Static Checking of Dynamically Generated Queries in Database Applications *

Carl Gould, Zhendong Su, and Premkumar Devanbu
Department of Computer Science
University of California, Davis
{gould,su,devanbu}@cs.ucdavis.edu

Abstract

Many data-intensive applications dynamically construct queries in response to client requests and execute them. Java servlets, e.g.*, can create string representations of SQL queries and then send the queries, using JDBC, to a database server for execution. The servlet programmer enjoys static checking via Java's strong type system. However, the Java type system does little to check for possible errors in the dynamically generated SQL query strings. Thus, a type error in a generated selection query (*e.g.*, comparing a string attribute with an integer) can result in an SQL runtime exception. Currently, such defects must be rooted out through careful testing, or (worse) might be found by customers at runtime. In this paper, we present a* sound, static, *program analysis technique to verify the correctness of dynamically generated query strings. We describe our analysis technique and provide soundness results for our static analysis algorithm. We also describe the details of a prototype tool based on the algorithm and present several illustrative defects found in senior software-engineering student-team projects, online tutorial examples, and a real-world purchase order system written by one of the authors.*

1. Introduction

Data-intensive applications often dynamically construct database query strings and execute them. For example, a typical Java servlet web service constructs SQL query strings and dispatches them over a JDBC connector to an SQL-compliant database. In this example scenario, the Java servlet program generates and manipulates SQL queries as string data. Here, we refer to Java as the *meta-language* used to manipulate *object-language* programs in SQL.

We use a concrete example (see below) throughout the paper to explain our analysis technique. Consider a front-end Java servlet for a grocery store, with an SQL-driven database back-end. The database has a table `INVENTORY`, containing a list of all items in the store. This table has three columns: `RETAIL`, `WHOLESALE`, and `TYPE`, among others. The `RETAIL` and `WHOLESALE` columns are both of type integer, indicating their respective costs in cents. The `TYPE` column is an integer, representing the product type-codes of the items in the table. In the grocery store database, there is another table `TYPES` used to look up type-codes. This table contains the columns `TYPECODE`, `TYPEDESC`, and `NAME`, of the types integer, varchar (a string), and varchar, respectively.

The following example code fragment illustrates some common errors that programmers might make when programming Java servlet applications:

```
ResultSet getPerishablePrices(String lowerBound) {
  String query =   "SELECT '$' || "
                 + "(RETAIL/100) FROM INVENTORY "
                 + "WHERE ";
  if (lowerBound != null) {
    query += "WHOLESALE > " + lowerBound + " AND ";
  }
  query +=  "TYPE IN (" + getPerishableTypeCode()
                        + ");";
  return statement.executeQuery(query);
}

String getPerishableTypeCode() {
  return   "SELECT TYPECODE, TYPEDESC FROM TYPES "
         + "WHERE NAME = 'fish' OR NAME = 'meat'";
}
```

The method `getPerishablePrices` constructs the string `query` to hold an SQL `SELECT` statement to return the prices of all the perishable items, and executes the query. It uses the string returned by the method `getPerishableTypeCode` as a sub-query. In the code, `||` is the concatenation operator, and the clause `TYPE IN (...)` checks whether the type-code `TYPE` matches any of the type-codes of the perishable items. If `lowerBound` is "595", then the query to be executed is:

```
SELECT '$' || (RETAIL/100) FROM INVENTORY
   WHERE WHOLESALE > 595 AND TYPE IN
     (SELECT TYPECODE, TYPEDESC FROM TYPES
        WHERE NAME = 'fish' OR NAME = 'meat');
```

* The first and second authors were supported by a Startup Fund from the University of California, Davis to the second author. The last author was supported by NSF (both CISE & ITR programs).

Several different runtime errors can arise with this example. We list them below; note that *none of these* would be caught by Java's type system:

Error (1). The expression `'$' || (RETAIL/100)` concatenates the *character* `'$'` with the result of the *numeric* expression `RETAIL/100`. While some database systems will implicitly type-cast the numeric result to a string, many do not, and will issue a runtime error.

Error (2). Consider the expression `WHOLESALE > lowerBound`. The variable `lowerBound` is declared as a string, and the `WHOLESALE` column is of type integer. As long as `lowerBound` is indeed a string representing a number, there are no type errors. However, this is risky: nothing (certainly not the Java type system itself) keeps the string variable `lowerBound` from containing non-numeric characters.

Error (3). The string returned by the method `getPerishableTypeCode()` constitutes a sub-query that selects two columns from the table `TYPES`. Because the `IN` clause of SQL supports only sub-queries returning a single column (in this context), a runtime error would arise. This can happen if the method `getPerishableTypeCode()` did return a single column before, but was inadvertently changed to return two columns.

This specific combination of Java as the meta-language and SQL as the object-language is widely used today. The databases receiving these constructed SQL queries certainly perform syntax and semantic checking of the queries. But because these queries are dynamically generated, errors are only discovered at runtime. It would be desirable to catch these errors statically in the source code.

In this paper, we present a static analysis to flag potential errors or guarantee their absence in dynamically generated SQL queries. Our approach is based on a combination of automata-theoretic techniques [9], and a variant of the context-free language (CFL) reachability problem [14, 15]. As a first step, our analysis builds upon a static string analysis to build a *conservative* representation of the generated query strings as a finite-state automaton. Then, we statically check the finite- state automaton with a modified version of the context-free language reachability algorithm. Our analysis is sound in the sense that if it does not find any errors, then such errors do not occur at runtime. We have implemented the analysis and tested our tool on realistic programs using JDBC, including senior software-engineering student-team projects, online tutorial examples, and a real-world purchase order system written by one of the authors. Our tool is able to detect some known and unknown errors in these programs. Although it has not been tuned for performance, the analysis finishes in a few minutes on all test programs. Furthermore, our analysis empirically appears quite precise, with a low false-positive error rate.

The rest of the paper is structured as follows. We begin with background on the string analysis and context-free language reachability and a brief overview of our analysis (Section 2). Then we present our analysis in more detail (Section 3) and discuss our experimental setup and results (Section 4). Finally, we survey related work (Section 5) and conclude with a discussion of possible future work (Section 6).

2. Background and Analysis Overview

We now describe the technical context of our work.

2.1. Static String Analysis of Java Programs

As mentioned earlier, our analysis makes use of the string analysis of Java programs reported in [6]. Essentially, it is an interprocedural data-flow analysis [10, 12] to approximate the semantics of string manipulation expressions of a program. The analysis is similar to a pointer analysis [3] for imperative languages or a control-flow analysis(0-CFA) [17] for functional languages. It approximates the set of possible strings that the program *may* generate for a particular string variable at a particular program location of interest; these locations are called *hotspots*. The string analysis produces a finite state automaton (FSA) that conservatively approximates the set of possible strings for each hotspot specified; that is, the automaton accepts a larger set of strings than that is actually produced by the program, for that hotspot. In our earlier example,the statement

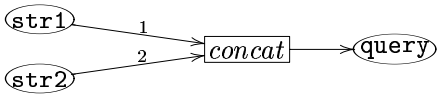
```
return  statement.executeQuery(query);
```

is a hotspot for that program.

The string analysis works on Java bytecode. It starts by finding the hotspots in the Java program. We simply mark, in the program, every location with a call to the method `executeQuery` (such as `return statement.executeQuery(query)` in our example) as a hotspot. Then the analysis abstracts away the control flow of the program, and creates a *flow graph* representing the possible string expressions. The flow graph captures the flow of strings and string operations in a program; all else is abstracted away. The nodes in a flow graph correspond to variables or expressions in the program, and the edges represent directed *def-use relationships* for the possible data-flow. For example, for the statement

```
query = str1 + str2;
```

the following graph nodes and edges are created:



In the graph, the node labeled "*concat*" represents the concatenation expression `str1 + str2`, with the edges labeled `1` and `2` the corresponding first and second arguments.

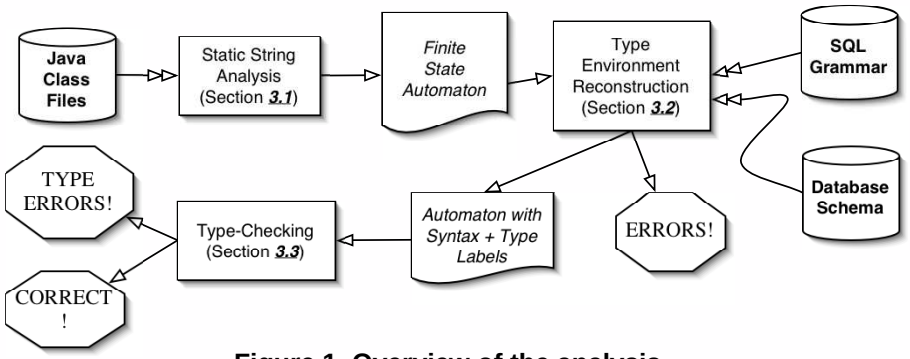


Figure 1. Overview of the analysis.

The edge between nodes labeled "*concat*" and "`query`" indicates the assignment. The other expressions and operators are treated similarly; full details can be found in [6]. Next, this flow graph is reduced to an extended context-free grammar [1] by treating the nodes of the flow graph as terminals and nonterminals of the grammar. For example, the flow graph given earlier in this section yields the following grammar rules:

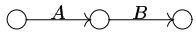
$$\begin{array}{lcl} C & ::= & S_1\ S_2 \\ Q & ::= & C \end{array}$$

where S_1, S_2, C, and Q correspond to the respective nodes for `str1`, `str2`, *concat*, and `query`. In general, the grammar generated from a flow graph is not regular (not even context-free as mentioned earlier). To make further analysis computationally tractable, we widen this grammar to a regular language. The widening step allows syntax checking of the generated strings against a grammar. [2] Since the (fairly technical) details of this step are not the main focus of this work, we omit them here, and refer the interested readers to [6].

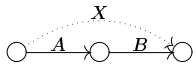
2.2. Context-Free Language Reachability

In the next step, the FSA is processed by a context-free language (CFL) reachability algorithm which forms the foundation of our analysis. We give a brief description of the problem and the algorithm here (cf. [14, 15] for details). The CFL-reachability problem takes as inputs a context-free grammar G with terminals T and nonterminals N, and a directed graph D with edges labeled with symbols from $T \cup N$. Let S be the start symbol of G, and $\Sigma = T \cup N$. A path in the graph is called an *S-path* if its word is derived from the start symbol S. The CFL-reachability problem is to find all pairs of vertices s and t such that there is an S-path between s and t.

The algorithm to solve the CFL-reachability problem uses dynamic programming, and also relates to dynamic transitive closure [23], [3] which underlies many standard program analysis algorithms such as type systems based on subtyping, alias analysis, and control-flow analysis [2,3,17]. The algorithm first normalizes the grammar G such that each production's right-hand side contains at most two symbols. This is easily done by introducing new nonterminal symbols. Then new derived edges are added to D based on the productions of G. For example, suppose G has the production $X ::= A\ B$, and A contains the following edges:



The algorithm adds a dotted edge:



The algorithm repeatedly applies the above transformation to the graph D until no more new edges can be added. Any pair of nodes s and t with an edge labeled X in the final graph has an X-path from s to t in the original graph D. The running time of the algorithm is cubic in both the size of the alphabet and the size of the graph, *i.e.*, $O(|\Sigma|^3|D|^3)$.

We make use of CFL-reachability in two distinct phases of our analysis, as we explain next.

2.3. Overview of Our Analysis

Figure 1 gives an overview of our analysis. There are two main steps. In the first step (detailed in Section 3.1), we generate a finite state automaton to conservatively approximate the set of object-programs. In the second step,

1 This extension handles operators or functions, if any.

2 By contrast, the grammar is *narrowed* to a finite state automaton in [6] for syntax checking. This is done because, in general, checking the containment of a regular language by a context-free language is undecidable [9]. However, it is possible to work directly with the grammar from the fbw graph or with one widened to a context-free grammar because the intersection of a context-free grammar with a regular expression is still context-free, and furthermore, it is easy to check the emptiness of a context-free grammar.

3 The problem is to maintain transitive closure of a graph while new basic graph edges can be added during graph closure.

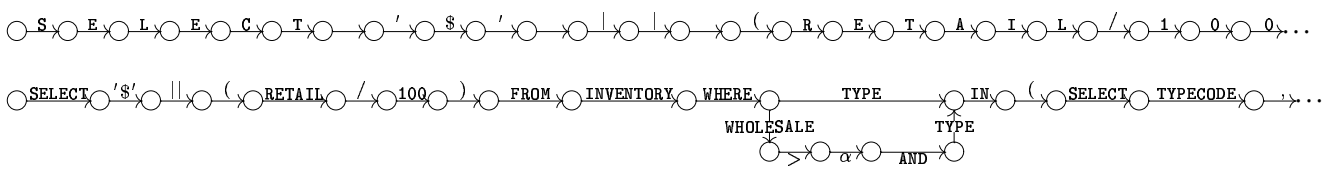


Figure 2. Automaton transformation illustrated.

we process this automaton in two sub-stages. First, we apply CFL-reachability, using SQL grammar, to find scoping information and typing contexts (Section 3.2). Second, we apply CFL-reachability again, using the database schema, to perform type-checking (Section 3.3). Semantic errors, if found, are reported during both phases. Note that our analysis differs from a standard SQL type-checker, which analyzes a single query at execution time. We statically analyze a potentially infinite set of queries.

3. Main Steps of Our Analysis

In this section, we present the major components and steps of our analysis, and illustrate them with our working example from Section 1.

3.1. Automaton Generation and Transformation

In this first step, we apply the string analysis in [6] to generate, for each hotspot in the program, an FSA representing the possible set of query strings that the hotspot can have. The transitions of the automaton are over single letters from the alphabet of the source language. For convenience, we perform a simple compaction on the automaton, so that all transitions are over keywords, delimiters, or literals in the object-language.

Consider again the example from Section 1. Figure 2a shows a fragment of the automaton that the string analysis in [6] produces. After our transformation of the automaton, we have the FSA shown in Figure 2b. [4] To achieve this, we use a depth-first traversal of the original automaton, which groups letters into tokens (in the same sense as those in the lexical analysis phase of a compiler [1]). We then use these tokens to create an equivalent FSA with transitions over the keywords, literals, and delimiters of our object-language. In addition, white-spaces are removed from the automaton in this step.

3.2. Reconstruction of Type Environments

For an SQL query, the declared types of various columns are given in a database schema. This is similar to the notion of a type environment in standard type systems for language such as C, Java, and ML to look up types of variables. To illustrate, consider the sample SQL query

4 In the figure, α denotes an unknown string.

Figure 3. An FSA with two table contexts.

```
SELECT NAME FROM EMPLOYEE WHERE SALARY > 20,000.
```

The information that `NAME` is of type `varchar`, and `SALARY` is of type `integer` is not explicit in the above query expression, but stored separately in the table `EMPLOYEE`'s schema. For the same reason, our generated FSA does not have this information either. We need to reconstruct it from the database schemas. We now describe how we use the CFL-reachability analysis to obtain the column-name to type mapping from the schema.

In this step, we assume that the generated FSA is syntactically correct, *i.e.*, the query strings produced by the FSA are all of valid SQL syntax. This assumption is enforced by the string analysis [6] because it performs syntax-checking of the generated automaton.

The type environment reconstruction for the FSA is nontrivial; the type of any given column depends upon its context, *i.e.*, where it occurs. Depending on the structure of the FSA, a given column may appear in many contexts. For example, suppose we have the automaton shown in Figure 3. The type of the column `NAME` can be different depending on which one of the two paths in the automaton is taken. In one path, its type is determined by the schema's definition of `TABLE1`; in the other, it is determined by that of `TABLE2`.

Our solution to this problem is based on a variant of the CFL-reachability algorithm. We apply the algorithm with the context-free grammar for SQL queries and our transformed automaton as inputs. In essence, we use the CFL-reachability algorithm to parse the automaton. This is very similar to general context-free parsing (which is also of cubic time complexity). However, instead of parsing a particular query, we work with an automaton which produces potentially infinite number of query strings.

In Table 1, we show an excerpt of the grammar we use for SQL's `SELECT` statement [7]. Nonterminals are in *italic*, and terminals are using the `typewriter font`. Our grammar is not yet complete, but could be easily made so by adding more rules. The CFL-reachability algorithm described in [14] requires a normalized grammar such that the right-hand side of any production has at most two symbols.

Nonterminal		Productions
select_stmt	::=	*select* `;`
select	::=	*select_part1 select_part2*
select_part1	::=	`SELECT` *column_list*
select_part2	::=	`FROM` *table_list*
	|	`FROM` *table_list where_clause*
where_clause	::=	`WHERE` *condition*
condition	::=	*logical_term*
	|	`NOT` *logical_term*
	|	*condition* `OR` *logical_term*
	|	*exp_simple* `IN` *subquery*
. . . Additional rules elided		

Table 1. SQL `SELECT` statement grammar.

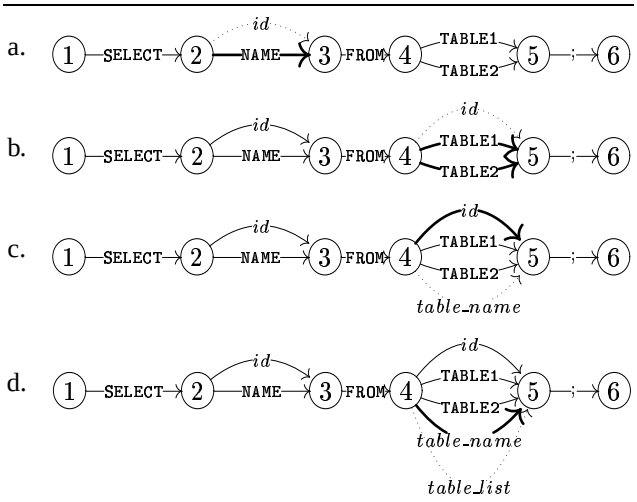


Figure 4. Sample steps in discovering type environments.

Our implementation of CFL-reachability has been extended so that it works with productions with at most three symbols on their right-hand sides. This extension allows us to use more naturally written grammars.

Our modified CFL-reachability algorithm enables us to find the type context (*i.e.*, type environment) of each path through the automaton. We can then use this information to match every column with all of its possible types. The type contexts are discovered by annotating the automaton with the derivation in use while running the CFL-reachability algorithm. In particular, whenever the CFL-reachability process adds a nonterminal edge to the automaton, we store references in that edge to the edges making up this derivation. This is similar to actions in syntax-directed translation [1]; we build a collection of parse trees for the automaton. In fact, the complete type environment reconstruction step is similar to attribute grammars in syntax directed translation [1].

To illustrate this process, we will run through a few steps using an example. First, let us return to the simple example in Figure 3. Here are a few steps (shown in Figure 4) of running our algorithm for discovering the type environments:

Figure 4a. Because `NAME` is not a keyword, it must be an identifier, so an edge labeled *id* is added between nodes 2 and 3.

Figure 4b. The same goes for the two edges between nodes 4 and 5. However, the *id* edge from node 4 to 5 has two distinct derivations. This is different from the standard CFL-reachability algorithm. With the standard algorithm, if an edge to be added is already present, then nothing needs to be done for that edge. In our example, suppose there is already an *id* edge from 4 to 5, which was added through the edge labeled `TABLE1`. When the edge labeled `TABLE2` is to be processed, another *id* edge from 4 to 5 is to be added. However, the edge is already present in the automaton. Instead of simply stopping (as is done in the standard CFL-reachability algorithm), our algorithm adds a second derivation reference to the already-present *id* edge. This allows each context to be discovered when searching through the derivation edges. Notice that we can handle loops in the automaton by exploiting its simple looping structure. Due to space limitations, we omit the details here.

Figure 4c. After these *id* edges are added, an edge labeled *table_name* is added from node 4 to node 5, which has a *single* derivation—the *id* edge from node 4 to node 5.

Figure 4d. Then, an edge labeled *table_list* will be added from node 4 to node 5. Next, an edge labeled *select_part2* will be added from node 3 to node 5. This process continues, and eventually an edge labeled *select_stmt* is added from node 1 to node 6 (final graph not shown due to space limitations).

Now let us go back to our example in Section 1. After running our type environment reconstruction algorithm on this example, there is an edge from the start state of the automaton to its final state, labeled *select_stmt*. The edge has a single derivation with two parts: a *select* edge and an edge labeled with the delimiter `;`. The *select* edge also has a single derivation: a link to a *select_part1* edge and a link to a *select_part2* edge. These links form different parse trees for the automaton. By a top-down traversal of parse trees, we can determine which tables apply to which column lists, and with the schema for these tables, we can determine the possible types of each column. Then, we add type edges to the graph, replacing column names with their corresponding types. Notice, because of the possibility of multiple contexts, a column may have more than one type. In our example, we know that `RETAIL` is a column in the ta-

Nonterminal		Productions
integer	::=	*integer* + *integer*
	|	*integer* - *integer*
	|	*integer* * *integer*
	|	(*integer*)
	|	ABS *integer*
decimal	::=	(*decimal*)
	|	*integer* / *integer*
varchar	::=	(*varchar*)
	|	UPPER *varchar*
	|	*varchar* || *varchar*
boolean	::=	*integer compare_op integer*
	|	*integer* IN (*integer*)

Table 2. SQL `SELECT` statement type-system grammar

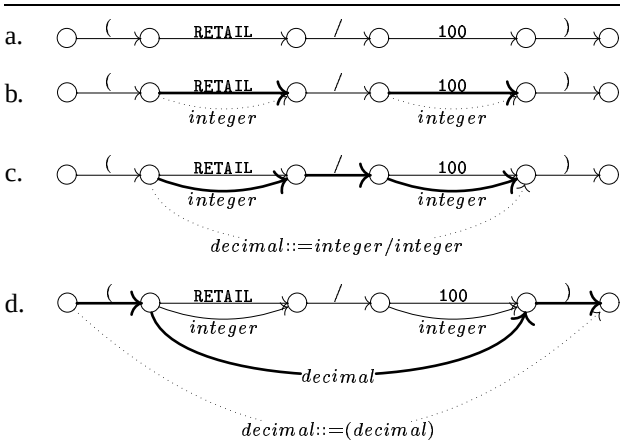


Figure 5. Sample steps of running CFL-reachability using the type grammar.

ble `INVENTORY`, so we add an edge labeled `integer` to the graph. When we come across a primitive such as the edge labeled `100`, we determine that it is not a column name, and must be a literal. At this point, we determine the literal's type and add it to the graph as an edge.

In type environment reconstruction, some errors can be discovered. For instance, we can determine whether there is an invalid column that does not exist in any of the applicable tables for that column. If this happens, an error can be reported as either an improperly-quoted literal, or a non-existent column. Other errors are also detected in this step, including duplicate table references (the same table appears more than once in the `FROM` clause), duplicate uses of the same table alias (two tables are assigned the same alias), and non-existent tables (the schema does not have a table referenced in the `FROM` clause).

3.3. Type-Checking

In the final step of the analysis, we perform type-checking on the automaton produced in the previous step, as described in the previous section. At this stage, the automaton has been annotated to show the types of column names and literals. SQL's simple type system lets us treat the type system as a context-free grammar. For example, the type rule for additions over integers looks like:

$$\frac{\Gamma \vdash e_1 : integer \qquad \Gamma \vdash e_2 : integer}{\Gamma \vdash e_1 + e_2 : integer}$$

The above rule can be viewed as equivalent to the grammar rule: $integer ::= integer + integer$, which states that an integer plus an integer is again an integer. The other rules for type-checking SQL expressions can be handled in a similar manner. This is possible due to SQL's simple type language—a collection of atomic types. This is in contrast to general purpose programming languages that have more complicated type structures. Table 2 shows a small subset of the context-free grammar for the type-checking rules of SQL's `SELECT` statement. All the rules are straightforward, except the last one, which merits some explanation. The rule says that the conditional expression `IN` is well-typed if the sub-query inside the parentheses (here required to be of type `integer`) reduces to a single column of type `integer`, when the outer expression is an integer. Thus, this rule requires the sub-query to be of the correct type. A similar rule, not shown here, exists for the `varchar` type. Notice that we did not specify all the rules, for example the rules for the `SELECT` statement. These rules are obvious, and we omit them in this paper due to space limitations. As in a standard type system, if none of the rules apply for a language construct, then a type error is discovered.

We apply the CFL-reachability algorithm using the grammar in Table 2 to propagate type information. If during the process, there is an expression that does not match any one of the right-hand-sides of the rules, then an error is discovered. In some sense, there are implicit *error rules* in the grammar, such as $errortype ::= integer + varchar$.

We illustrate type-checking with an example. Consider the small snippet shown in Figure 5a taken from our working example in Section 1. Before type-checking begins, the automaton is annotated with type information for the column names and literals, as shown in Figure 5b. Figures 5c-d show a few steps of type propagation using our grammar rules.

If our type-checking step does not produce any edges labeled *errortype*, then all the object-programs specified by the automaton are type-correct. On the other hand, if type-checking does produce an *errortype* edge, the analysis reports potential errors and displays a sample derivation that causes the type error. Note, however, that a reported error

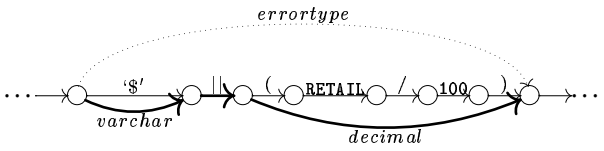


Figure 6. Discovering a type error.

may not be an actual error in the original Java program due to imprecision in the automaton characterization of the object-programs. In the case of SQL, our analysis is precise under the assumption that all the object-programs specified by the automaton are feasible in the original Java source program.

Figure 6 shows the edge *errortype* being added to a snippet of our example program, corresponding to the concatenation error between the character `'$'` and the numeric result of the division. Note that many irrelevant edges are omitted in the figure. The two other errors present in our example can also be discovered in a similar manner.

3.4. Correctness of the Analysis

We now state and briefly argue the soundness of our analysis. Due to conservative approximations, our analysis may report a spurious (infeasible) error. In Section 4, we present experimental data to support the claim that the analysis is rather precise and has low false-positive rates.

Theorem 3.1 (Soundness) *Our analysis is sound. In other words, if the analysis does not report any errors, then the generated object programs are type-safe.*

Proof. [Sketch] We give a brief justification of the soundness theorem. We assume that the automaton that we operate on is a conservative approximation of the set of possible SQL query strings for a particular hotspot. This is guaranteed by the correctness of the string analysis in [6]. We also make the assumption that the query strings produced by the automaton are syntactically correct, [5] which is crucial for the soundness of our analysis. Next, we can show that CFL-reachability considers all possible derivations of the query strings because the query strings are all of the correct syntax. Thus, the type-environment reconstruction step is correct, meaning that a column name is considered in all possible tables. For example, if we annotate an edge labeled by a column with the type integer, there must be a path in the automaton containing that edge such that the particular column is found to be of type integer. At the end of this step, all columns are labeled with a superset of its actual types. Finally, if the resulting automaton contains a path with a type error, it will be detected because CFL-reachability considers all possible edge combinations. □

5 This can be enforced by the string analysis in [6].

4. Experimental Evaluation

We have built a prototype tool, embodying our approach, and have tested its ability to detect programming errors in Java/JDBC applications. As any SQL developer will attest, every database vendor implements a different version of SQL; thus checkers such as ours require some porting effort for each different database. We have implemented our analysis for the `SELECT` statement specified by the grammar for Oracle version 7 [7]. This grammar is a subset of what is specified in the SQL-92 standard. Adding support for other statements or different vendors is not difficult, because we have separated the type environment reconstruction and type-checking steps. In most cases, we would simply need to modify our syntax grammar and/or type-system rules (specified as input files to our analysis) and the recursive-descent code to traverse the parse trees, and map column names with their possible types. With the goal of having a sound analysis, we have built a strict semantics into our tool: if a program is deemed type-safe by our analysis, it should be type-safe on any database system. Because the semantics of many database systems is not as strict as the one enforced by our tool, the tool may report an error which some database systems consider legitimate.

Our tool is implemented in Java and uses the string analysis in [6] for computing the FSA, which in turn uses the Soot framework [20] to parse class files and compute interprocedural control-flow graphs. We have tested our tool on various test programs, including student team projects from an undergraduate software engineering class, sample code from online tutorials found on the web, and code from other projects made available to us. Table 3 lists the test programs and summarizes our results. For each test program, we list the Java source code size (number of lines of source code), number of hotspots in the program, number of columns in the database schema, generated automaton size (number of edges and nodes), analysis time (split into automaton generation and semantic analysis), numbers of various warnings and errors found (cf. Table 4). Note that the test programs are sorted by automaton size, since it is a good measure of the complexity of the programs for our analysis. All experiments were done on a machine with a 2GHz Intel Xeon processor and 1 GB RAM, running Linux kernel 2.4.20. The results indicate that our analysis is rather precise, *i.e.*, with low false-positive rates. Because our analysis is sound, if the tool does not report any error on a program, then we have verified that the program is type-correct. In addition, although we have not tuned the performance of our implementation, the analysis is still quite efficient; it was able to analyze each of our test programs within a matter of minutes. We expect our analysis to scale to large systems, because analyses based on the same underlying algorithms have been shown to scale to millions of source lines of C [8, 18]. Further experiments are needed to verify our

Test Programs (S) Student (W) Web Download (I) Industrial	Size				Analysis Time (sec)		Errors Found			
	Java Program (Lines)	Hotspots	Schema (Columns)	Total Automaton Edges/Nodes	Automaton Generation	Semantic Analysis	Warnings	Total Errors	Confirmed Errors	False Errors
Smi (S)	1559	1	19	35 / 27	1.5	7.1	0	0	0	0
CFWorkshop (S)	36	5	13	47 / 52	0.6	1.3	0	0	0	0
TicTacToe (S)	2888	2	26	134 / 121	6.4	144.8	3	1	1	0
WebBureau (W)	50	10	21	152 / 162	0.5	2.5	0	1	1	0
Checkers (S)	6615	4	36	181 / 138	11.8	97.8	0	15	15	0
JuegoParadis (S)	6135	13	29	259 / 206	27.0	45.0	0	9	0	9
Reservations (S)	2385	22	54	368 / 383	1.7	29.1	0	0	0	0
OfficeTalk (S)	5812	29	14	655 / 525	7.0	120.8	0	2	2	0
PurchaseOrders (I)	642	51	82	1324 / 1373	1.3	173.3	41	10	9	1

Table 3. Experimental results.

Error Kind	Description	test programs
Type Mismatch	Concatenation of fields with wrong types.	PurchaseOrders
Type Mismatch	Possibly unquoted string compared with a varchar column.	PurchaseOrders
Type Mismatch	Quoting a numerical value and treating it numerical.	Checkers, Office-, Talk, TicTacToe
Semantic Error	Ambiguous column selection.	WebBureau
Semantic Error	Column not found.	Checkers, PurchaseOrders
Spurious Error	Column not found (due to imprecision of the string analysis).	PurchaseOrders, JuegoParadis
Warning	Comparing a numerical value with a possibly non-numerical value.	PurchaseOrders, TicTacToe

Table 4. Breakdown of errors and warnings.

claim.

Table 4 shows a breakdown of the kinds of errors that we found in the test programs. We next explain these errors in more detail:

Concatenation of fields with wrong types. This is the same error as in the concatenation `'$' || (RETAIL/100)` (Section 1). After discovering this error in porting a program to a different (more strict) database, our tool has been used to find all instances of the error in the "PurchaseOrders" program.

Possibly unquoted string. Assume we have a comparison such as `NAME =` α, where α represents an unknown string. If there are no quotes in a string that α possibly represents, then such an error occurs.

Quoting a numerical value. This happens when a numerical value is quoted but still treated as a numerical literal. This is a common error found in student projects. They were using MySQL, which permits numerical literals to be quoted. Many other database systems consider this an error because quoted numerical literals are of type `varchar`.

Ambiguous column selection. Our tool detected such an error in some sample code from a tutorial website (`http://web-bureau.com/modules/sql.php`). This error is quite subtle, and it appears unknown. The particular statement is:

```
SELECT customer_id FROM customers c, orders o
       WHERE c.customer_id = o.customer_id;
```

The error is that the database does not know which table's `customer_id` to choose. Certainly, it seems not matter which `customer_id` to select in this particular statement, but in general, the semantics of the column list should not depend on the outcome of the `WHERE` clause.

Column not found. This error happens when a column name does not exist in any of the tables in the `FROM` clause. We found two distinct causes of this error—one a real error and the other a spurious error:

Real error The schema of the database does not include this column. This can be caused by either selecting a nonexistent column, or missing the quotes around a literal, and thus being treated as a column.

Spurious error This is due to the imprecision in the string analysis. Consider the following example, where `makeQuery` is a public method taking a string parameter `tables` to construct a `FROM` clause:

```
public String makeQuery(String tables) {
     return "SELECT name FROM " + tables;}
```

The string analysis adds an α edge to the table list of the `FROM` clause (meaning any table is possible), in addition to the concrete table list it finds through analyzing other input classes. The reason is that this method is public, and the string analysis expects other calls to the method possible and thus views the input classes incomplete. The presence of this α edge causes the analysis to search for the column in an empty table list, which certainly fails. This is the only kind of spurious errors we found in our test programs. These errors can easily be filtered out by modifying the string analysis to consider its input as a complete set of classes

Warning. We found one type of warning in our test programs. It is the same as the one illustrated in our running ex-

ample in Section 1: to compare a numerical column with a *possibly* non-numerical value at runtime.

We have shown that our tool can detect errors in non-trivial programs. To further evaluate our tool and to provide a programming aid to students, we plan to experiment with using the tool in an undergraduate software engineering class.

5. Related Work

In this section, we survey closely related work. Perhaps the most closely related is the string analysis of Christensen, Møller, and Schwartzbach [6] that forms the basis of our analysis. Their string analysis ensures that the generated object-programs are syntactically correct. However, it does not provide any semantic correctness guarantee of the object-programs. Many domain specific languages have been proposed for ensuring correctness of dynamically generated web documents. Most of these are language extensions enhanced with tree manipulation capabilities, instead of string manipulations that we deal with in this work. In addition, they usually provide only guarantee of correct syntax; semantic correctness is not guaranteed. We mention two research efforts in static validation of dynamically generated web documents such as HTML. In [16] the authors propose a typed, higher-order template language that provides safety of the dynamically generated web documents within the `<bigwig>` project [5], an extension to Java for high-level web service development. Their type system is based on standard data-flow analysis techniques [10, 12]. Another work along the same lines is the work by Braband, *et al* [4] to statically validate dynamically generated HTML documents against the official DTD for XHTML. The work again is based on a data-flow analysis that computes a summary of all possible documents at a particular point of the program. The summary graph is then validated against the official DTD for XHTML. This work is again done in the context of the `<bigwig>` language. In [11], a test adequacy criterion for data-intensive applications is presented. Our approach is complementary; static analysis can save testing time, but testing can discover logical defects not related to SQL query construction.

To be put in a broader context, our research can be viewed as an instance of providing static safety guarantee for meta-programming [19]. Macros were the earliest meta-programming technique, where the issue of correctness of generated code first arose. Macro programmers using powerful macro programming languages clearly need to worry about the correctness of the generated code. How can such macro meta-programs be statically checked for correctness? The widely used `cpp` macro pre-processor does little checking, and allows one to write arbitrary macros, without regard to correctness. The programmable syntax macros of Weise & Crew [22] work at the level of correct abstract-syntax tree (AST) fragments, and guarantee that generated code is syntactically correct with respect (specifically) to the C language. Static type-checking is used to guarantee that AST fragments (e.g., Expressions, Statements, and etc) are assembled correctly by macro meta-programs. Another issue is scoping of generated names: macro expansion should not "capture" variable names in an unexpected manner. Hygienic macro expansion algorithms, beginning with Kohlbecker *et al* [13] provide these guarantees. More recent work seeks to extend syntactic and scoping guarantees, with *semantic guarantees* for the generated code. The work of Taha & Sheard [19] and others is concerned with (in a functional programming setting) guaranteeing that generated code is type-safe. We do not introduce a new macro language, like [22], nor work in a uniform functional setting, like [19], with functional languages both at meta- and target-levels. Our goal is simply to ensure that strings passed into a database from an arbitrary Java program are type-safe SQL queries from the perspective of a given database schema. We expect that the general technique outlined in this paper can be extended to apply in other settings as well.

6. Conclusions and Future Work

We have presented a sound, static analysis technique for verifying the correctness of dynamically generated SQL query strings in database applications. Our technique is based on applications of a string analysis for Java programs and a variant of the context-free language reachability algorithm. We have implemented our technique and have performed extensive testing of our tool on realistic programs. The tool has detected known and unknown errors in these programs, and it is rather precise with low false-positive rates on our test programs.

For future work, there are a few interesting directions. To increase the usability of our tool for debugging, it would be interesting to map an error path that we find in the automaton to the original Java source. One possible approach is to carry line numbers of the flow-graph nodes in the source code to the automaton, so that a path in the automaton can be associated with a set of source lines in the original Java program. A related problem is to check the correct uses of the query results in the source program. Consider the following example:

```
query = ''SELECT NAME, SALARY FROM EMPLOYEE'';
rset = statement.executeQuery(query);
...
salary = rset.getInt(1);  // should be getInt(2)
name = rset.getString(2); // should be getString(1)
```

The result set `rset` is a table of pairs of type `String` (the `NAME` column) and `int` (the `SALARY` column). The statement `salary = rset.getInt(1)` attempts to read the

first field of the pair, a string, and treat it as an integer. The last statement has a similar error. By mapping our analysis results back to the original program, we can detect this class of errors. The class of embedded SQL command injection problems [21] in web and database applications is also interesting to look at. The problem is, without proper input validation, an attacker can supply arbitrary code to be executed by a web or database server, which is extremely dangerous. We plan to extend our approach to deal with this class of errors. Finally, we have considered SQL so far in this paper. We believe our technique is general and plan to investigate how to extend it to analyze dynamically generated programs in other languages.

Acknowledgments

We thank Anders Møller and Aske Simon Christensen for useful discussions about this research and several students of ECS 160 at UC Davis for providing us with source code of their projects in the evaluation of our tool. Finally, we thank the anonymous reviewers of ICSE 2004 for their valuable comments.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proceedings of the 18th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–118, 1991.

[3] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[4] C. Braband, A. Møller, and M. Schwartzbach. Static validation of dynamically generated HTML. In *ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 38–45, June 2001.

[5] C. Brabrand, A. Møller, and M. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[6] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium*, pages 1–18, 2003.

[7] J. Guyot. BNF index of SQL for Oracle 7. Available at `http://cui.unige.ch/db-research/Enseignement/analyseinfo/SQL7/`.

[8] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[9] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, Reading, MA, 1979.

[10] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, Jan. 1976.

[11] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 98–107, 2003.

[12] G. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM SIGACT and SIGPLAN, 1973.

[13] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. Hygenic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–159, 1986.

[14] D. Melski and T. Reps. Interconvertibility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pages 74–89, 1997.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[16] A. Sandholm and M. Schwartzbach. A type system for dynamic web documents. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 290–301, 2000.

[17] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[18] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[19] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations*, pages 203–217, 1997.

[20] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot–a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference, CASCON'99*. IBM, Nov. 1999.

[21] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison Wesley, 2001.

[22] D. Weise and R. Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.

[23] D. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, July 1993.