as well as potentially unintended users. As a visual way to describe various attacks, we present a simple program that could commonly be used as an authentication page for a website.

3 The SQL Type System

In this section we describe a static type system for dynamically generated SQL queries. The table schemas themselves serves as a basis for the specification of the type of the database. The fact that the type checker requires access to the table schemas means it is implemented as a trusted servers IDE application.

3.1 Abstract Syntax for SQL

The abstract syntax is for a core subset of the SQL query language. SQL is known for its Baroque syntax and we have only target a subset with the abstract syntax. The subset of the language presented here includes the constructs required to specify typed relational schemas, to populate them with data, and to query them.
The syntactic classes are given as follows.

$TName$: *table names*	$FName$: *field names*
E : *expressions*	N : *numerals*
S : *schemas*	T : *tables*
V : *tuples of values*	$Type$: *types*

The abstract syntax of the language is presented in Figure 1. If A is a syntax class, we use the notation $[A]^*$ to denote a list of elements of class A and $[A]^+$ to denote a non-empty list of elements of class A.

We assume the values of the primitive types (`BOOL`, `INT` , `STRING`, and `CHAR`$[k]$) are understood. We use the notation $\langle\rangle$ to denote the empty tuple and abuse notation by using the same constant to denote the single element of the type `UNIT`. The type constructor `NULL` is intended to indicate that that type is nullable. If θ is a type `NULL` $\theta = \{\langle\rangle\} \cup \theta$. Note that representations of null vary in different SQL implementations.

If T is a type the values in `NULL` T are $\{\langle\rangle\} \cup T$.

The syntactic class I is a collection of simple identifiers. The class $TName$ includes simple identifiers as well as the special table name $\star_{[FName]^*}$. This name is used as a placeholder for anonymous tables that may arise from a query. The parameter $[FName]^*$ indicates the field names included in the schema for $\star$. We use meta-variables $\{I\star, I'\star, I_1\star, \cdots\}$ to denote table names which might be simple identifiers or anonymous names of the form $\star_{names}$. Field names are captured by the class $FName$ and includes simple identifiers as well as indexed field names of the form $I_1\star.I_2$ where $I_1\star$ is a table name and I_2 is a simple identifier naming the field.

$$
\begin{array}{lll}
I & ::= & \mathit{identifier} \\
\mathit{TName} & ::= & \star_{[\mathit{FName}]^*} \mid I \\
\mathit{FName} & ::= & I \mid \mathit{TName}\,.\,I \\
\mathit{Type} & ::= & \texttt{UNIT} \mid \texttt{BOOL} \mid \texttt{INT} \mid \texttt{STRING} \mid \texttt{CHAR}[k] \mid \texttt{NULL}\,\mathit{Type} \\
S & ::= & \langle I_1 : \theta_1, \cdots, I_k : \theta_k \rangle \text{ where } k \geq 0 \quad \text{and } \theta_i \in \mathit{Type} \\
V & ::= & \langle v_1, \cdots, v_k \rangle \text{ where } k \geq 0 \quad \text{and } v_i \text{ are values.} \\
E & ::= & N \mid \mathit{FName} \mid E_1 \texttt{ OR } E_2 \mid E_1 \texttt{ AND } E_2 \mid \texttt{NOT } E \\
 & & \mid E_1 = E_2 \mid E_1 < E_2 \mid E_1 + E_2 \mid E_1 * E_2 \\
T & ::= & \mathit{TName} \\
 & & \mid \texttt{CREATE TABLE } \mathit{TName}\ S \\
 & & \mid \texttt{INSERT INTO } T\ V \\
 & & \mid \texttt{JOIN } T_1\ T_2\ \texttt{ ON }\ E \\
 & & \mid \texttt{SELECT } [\mathit{FName}]^+ \texttt{ FROM }\ T \\
 & & \mid \texttt{SELECT } [\mathit{FName}]^+ \texttt{ FROM }\ T \texttt{ WHERE }\ E \\
 & & \mid T_1 ; T_2 \\
 & & \mid T_1 \texttt{ GO } T_2
\end{array}
$$

Fig. 1. An Abstract Syntax for SQL

3.2 Supporting operations

In this section we define schemas and type assignments and the operations on them. The syntactic class S of schemas plays a fundamental role in type checking.

Definition 1 (Schema). *A* schema *is a binding of names to types; they are of the form* $\langle I_1 : \theta_1, \cdots, I_k : \theta_k \rangle$*. We will use meta-variables* $\{\sigma, \sigma', \sigma_1, \cdots\}$ *to denote schemes. We insist that the binding be functional, if* $I_i = I_j$ *and* $I_i : \theta_i$ *and* $I_j : \theta_j$ *are field name-type pairs in* σ *then* $\theta_i = \theta_j$*.*

We will write $|\sigma|$ to denote the *length* of the tuple σ. We write $dom(\sigma)$ to be the *domain* of σ – the tuple of field names in the order in which they appear in σ. We write $Ty(\sigma)$ to be the type of tuples inhabiting a table of type σ *i.e.* $Ty\,\langle I_1 : \theta_1, \cdots, I_k : \theta_k \rangle = \theta_1 \times \cdots \times \theta_k$.

For $i \in \{1..|\sigma|\}$ we write $\sigma[i]$ to denote the i^{th} pair in the tuple σ. We write *fst* and *snd* to denote the projection functions on pairs; thus $fst(\sigma[i])$ is the field name of the i^{th} pair in σ and $snd(\sigma[i])$ is the type of the i^{th} pair in σ. We write $\sigma @ \sigma'$ to denote the concatenation of the schemas σ and σ'. Not that this operation is not defined if the resulting tuple does not satisfy the functionality condition on schemas. We write $\sigma' \subseteq \sigma$ ext ρ to mean that that there is an injection $\rho : \{1..|\sigma'|\} \rightarrow \{1..|\sigma|\}$ which which preserves field names and types. This function is evidence for $\sigma' \subseteq \sigma$. The condition on ρ that guarantees the injection is name and

type preserving is given as:

$$\forall i : \{1..|\sigma'|\}.\ (\sigma'[i]) = (\sigma[\rho(i)])$$

Type assignments do not appear in the object language given by the abstract syntax but are an important structure used in to specify the typing rules.

Definition 2 (Type Assignment). *A* type assignment *is a binding of a table names to their schemes and is of the form* $\langle I_1\star : \sigma_1, \cdots, I_k\star : \sigma_k\rangle$. *We use meta-variables* $\{\pi, \pi', \pi_1, \cdots\}$ *to denote type assignments. Type assignments will be used to keep track of the tables and their schemes in the database. We assume the binding is functional, thus if* $I_i\star$ *and* $I_j\star$ *are equal table names in* π *and* $\langle I_i\star : \sigma_i\rangle \in \pi$ *and* $\langle I_j\star : \sigma_j\rangle \in \pi$, *then* $\sigma_i = \sigma_j$.

We will write $|\pi|$ to denote the *length* of the type assignment π. We write $dom(\pi)$ to be the *domain* of π – the collection of table names in π. For type assignments π and π' we write $\pi \uplus \pi'$ to denote the type assignment in which the bindings in π' take precedence over those in π.

Definition 3 (Unique fieldname membership $I \in_1 \pi$). *We introduce the predicate* $I \in_1 \pi$ *to mean that the unqualified field name* I *occurs exactly once in the type assignment* π. *That means it occurs in exactly one schema in* π. *Here is a formal definition:*

$$I \in_1 \pi \stackrel{\text{def}}{=} \exists! I_1.\exists \sigma.\langle I_1, \sigma\rangle \in \pi \wedge I \in dom(\sigma)$$

The use of unique existence[1] *(*$\exists!$*) in the definition of* $\in_1$ *means that unqualified field names can only occur in exactly one schema in* π.

Definition 4 (Type Assignment Lookup). *For an identifier* I *and a type assignment* π, *if* $I \in_1 \pi$ *then we write* $\pi(I)$ *to denote the unique* θ *such that the following condition holds:*

$$\pi(I) = \theta \stackrel{\text{def}}{=} \exists! I_1.\exists \sigma.\langle I_1, \sigma\rangle \in \pi \wedge \langle I, \theta\rangle \in \sigma$$

If $I \notin_1 \pi$ *then* $\pi(I)$ *is undefined.*

[1] Note that $\exists!$ is notation indicating unique existence. It can be defined in terms of ordinary existence as follows: $\exists! x.\phi[x] \stackrel{\text{def}}{=} \exists x.\phi[x] \wedge \forall y.\ \phi[y] \Rightarrow y = x$

Definition 5 (Schema arising from a restriction). *We introduce an operation for constructing a schema σ from a type assignment π and a list of field names names. The idea is for each field name $n \in names$ to be paired with its type in π to create a new schema. Note that field names can be simple identifiers or they might be qualified names of the form $(I_1\star.I_2)$ where $I_1\star$ is the table name and I_2 is the field name in that table. If the field name is simple, we insist that there is only one table in the type assignment π with that field name.*

The restriction $(\pi \downarrow names)$ is defined by recursion on the structure of the list names [2].

$$\begin{array}{ll}
\pi \downarrow [\,] = \langle\rangle & \\
\pi \downarrow ((I_1.I_2) :: ns) = \langle I_2 : \theta\rangle :: (\pi \downarrow ns) & \text{where } \langle I_1 : \sigma\rangle \in \pi \wedge \langle I_2 : \theta\rangle \in \sigma \\
\pi \downarrow (I :: ns) = \langle n : \theta\rangle :: (\pi \downarrow ns) & \text{where } \exists! I1.\langle I1 : \sigma\rangle \in \pi \wedge \langle I : \theta\rangle \in \sigma
\end{array}$$

Note that if there is a field name in *names* that does not occur in the type assignment π then $\pi \downarrow names$ is not defined.

3.3 The Typing Rules

Typing Table names The type of a table name (elements of the syntactic class *TName*) is the schema for the table bound to that name in the type assignment π.

For table names that are simple identifiers, the name must appear in the type assignment π bound to the schema σ.

$$\frac{}{\pi \vdash I : \sigma} \text{ if } \langle I : \sigma\rangle \in \pi$$

There is a special rule for the name $\star_{names}$ which arises from a query. The table name $\star_{names}$ is used for tables that are constructed from queries projecting out the fields listed in *names* and so, their field names and associated types must already exist in some schema in the type assignment π. The following is the typing rule for the table name $\star_{names}$:

$$\frac{}{\pi \vdash \star_{names} : \sigma} \text{ if } \sigma = (\pi \downarrow names)$$

The rule says that the field names *names* must appear in the domain of σ in the same order and the restriction of π to the names in the schema σ must be σ itself. This condition is only satisfied if the names in σ are already names in π and the types associated with those names match the types of those fields in π.

[2] We use F#'s double colon notation for cons (::) and to build tuples.

Typing field names The types of field names are gathered from their schemas in a type assignments π. If the name is qualified by the table name, the typing is easy.

$$\frac{}{\pi \vdash I_1.I_2 : \theta} \text{ if } \langle I : \sigma \rangle \in \pi \wedge \langle I_2 : \theta \rangle \in \sigma$$

To avoid ambiguity, unqualified names must occur uniquely as field names in π.

$$\frac{}{\pi \vdash I : \theta} \text{ if } I \in_1 \pi \wedge \pi(I) = \theta$$

Note that if either $I \in_1 \pi$ or $\pi(I) = \theta$ are undefined, the rule does not hold.

Typing Types Syntactically well-formed elements of the class *Type* are well-typed – there is nothing to check.This gives the following four rules.

$$\frac{}{\texttt{UNIT} : \mathit{Type}}$$

$$\frac{}{\texttt{BOOL} : \mathit{Type}} \qquad \frac{}{\texttt{INT} : \mathit{Type}}$$

$$\frac{\theta : \mathit{Type}}{\texttt{NULL}\ \theta : \mathit{Type}} \qquad \frac{}{\texttt{CHAR}[k] : \mathit{Type}} \text{ if } k \geq 0$$

Obviously, if we extended the language of *Type* to include user defined types, checking them may be more complex.

Typing Schemas A schema $\sigma = \langle I_1 : \theta_1, \cdots, I_k : \theta_k \rangle$ is well typed if it is functional in its names and all the types θ_i are well formed.

$$\frac{\vdash \theta_i : \mathit{Type}}{\vdash \sigma : \mathit{Ty}\,\sigma} \text{ if } \begin{array}{l} \sigma = \langle I_1 : \theta_1, \cdots, I_m : \theta_m \rangle \\ \forall j, k : \{1..m\}.\ (i \neq j \wedge I_i = I_j) \Rightarrow \theta_i = \theta_j \end{array}$$

Typing Tuples The syntax class V of tuples of values is well-typed with respect to a schema. Given a tuple $V = \langle v_1, \cdots, v_m \rangle$ and a schema $\sigma = \langle I_1 : \theta_1, \cdots, I_m : \theta_m \rangle$, then the rule appears as follows:

$$\frac{\vdash v_i : \theta_i}{\sigma \vdash V} \text{ if } |V| = |\sigma| \text{ and } i \in \{1..|V|\}$$

We assume that there are type rules for each primitive type, thus

$$\vdash v_i : \theta_i$$

is understood. For example, determining whether a number is an integer is understood. For the nullable types $\texttt{NULL}\ \theta$ the value must be the null value $\langle\rangle$ or of type θ; thus, $\sigma \vdash v_i : \ \texttt{NULL}\ \theta$ is well-typed if $v_i = \langle\rangle$ or $v_i : \theta$.

Note that if the schema and the tuple have different lengths, the rule does not apply.

Typing Expressions The meaning of expressions will (almost always) turn out to depend on the values in the positions of tuples in a particular schema. The semantics determines the typing rule. This means the type of the syntax class E of expressions is naturally modeled as a functions from schemas to the result type of the expression. In the end, the expression E of type $\sigma \to \theta$ can be applied to any tuple of type σ' where $\sigma \subseteq \sigma'$.

The domain schemas get built in the rule for typing a field name as an expression. In that case, the type of field name is the expression is determined by the type of that field in π. Thus if $F \in \mathit{FName}$ (F could be a simple identifier or a qualified name), we have the following typing rule. This is the only rule that actually uses the type assignment π and builds the domain for the expression.

$$\frac{\pi \vdash F : \theta}{\pi \vdash F : \langle F : \theta \rangle \to \theta}$$

Recalling that $(\sigma @ \sigma')$ denotes the functional concatenation of schemas, the other cases are given as follows.

$$\frac{}{\pi \vdash N : \langle\rangle \rightarrow \texttt{INT}}$$

$$\frac{\pi \vdash E_1 : \sigma_1 \rightarrow \texttt{BOOL} \quad \pi \vdash E_2 : \sigma_2 \rightarrow \texttt{BOOL}}{\pi \vdash E_1 \texttt{ OR } E_2 : (\sigma_1 @ \sigma_2) \rightarrow \texttt{BOOL}}$$

$$\frac{\pi \vdash E_1 : \sigma_1 \rightarrow \texttt{BOOL} \quad \pi \vdash E_2 : \sigma_2 \rightarrow \texttt{BOOL}}{\pi \vdash E_1 \texttt{ AND } E_2 : (\sigma_1 @ \sigma_2) \rightarrow \texttt{BOOL}}$$

$$\frac{\pi \vdash E : \sigma \rightarrow \texttt{BOOL}}{\pi \vdash \texttt{ NOT } E : \sigma \rightarrow \texttt{BOOL}}$$

$$\frac{\pi \vdash E_1 : \sigma_1 \rightarrow \theta \quad \pi \vdash E_2 : \sigma_2 \rightarrow \theta}{\pi \vdash E_1 = E_2 : (\sigma_1 @ \sigma_2) \rightarrow \texttt{BOOL}}$$

$$\frac{\pi \vdash E_1 : \sigma_1 \rightarrow \theta \quad \pi \vdash E_2 : \sigma_2 \rightarrow \theta}{\pi \vdash E_1 < E_2 : (\sigma_1 @ \sigma_2) \rightarrow \texttt{BOOL}}$$

$$\frac{\pi \vdash E_1 : \sigma_1 \rightarrow \texttt{INT} \quad \pi \vdash E_2 : \sigma_2 \rightarrow \texttt{INT}}{\pi \vdash E_1 + E_2 : (\sigma_1 @ \sigma_2) \rightarrow \texttt{INT}}$$

$$\frac{\pi \vdash E_1 : \sigma_1 \rightarrow \texttt{INT} \quad \pi \vdash E_2 : \sigma_2 \rightarrow \texttt{INT}}{\pi \vdash E_1 * E_2 : (\sigma_1 @ \sigma_2) \rightarrow \texttt{INT}}$$

Note that we assume the order operator $<$ is defined for all types $\theta \in \mathit{Type}$ but that addition and multiplication are only defined on type INT .

Typing Tables The syntactic class T includes phrases that denote tables. A single table phrase is typed by a name paired with its schema. The simplest case is when a table name is mentioned.

Typing a *TName* as a Table: The mention of a table name (say $I\star$) is well typed under type assignment π if $I\star$ names some well typed schema in π.

$$\frac{\pi \vdash I\star : \sigma}{\pi \vdash I\star : \{I\star : \sigma\}}$$

`CREATE TABLE` *:* The rule for creating a new table just says that the schema must be well-typed. Note that the typing rule for `CREATE` forces the name of a created table to be an identifier (and not $\star$).

$$\frac{\vdash \sigma : Ty\ \sigma}{\pi \vdash \texttt{CREATE TABLE}\ I\ \sigma : \{I : \sigma\}}$$

`INSERT INTO` *:* The rule for inserting a tuple of values into a table T checks that the type of the tuple matches the schema for the table being inserted into.

$$\frac{\vdash T : \{I\star : \sigma\} \qquad \sigma \vdash V}{\pi \vdash \texttt{INSERT INTO}\ T\ V : \{I\star : \sigma\}}$$

`JOIN ON` *:* Joins are used to consistently combine tables that possibly share common field names. The `ON` condition is an expression that can be used to specify how fields should be related.

$$\frac{\pi \vdash T_1 : \{I_1\star : \sigma_1\} \quad \pi \vdash T_2 : \{I_2\star : \sigma_2\} \quad \pi \vdash E : (\sigma_1 @ \sigma_2) \to \texttt{BOOL}}{\pi \vdash \texttt{JOIN}\ T_1\ T_2\ \texttt{ON}\ E : \{\star_{names} : (\sigma_1 @ \sigma_2)\}}$$

where $names = dom(\sigma_1) @ dom(\sigma_2)$.

`SELECT` *:* The `SELECT` statement is the core of the query language.

$$\frac{\pi \vdash T : \{I\star : \sigma\} \qquad \pi \vdash E : \sigma' \to \texttt{BOOL}}{\pi \vdash \texttt{SELECT}\ names\ \texttt{FROM}\ T\ \texttt{WHERE}\ E : \{\star_{names} : \pi \downarrow names\}} \begin{array}{l} \sigma' \subseteq \sigma\ \mathsf{ext}\ \rho' \\ names \subseteq dom(\sigma) \end{array}$$

`SEQUENCING` The sequencing operator is denoted by a semi-colon and allows a sequence of table operations where the latter operation has access to the database resulting from the previous operation.

$$\frac{\pi \vdash T_1 : \{I_1\star : \sigma_1\} \quad \pi \uplus \{I_1\star : \sigma_1\} \vdash T_2 : \{I_2\star : \sigma_2\}}{\pi \vdash T_1; T_2 : \{I_2\star : \sigma_2\}}$$

`GO` Parallel composition of table queries is typed as follows:

$$\frac{\pi \vdash T_1 : \{I_1\star : \sigma_1\} \quad \pi \vdash T_2 : \{I_2\star : \sigma_2\}}{\pi \vdash T_1; T_2 : \{I_2\star : \sigma_2\}}$$

4 Implementation

Our implementation was designed to be used as a tool for enterprise level software development. It is implemented in F# and utilizes Microsoft based products, though the implementation can easily be used by anyone interested in protecting their data from SQL Injection. The implementation sits as an extension of the server-side code in the application hierarchy. Due to the nature of the target market, the code was written in F#, a Microsoft Common Language Runtime (CLR) compatible object oriented functional language. By using a CLR compatible language, we are able to easily integrate with other CLR compatible languages and utilize their functionality and libraries as well. This strategy allowed us to use supported database communications within our implementation, through the MySQL C# database interaction dynamic link libraries.

4.1 Schema Retrieval

At the start of the application, an initial call is made to build the type assignment. Through the utilization of the SQL information tables, in the case of MySQL the `Information_Schema` tables, the type checker queries the shape of the SQL Instance and recovers the necessary information needed to perform type and syntax checking. Since the type system checks against an existing database, is imperative to know the table names, field names, and types before a request for information can be made. It is important to note, that since the system caches type assignment information, if the table structures in the database were to be altered the application would need to be reset as well. For the web application software we intend to target, this is not an unreasonable constraint, since changes in the underlying data base structure often require additional code changes and a restart of the application anyhow. For more dynamic applications,an implementation that does not cache this information could be built as well. The trade-off for this flexibility is in the overhead of repeatedly retrieving the SQL information tables before each SQL query is processed.

4.2 Parsing and Type Checking

Evaluating a dynamically generated query on the SQL server is a multi-stage process.

Once a query string has been passed to the server side, the first step in the process is to parse the input. The parser has been implemented

using FsLex and FsYacc. The concrete syntax is a sub-set of the full SQL grammar and the parser maps strings into recursive types representing the abstract syntax presented above. For example, the abstract syntax of phrases in class of table statements are represented by the following recursive type.

```
type TStatement =
| TName of TName
| Create of TName * Schema
| Insert of TName * V
| Join of TName * TName * Expression
| Select of FName list * TName
| SelectWhere of FName list * TName * Expression
| Semi of TStatement * TStatement
| Go of TStatement * TStatement
```

The parser maps strings into the corresponding abstract syntax representations which serve as the basis for the type checking algorithm. The typing rules are naturally implemented by recursion on the structure of the abstract syntax. As an example consider the first few cases of the function checking the type of table statements.

```
let rec WellTypedTStatement (TA ta) statement =
    match statement with
    |TName (tn) -> WellTypedTName (TA ta) tn
    |Create(tn, schema) -> WellTypedSchema schema
    |Insert(tn, values) ->
        WellTypedTName (TA ta) tn
         && (let schema = SchemaOfTable (TA ta) tn in
                WellTypedTuple schema values)
    |Join(tn1,tn2,e)->
        WellTypedTName (TA ta) tn1
         && WellTypedTName (TA ta) tn2
         && WellTypedExpression (TA ta) e
     ...
```

5 Conclusions

The typing rules above have served as a specification for an SQL type checker implemented in F#. The database system being used is MySQL and although only a subset of SQL is encoded by the abstract syntax it

is rich enough for us to examine simple web applications. We have also used these typing rules to define a denotational semantics for this subset of SQL. We have implemented a parser for taking string based queries from a client side into an accepted SQL query. We have used FsLex and FsYacc to implement the parser.

Since the typing rules depend on the database schema, the application must run as trusted code. It queries the database for its own schema by requesting the `information_schema` tables. The type checker reconstructs the schemas of the various tables in the database from these schemas.

Also note that the abstract syntax does not tightly bind us to any particular SQL implementation although the parser does.

References

1. Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th international workshop on Software Engineering and Middleware*, SEM '05, pages 106–113, New York, NY, USA, 2005. ACM.
2. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer Berlin / Heidelberg, 2007.
3. William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL-Injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.
4. Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.
5. Atsushi Ohori and Katsuhiro Ueno. Making standard ML a practical database programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 307–319, New York, NY, USA, 2011. ACM.
6. Benjamin C Pierce. *Types and Programming Languages*. MIT, 2002.
7. David A. Schmidt. *The Structure of Typed Programming Languages*. MIT, 1994.
8. N. Swamy, B.J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 369 –383, may 2008.
9. Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), September 2007.
10. Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, January 2000.