Static Checking of Dynamically Generated Queries in Database Applications

14

GARY WASSERMANN, CARL GOULD, ZHENDONG SU,
and PREMKUMAR DEVANBU
University of California, Davis

Many data-intensive applications dynamically construct queries in response to client requests and execute them. Java servlets, for example, can create strings that represent 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 article, we present a *sound*, *static* program analysis technique to verify that dynamically generated query strings do not contain type errors. 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.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability, validation, formal methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms: Algorithms, Languages, Reliability, Verification

Additional Key Words and Phrases: Static checking, JDBC, database queries, context-free language reachability

An earlier version of this article was published as GOULD, C., SU, Z., AND DEVANBU, P., Static checking of dynamically generated queries in database applications, In *Proceedings of the International Conference on Software Engineering (ICSE)* (Edinburg, Scotland, May). ACM, New York, 2004, 645–654.

This research was supported in part by an NSF CAREER Grant (No. 0546844) and a generous gift from Intel.

The information presented here does not necessarily reflect the position or the policy of the Government and no officle endorsement should be inferred.

Authors' address: Department of Computer Science, University of California, Davis, CA 95616-8562, email: {wassermg, gould, su, devanbu}@cs.ucdavis.edu.

ACM Reference Format:
Wassermann, G., Gould, C., Su, Z., and Devanbu, P. 2007. Static checking of dynamically generated queries in database applications. ACM Trans. Softw. Eng. Methodol. 16, 4, Article 14 (September 2007), 27 pages. DOI = 10.1145/1276933.1276935 http://doi.acm.org/10.1145/1276933.1276935

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 this article 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 `getPerishable-TypeCode` 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');
```

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. However, 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 article, 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 [Hopcroft and Ullman 1979], and a variant of the context-free language (CFL) reachability problem [Reps et al. 1995; Melski and Reps 1997]. As a first step, our analysis builds upon a static string analysis to construct 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, we observe a low false-positive rate in practice, so empirically our analysis is quite precise.

The rest of the article is structured as follows. We begin with background on the string analysis 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

We now describe the technical context of our work.

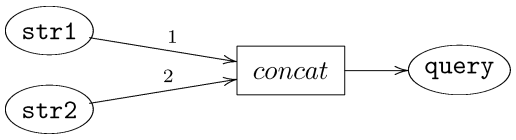
2.1 Static String Analysis of Java Programs

As mentioned earlier, our analysis makes use of a static string analysis of Java programs [Christensen et al. 2003]. Essentially, it is an interprocedural data-flow analysis [Kildall 1973; Kam and Ullman 1976] to approximate the semantics of string manipulation expressions of a program. The analysis is similar to a pointer analysis [Andersen 1994] for imperative languages or a control-flow analysis (0-CFA) [Shivers 1988] 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 following 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; everything 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` corresponding to the first

and second arguments. 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 Christensen et al. [2003]. Next, this flow graph is reduced to an extended context-free grammar by treating the nodes of the flow graph as terminals and nonterminals of the grammar. The extension handles operator or functions on strings, if any.[1] For example, the flow graph given earlier in this section yields the following grammar rules:

$$C ::= S_1\ S_2$$
$$Q ::= C$$

where S_1, S_2, C, and Q correspond to the respective nodes for `str1`, `str2`, *concat*, and `query`.

In general, the grammar derived from a flow graph is not regular (not even context-free as mentioned earlier). To make further analysis computationally possible, we widen this grammar to a regular language. As an example, the widening step would change a language such as $\{(^n\mathtt{a})^n\}$ to $\{(^*\mathtt{a})^*\}$. The purpose of the widening step is to allow syntax checking of the generated strings against a context-free grammar. In practice, we do not find a function that concatenates some string, the return value of a recursive call to itself, and another string (which would construct a language such as $\{(^n\mathtt{a})^n\}$), so this widening step does not hurt the precision of the analysis.

In addition, we need to *narrow* (i.e., find an under-approximation of) the SQL grammar to a regular language for syntax checking. As an example, if the SQL language were $\{(^n\mathtt{a})^n\}$, it could be narrowed to $\{\mathtt{a}|(\mathtt{a})|((\mathtt{a}))\}$. The narrowed language contains a subset of the strings in the original language. This needs to be done because, in general, checking the containment of a regular language by a context-free language is undecidable [Hopcroft and Ullman 1979]. In summary, we ensure that all generated strings are syntactically correct by inferring a grammar that over approximates the set of generated strings, widening that grammar to a regular language R_{gen}, and checking whether R_{gen} is a subset of a narrowed regular approximation of the SQL grammar.

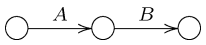
The subsequent steps of our algorithm assume that all strings in R_{gen} are syntactically correct. Since the details of how the widening and narrowing are accomplished algorithmically are not the main focus of this work, we omit them here, and refer the interested readers to Christensen et al. [2003]. We note, however, that the widening step is not necessary in principle. It is possible to work directly with the grammar from the flow graph or with one widened to a context-free grammar because of two well-known results: (i) The intersection of a context-free language with a regular language is still context-free; and (ii) It is straightforward to check the emptiness of a context-free language. We use the string analyzer as a library or a black-box in our tool, so we have not implemented these potential modifications. We have not found the narrowing step to be a limitation because, in practice, generated queries do not use deeply nested parentheses.

[1]For the details on this step, the interested reader is referred to Christensen et al. [2003].

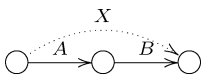
2.2 Context-Free Language Reachability

In the next step, the FSA is processed by a context-free language (CFL) reachability algorithm that forms the foundation of our analysis. We give a brief description of the problem and the algorithm here (cf. Reps et al. [1995] and Melski and Reps [1997] for 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 [Yellin 1993],[2] which underlies many standard program analysis algorithms such as type systems based on subtyping, alias analysis, and control-flow analysis [Andersen 1994; Shivers 1988; Amadio and Cardelli 1991]. 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 D contains the edges respectively labeled A and B:



The algorithm adds a dotted edge labeled X:



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.

3. OUR ANALYSIS

Figure 1 gives an overview of our analysis. There are two main steps. In the first step (outlined in Section 3.1), we generate a finite state automaton to conservatively approximate the set of object-programs. In the second step, we process this automaton in two sub-stages. First, we apply CFL-reachability, using the

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

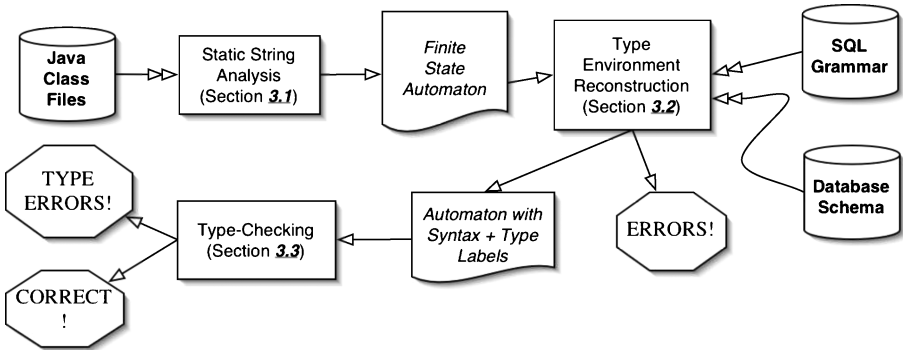


Fig. 1. Overview of the analysis.

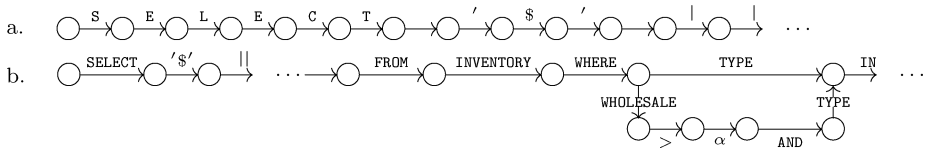


Fig. 2. Automaton transformation illustrated.

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.

We now 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 Java string analysis we mentioned earlier [Christensen et al. 2003] 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 Java string analysis produces. In the figure, we use "..." to indicate some omitted details of the automata. After our transformation of the automaton in Figure 2a, we obtain the FSA in Figure 2b (where α denotes an unknown string). 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 [Aho et al. 1986]). We then use these tokens to create an equivalent FSA with transitions over the keywords, literals, and

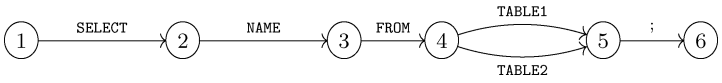


Fig. 3. An FSA with two table contexts.

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 following sample SQL query:

```
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 [Christensen et al. 2003] 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 input. 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 that produces a potentially infinite number of query strings.

In Table I, we show the grammar we use for SQL's `SELECT` statement [Guyot 1998]. Nonterminals are in *italic*, and terminals are shown 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 Melski and Reps [1997] requires a normalized grammar such that the right-hand side of any production has at most two symbols. 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

Table I. SQL SELECT Statement Grammar.

Nonterminal		Productions
select_stmt	::=	*select* ;
select	::=	*select_part1 select_part2*
select_part1	::=	SELECT *column_list*
	|	SELECT DISTINCT *column_list*
	|	SELECT ALL *column_list*
select_part2	::=	FROM *table_list* | FROM *table_list where_clause*
column_list	::=	*displayed_col* , *column_list* | *displayed_col* | *
displayed_col	::=	*exp_simple* | *exp_simple* AS *id*
exp_simple	::=	*exp_simple addop term* | *term* | *addop term*
term	::=	*factor* | *term multop factor* | *term* || *factor*
factor	::=	*id* | (*exp_simple*)
	|	*function func_paren* | *group_function group_func_paren*
function	::=	UPPER | LOWER | ABS | LENGTH
func_paren	::=	(*exp_simple*) | (*func_paren_double*)
func_paren_double	::=	*exp_simple* , *exp_simple*
group_function	::=	AVG | COUNT | MAX | MIN | SUM
group_func_paren	::=	(*exp_simple*) | (*)
table_list	::=	*table_list* , *table_name* | *table_name*
table_name	::=	*id* | *id id* | *id* AS *id*
where_clause	::=	WHERE *condition*
condition	::=	*logic_term* | NOT *logic_term* | *condition* OR *logic_term*
logic_term	::=	*logic_factor* | *logic_term* AND *logic_factor*
logic_factor	::=	*exp_simple compare_op exp_simple*
	|	*exp_simple* IN *subquery*
subquery	::=	(*select*)
addop	::=	+ | -
multop	::=	* | /
compare_op	::=	= | < | >
id	::=	< any non-keyword >

more naturally written grammars—a feature intended to make extending this tool easier.

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 [Aho et al. 1986]; 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 [Aho et al. 1986].

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.

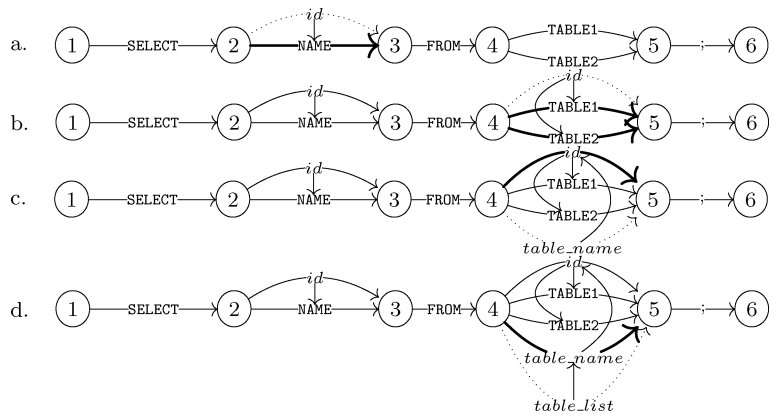


Fig. 4. Sample steps in discovering type environments.

***Figure* 4*b*.** 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.

***Figure* 4*c*.** After the *id* edge is 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* 4*d*.** Then, an edge labeled *table_list* will be added from node 4 to node 5. This process continues until no further edges can be added based on grammar productions.

The purpose of the first application of the CFL-reachability algorithm is to find the types of columns. Finding the type of the columns requires first identifying which tokens are column names, second matching them with the appropriate table names, and finally looking up that column in the database schema to determine its type. Applying CFL-reachability with the SQL grammar essentially builds parse trees for the automaton. We use these parse tree to find which tokens are column names and match them with the appropriate tables. When the types are determined from the schema, type-edges are added to the automaton.

Figure 5 illustrates the final steps of this process. Note that Figure 5 shows the edges that are relevant to the completion of this algorithm, not every edge that has been added. For example, because of the *id* edge from node 4 to node 5,

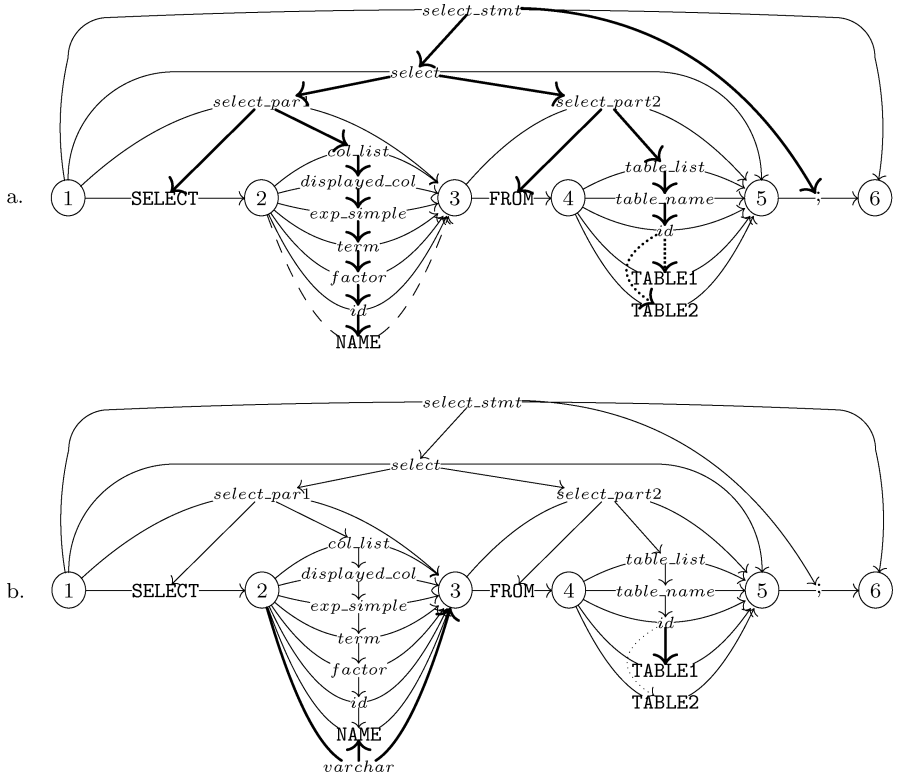


Fig. 5. Making the type environment explicit.

there is also a *factor* edge from node 4 to node 5, which is not shown in the figure.

Figure 5a. At each step of the CFL-reachability algorithm, if a sequences of edges matches the *rhs* of a production, references are added between the edge representing the *lhs* of the production and the edge(s) representing the *rhs* of the productions. These references form a parse tree for each complete query. The references from the *id* edge between nodes 4 and 5 are dotted to show that they represent two distinct matchings rather than a matching from a single production with two symbols on the *rhs*.

Figure 5b. Productions that represent complete statements have associated handlers, which recursively descend parse trees. Handlers are used for descending parse trees and performing a task. In the case of SQL data statements, this task is matching column names with table names. The *select_stmt* production represents a complete `SELECT` statement, and its handler is called on the edge between nodes 1 and 6. The handler first finds `NAME`'s type based on the column types in `TABLE1`—we assume that type is *varchar*. Since no *varchar* edge between nodes 2 and 3 exists, a new one is added with a reference to the `NAME` edge. The same will be done using `TABLE2`. When we come across a primitive (e.g., an edge labeled `100` in the example in Section 1), 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 nonexistent 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 nonexistent 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:

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

The above rule can be viewed as equivalent to the grammar rule: $\mathit{integer} ::= \mathit{integer} + \mathit{integer}$, 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 II 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 subquery 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 subquery 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 our presentation. 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 II 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 $\mathit{errortype} ::= \mathit{integer} + \mathit{varchar}$.

We illustrate type-checking with an example. Consider the small snippet shown in Figure 6a 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 6b. Figures 6c-d show a few steps of type propagation using our grammar rules.

Table II. SQL SELECT Statement Type-System Grammar

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*)

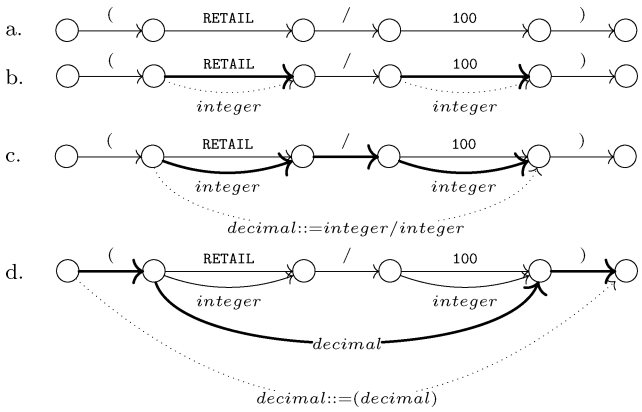


Fig. 6. Sample steps of running CFL-reachability using the type grammar.

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 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 and that no like-named columns have different types.

Figure 7a 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. Figure 7b shows an *errortype* edge being added for the second error in our example program. Figure 7c shows the *errortype* edge for the third error, but we use more than just the type grammar to discover it. Because lists (or tuples) of arbitrary length can be constructed, lists with at least two elements are simply given the type *listtype*. The type-abstracted

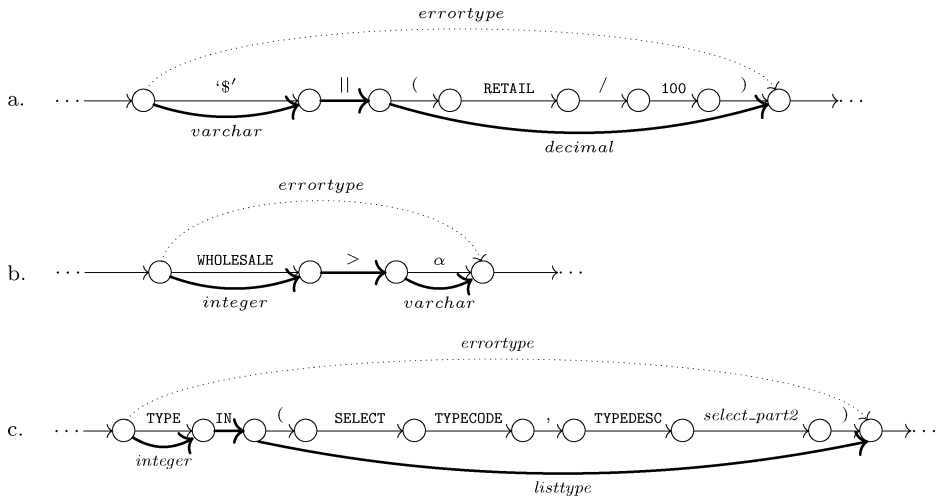


Fig. 7. Discovering a type error.

expression "*integer* `IN` *listtype*" is always an error. If there are two *listtype*'s in the same expression, list-handling code descends the parse trees to check for compatibility.

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 spurious (infeasible) errors. 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 SQL query strings are type-safe.*

PROOF. We justify the soundness theorem here. 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 Christensen et al. [2003]. We also make the assumption that the query strings produced by the automaton are syntactically correct,[3] which is crucial for the soundness of our analysis. If the query strings are guaranteed to be syntactically correct, then the sequences of character transitions that comprise a token in the queries are preceded and followed by token delimiters (e.g., ' ', ',', etc.). This guarantees the generated automaton can be lexed correctly into an automaton over tokens, and the language accepted by the lexed automaton is the same as the language accepted by the original automaton.

Next, we can show by an inductive argument that CFL-reachability considers all possible derivations of the query strings because the query strings are all of the correct syntax. Consider an arbitrary string *s* generated by the automaton.

[3]This can be enforced by the string analysis in Christensen et al. [2003].

Assume that the string has been parsed and that the grammar rules have two symbols on their right-hand sides. Suppose that CFL-reachability has added transitions representing the nonterminals in the grammar to depth n. Consider a node p in s's parse tree at height $n+1$. Because p is at height $n+1$, p's children, at height n, have corresponding edges in the automaton. When the second of these edges is added to the automaton, it was added to a worklist. When it is removed from the worklist, the grammar rule that added p to s's parse tree will be used to add an edge p' corresponding to p to the automaton, if p' does not exist already. References from p' to the edges that allowed it to be added will also be added, if they do not exist already. This argument holds for the base case when $n = 0$ because all edges are initially added to the worklist. The argument generalizes to grammars whose productions have three-symbols on their right-hand sides.

We now argue that the type reconstruction is correct. Consider an arbitrary edge e labeled with a column name. The references between edges added by the CFL-reachability algorithm form parse trees for all complete queries in 2 the graph, so for all queries q generated which use e, there exists an edge r in the graph such that r is at the root of the parse tree for q. Recursive descent code gets called on each edge r which is at the root of the parse tree for a complete statement. For each edge r, the recursive descent code or handler that gets called depends on r's label, for example, if r's label is *select_stmt*, a handler for `SELECT` statements is called. For each unique label that a root edge r may have, the handler is designed to find the table lists in the parse tree and match them with all column names. Therefore, the column named at e will get matched with the appropriate table lists. For each table t in the lists, the database schema tells whether t includes a column with the name that labels e, and if so, what its type is. Thus, if on some path a column name has type τ, the environment recovery phase adds an edge labeled with τ parallel to that column name. At the end of this step, each column is labeled with a superset of its actual types.

Finally, we argue that the type checking phase is correct. Consider an arbitrary binary operator *op* preceded by type τ_1 and followed by τ_2 in the graph. The complete sets of operators and types are enumerated statically and a grammar rule is made out of each possible combination. If τ_1 *op* τ_2 is typeable as τ_3, then an edge labeled τ_3 spanning the three edges it is derived from is added. If τ_1 *op* τ_2 is not typeable, then the grammar production with that sequence on its right-hand side will have *error_type* on its left-hand side, and an edge labeled *error_type* will be added. The addition of such an edge will cause a handler for the *error_type* to be called, and this handler will report an error. This argument also holds for unary operators, parentheses, and all other syntactically possible constructs. Thus, for all possible type errors, an error report will be issued. □

The incompleteness in our analysis comes from two sources. First and primarily, the string analysis may over-approximate the set of possible generated strings. In our experiments (see Section 4), this did result in some false positives. The precision of the string analysis could be improved, but finite state automata lack the expressive power to match precisely an arbitrary set of string generated by a program. The second source of incompleteness in our analysis

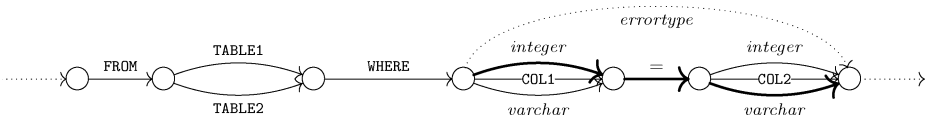


Fig. 8. Imprecision in type checking from CFL-reachability.

comes from decoupling the type environment recovery from the type checking. Consider the example in Figure 8. Suppose that `TABLE1.COL1` and `TABLE1.COL2` are integer columns and `TABLE2.COL1` and `TABLE2.COL2` are varchar columns. The type environment recovery would add type edges accordingly. The type checking phase would find that *integer* = *varchar* is a type error and would report an error even though all of the possible queries type check correctly. We have not encountered this, and we expect that it will be rare in practice. Most operators are defined only for one type or interchangeable types. Also, it seems to be uncommon that a database schema has two columns with the same name but different types. In part, this is because column names often imply column types (e.g., a column named "`TITLE`" probably has type varchar). Having two pairs of columns with the same names but different types would be quite rare.

3.5 Discussions

We now briefly discuss some technical issues in directly applying our technique to more expressive languages than SQL. The division of our semantic checking algorithm into two separate applications of context-free language reachability works well due to the simplicity of SQL and its type system. In fact, we can prove that our analysis is precise assuming that no like-named columns have different types and that the string analysis is precise (and surely the string analysis is not precise and cannot be precise in general). However, in extending our technique to handle more expressive languages, certain imprecisions do show up due to our decoupling of syntax reconstruction and type checking. We use an example to illustrate. Suppose we extend SQL's syntax for its `WHERE` clause to allow boolean primitives in the search condition. For example, the clause `WHERE ISNEWHIRE AND SALARY > 1000` is allowed, where `ISNEWHIRE` is a column of type Boolean indicating whether an employee is a newhire or not. Notice that standard SQL considers this clause illegal, and it needs to be rewritten as `WHERE ISNEWHIRE = 'true' OR SALARY > 1000`. With this rewriting, the correct use of the basic comparisons in the search condition guarantees the correct typing of the whole clause. To see the problem with our extension, notice that the three edges in bold results in an error with context-free language reachability (cf. Figure 9).

For more expressive languages, more precise information about the underlying syntactic structure of the object-programs is required. By unifying the two steps of CFL-reachability, we retain this information. In more detail, we apply CFL-reachability to reconstruct the complete set of possible derivations of the object-programs as a tree forest. Then, it is processed and checked top-down for errors by modifying standard analysis techniques for a single derivation. Each

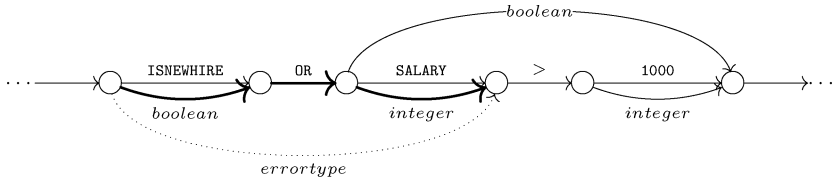


Fig. 9. Imprecision in type checking for more expressive languages.

expression in the object-programs carries with it a set of possible types instead of a single type. In some sense, this is related to languages with union types. This does potentially increase the complexity of the analysis, and in return gains more precision. Further theoretical and experimental study is required to understand this trade-off.

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 [Guyot 1998]. 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 of Christensen et al. [2003]. for computing the FSA, which in turn uses the Soot framework [Vallee-Rai et al. 1999] 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 III lists the test programs that we use. 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). Note that the test programs are sorted by automaton size, since it is a good measure of the complexity of the object-programs for our analysis. Table IV summarizes our experimental results. For each test program, we list the analysis time (split into automaton generation and semantic analysis), numbers of various warnings and errors found (cf. Table V). The semantic analysis here includes both the time to extract

Table III. Test Programs

Test Programs (S) Student (W) Web Download (I) Industrial	Size			
	Java Program (Lines)	Hotspots	Schema (Columns)	Automaton Edges/Nodes
Smi (S)	1559	1	19	35 / 27
CFWorkshop (S)	36	5	13	47 / 52
OrderUp (S)	6139	7	48	59 / 66
PizzaParlor (S)	4915	10	36	74 / 80
TicTacToe (S)	2888	2	26	134 / 121
Checkers (S)	6620	4	36	150 / 125
WebBureau (W)	50	10	21	152 / 162
JuegoParadis (S)	6139	13	29	248 / 226
PizzaToGo (S)	8491	35	47	295 / 330
Snowman (S)	6743	25	61	331 / 349
Reservations (S)	2385	22	54	368 / 383
OfficeTalk (S)	5812	29	14	524 / 525
Sanford (S)	8512	54	84	614 / 585
PurchaseOrders (I)	642	51	82	1324 / 1373

Table IV. Experimental Results

Test Programs	Analysis Time (sec)		Errors Found			
	String Analysis	Semantic Analysis	Warnings	Total Errors	Confirmed Errors	False Errors
Smi (S)	1.5	7.1	0	0	0	0
CFWorkshop (S)	0.6	1.3	0	0	0	0
OrderUp (S)	2.53	0.75	0	0	0	0
PizzaParlor (S)	3.13	0.97	0	0	0	0
TicTacToe (S)	6.4	144.8	3	1	1	0
Checkers (S)	11.8	97.8	0	15	15	0
WebBureau (W)	0.5	2.5	0	1	1	0
JuegoParadis (S)	27.0	45.0	0	9	0	9
PizzaToGo (S)	10.57	9.93	1	6	6	0
Snowman (S)	2.56	46.45	5	3	3	0
Reservations (S)	1.7	29.1	0	0	0	0
OfficeTalk (S)	7.0	120.8	0	2	2	0
Sanford (S)	11.23	16.68	0	28	28	0
PurchaseOrders (I)	1.3	173.3	41	10	9	1

the automaton from the string analyzer and the time to analyze the automata. All experiments were done on a machine with a 2 GHz Intel Xeon processor and 1 GB RAM, running Linux kernel 2.4.20. The results indicate that our analysis is rather precise, that is, 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.

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

—**Type Mismatch:**

(I) *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

Table V. Error Breakdown of Confirmed Errors.

	Type Mismatch			Semantic	
Test Programs	(I) Concatenation Of Wrong Types	(II) Unquoted String	(III) Quoted Number	(I) Ambiguous Column	(II) Column Not Found
Smi (S)	0	0	0	0	0
CFWorkshop (S)	0	0	0	0	0
OrderUp (S)	0	0	0	0	0
PizzaParlor (S)	0	0	0	0	0
TicTacToe (S)	0	0	1	0	0
Checkers (S)	0	0	14	0	1
WebBureau (W)	0	0	0	1	0
JuegoParadis (S)	0	0	0	0	0
PizzaToGo (S)	0	0	6	0	0
Snowman (S)	0	0	3	0	0
Reservations (S)	0	0	0	0	0
OfficeTalk (S)	0	0	2	0	0
Sanford (S)	0	0	28	0	0
PurchaseOrders (I)	5	1	0	0	3

has been used to find all instances of the error in the "PurchaseOrders" program.

(II) *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.

(III) *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`.

—**Semantic Errors:**

(I) *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.

(II) *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 non-existent 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. We did not implement this modification because we used the string analyzer as a black-box in our implementation.

—**Warning.** We found one type of warning in our test programs. It is the same as the one illustrated in our running example in Section 1: to compare a numerical column with a *possibly* non-numerical value at runtime.

The original version of our implementation is quite efficient even though we did not tune it for performance—it was able to analyze each of our test programs within a matter of minutes. As previously stated, our tool consists of three components: Soot to parse class files, the BRICS Java String Analyzer to analyze string variables and construct FSAs, and our own additions to type check the queries represented by the FSAs. Generating the automata proved to be the performance bottleneck in our tool, but we expect that this can be made much more efficient. The ratios of nodes to edges averaged approximately 1 : 1, which indicates that these automata are usually linear or nearly linear. We expect that heuristics designed to search for queries that are statically determined and possibly include user inputs can produce linear automata much more quickly and improve the tool's performance.

We did not attempt to re-implement or optimize either of the first two components, but we did study the effects of optimizing our analysis. Our analysis consists of three phases: lexing the automaton, recovering the type environment, and type checking. As explained in Section 3.2, we reconstruct the type environment by using CFL-reachability to parse the automaton. However, reconstructing the type environment only requires identifying constants and correctly matching columns with tables; the schema then yields the columns' types immediately, and no external information is needed to type constants. A more efficient algorithm can accomplish the same end. We re-implemented the environment reconstruction using a flow-based algorithm. Essentially, at each node after a `FROM` token or before a `WHERE` token, we find all tables which could be reached from that node or to that node, respectively. These are the nodes that terminate the table list at either end. The lists of possible tables are then propagated in either direction to match columns with tables. It is also necessary to find table and column aliases (e.g., `items AS i`) and propagate this information appropriately. The two implementations are interchangeable with the following

Table VI. Comparison of Performance when using CFL-Reachability vs. a Flow-Based Algorithm for Environment Recovery

		Environment Recovery			Type Checking		
Test Programs	Lexing (ms)	CFL (ms)	Flow (ms)	Percent Speedup	CFL (ms)	Flow (ms)	Percent Speedup
Smi (S)	7	54	18	200%	39	38	3%
CFWorkshop (S)	6	53	20	170%	34	37	−6%
OrderUp (S)	9	58	12	394%	53	40	31%
PizzaParlor (S)	10	84	14	500%	34	62	−45%
TicTacToe (S)	11	136	33	312%	723	251	188%
Checkers (S)	11	139	107	29%	110	97	14%
WebBureau (W)	25	131	24	447%	85	52	65%
JuegoParadis (S)	23	165	28	489%	74	72	3%
PizzaToGo (S)	25	149	21	597%	94	102	−8%
SnowMan (S)	23	218	24	795%	110	95	17%
Reservations (S)	26	240	38	525%	158	73	117%
OfficeTalk (S)	25	248	39	541%	460	102	351%
Sanford (S)	27	275	42	556%	42	109	30%
PurchaseOrders (I)	38	1089	76	1326%	1291	414	212%

exceptions:

—The flow-based implementation will not discover ambiguous column references or cardinality errors in `IN` clauses. In our tests, we found one of the first kind of error and zero of the second.

—If the automaton generates queries whose syntax cannot be parsed by the grammar that the tool implements (e.g., a query with an `ORDER BY` clause if `ORDER BY` is not in the tool's grammar), the CFL-reachability implementation will ignore those queries. The flow-based approach will attempt to analyze them, and may find errors that the other technique ignored.

—The type checking phase requires a larger type grammar if the flow-based implementation is used (see below).

Table VI summarizes the performance of our tool in the lexing, environment recovery, and type checking phases. The performance using the original (cubic time) CFL-reachability in both the environment recovery and type checking phases is shown under the column labeled "CFL," and the performance using a (quadratic time) flow-based algorithm is shown under the column labeled "Flow." In the largest example, using the flow-based algorithm improved the time for environment recovery by more than an order of magnitude. The flow-based algorithm clearly performs better than the CFL-reachability algorithm. On the other hand, the implementation based on CFL-reachability is easier to extend and maintain. The supported SQL grammar is implicit and diffuse in the flow-based algorithm, so modifying the grammar is more difficult.

The type checking phase was not modified, but in most cases it performed better when CFL-reachability was not used to recover the type environment. This is because CFL-reachability adds many new edges to the graph in the environment recovery phase, resulting in a larger input graph to the type checking phase. In some cases, however, the type checking phase was slower. This is

because the type grammar needs more rules when the edges that would be added during the environment reconstruction are not present. For example, the type grammar has a rule like:

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

rather than having separate rules for "+" and "−." If CFL-reachability is used, it adds an *add_op* edge during environment recovery. If it is not, the rule to add this edge must be included in the type grammar.

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 [Su et al. 2000; Heintze and Tardieu 2001]. Further experiments are needed to verify our claim. In many cases, however, large, Java-based web applications use an object-relational mapping API, so our analysis would not apply to them directly. We have shown that our tool can detect errors in nontrivial programs. Currently the tool is being evaluated by a few friendly users in both commercial and research-oriented settings.

5. RELATED WORK

In this section, we survey closely related work. Perhaps the most closely related is the string analysis of Christensen et al. [2003] 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. The string analysis has some limitations regarding precision, but related work [Choi et al. 2006] shows that it can smoothly be extended with standard techniques for better heap modeling and context sensitivity. 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. Sandholm and Schwartzbach [2000] proposed a typed, higher-order template language that provides safety of the dynamically generated web documents within the `<bigwig>` project [Brabrand et al. 2002], an extension to Java for high-level web service development. Their type system is based on standard data-flow analysis techniques [Kildall 1973; Kam and Ullman 1976]. Another work along the same lines is the work by Braband et al. [2001] 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 Kapfhammer and Soffa [2003], 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 [Taha and Sheard 1997]. 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 and Crew [1993] 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, 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. [1986] provide these guarantees. More recent work seeks to extend syntactic and scoping guarantees, with *semantic guarantees* for the generated code. The work of Taha and Sheard [1997] 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 Weise and Crew [1993], nor work in a uniform functional setting, like Taha and Sheard [1997], 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 article can be extended to apply in other settings as well.

Other techniques have been proposed for static checking of generated database queries, some since the first presentation of the ideas in this article. For completely static queries, SQLJ [1997] provides a higher-level API for using SQL in Java than JDBC. The SQLJ translator performs type checking and schema checking of SQL statements at program development time rather than at runtime. Safe Queries Objects [Cook and Rai 2005] provides the benefits of SQLJ for dynamically generated SQL queries written in Java. Safe query objects use object relational mapping and reflective metaprogramming to translate query classes into traditional database queries. In contrast to the above, our technique does not introduce a new API.

Other existing APIs provide object-relational mappings to databases. Hibernate [JBoss 2006] is open source and is a powerful, high-performance object-relational persistence and query service. Oracle's TopLink [Oracle 2006] provides roughly the same features plus caching, support for transactions, and performance tuning options. ADO.Net [Microsoft 2004] is another object-relation mapper; it was designed specifically for the web with scalability, statelessness, and XML in mind. All of these allow for queries to be passed directly as strings (as JDBC does), so the technique presented in this article would apply directly to such uses. Other uses of these API's provide some type checking but do not integrate into the static type system of the source language. For example, consider the JDO [Sun 2003] code in Figure 10. The types are declared in a

```
void PrintInfo(String prefix, int base)
{
    Command q = new Query(Employee.class);
    String paramDecl = "String prefix, int base";
    String filter = "emp.name.startsWith(prefix)"
        + " && emp.salary >= base";
    q.declareParameters( paramDecl );
    q.setFilter( filter );
    for ( Employee emp :  q.execute(prefix, base) )
        print( emp.name );
}
```

Fig. 10. Example JDO code.

string, which is not interpreted by source language's type checker. Our approach is not designed to discover potential type errors in this API. LINQ [Microsoft 2005], in contrast, fully integrates the types involved in database access into the source language. Cook and Ibrahim [2005] survey more broadly the problems of interfacing databases with programming languages (including type checking) and discuss the merits of each of these APIs with respect to several criteria Cook and Ibrahim [2005]. Each of these techniques does alleviate the problem of type errors in generated queries to some degree, but the merit of our approach is that it can be applied to many existing applications which have not been written with these APIs.

SQL command injection is a security threat, which is more dangerous than type errors in dynamically generated queries. Two papers have leveraged the techniques presented here to address this problem. The first is purely static [Wassermann and Su 2004]. It works by doing a more sophisticated analysis on the generated automata to discover access control errors and potential tautologies in the `WHERE` clauses. The second uses the generated automata as an inferred specification for the queries [Halfond and Orso 2005]. It then monitors the queries at runtime to ensure they conform to this specification. For a more extensive survey of techniques for alleviating SQL command injection, see Wassermann and Su [2004].

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. The technique is designed for programs that construct string queries directly; it is not intended to address queries produced by other APIs (e.g., object-relational mappers) or queries constructed by the database (e.g., with stored procedures). 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. First, 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. Second, the class of embedded SQL command injection problems [Viega and McGraw 2001] 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 article. 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 and TOSEM for their valuable comments on earlier drafts of this article.

REFERENCES

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

AMADIO, R., AND CARDELLI, L. 1991. Subtyping recursive types. In *Proceedings of the 18th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, FL). ACM, New York, 104–118.

ANDERSEN, L. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen. DIKU report 94/19.

BRABAND, C., MØLLER, A., AND SCHWARTZBACH, M. 2001. Static validation of dynamically generated HTML. In *ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, UT). ACM, New York, 38–45.

BRABRAND, C., MØLLER, A., AND SCHWARTZBACH, M. 2002. The <bigwig> project. *ACM Trans. Internet Tech. 2,* 2, 79–114.

CHOI, T.-H., LEE, O., KIM, H., AND DOH, K.-G. 2006. A practical string analyzer by the widening approach. In *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS 2006)* (Sydney, Australia). Springer-Verlag, New York, 374–388.

CHRISTENSEN, A., MØLLER, A., AND SCHWARTZBACH, M. 2003. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium* (San Diego, CA). Springer-Verlag, New York, 1–18.

COOK, W. R., AND IBRAHIM, A. H. 2005. Programming languages & databases: What's the problem? url: `http://www.cs.utexas.edu/~wcook/Drafts/2005/PLDBProblem.pdf`.

COOK, W. R., AND RAI, S. 2005. Safe query objects: Statically typed objects as remotely executable queries. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)* (St. Louis, MO). ACM, New York.

GUYOT, J. 1998. BNF index of SQL for Oracle 7. Available at `http://cui.unige.ch/db-research/Enseignement/analyseinfo/SQL7/`.

HALFOND, W. G., AND ORSO, A. 2005. Combining static analysis and runtime monitoring to counter SQL-injection attacks. In *Proceedings of the 3rd International ICSE Workshop on Dynamic Analysis (WODA 2005)* (St. Louis, MO). ACM, New York. url: `http://www.csd.uwo.ca/woda2005/proceedings.html`.

HEINTZE, N., AND TARDIEU, O. 2001. 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* (Snowbird, UT). ACM, New York, 254–263.

HOPCROFT, J., AND ULLMAN, J. 1979. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, Reading, MA.

JBOSS. 2006. Hibernate. url: `http://www.hibernate.org/`.

KAM, J., AND ULLMAN, J. 1976. Global data flow analysis and iterative algorithms. *J. ACM 23,* 1 (Jan.), 158–171.

KAPFHAMMER, G. M., AND SOFFA, M. L. 2003. 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* (Helsinki, Finland). ACM, New York, 98–107.

KILDALL, G. 1973. A unified approach to global program optimization. In *Conference Record of the ACM SIGACT and SIGPLAN Symposium on Principles of Programming Languages* (Boston, MA). ACM, New York, 194–206.

KOHLBECKER, E., FRIEDMAN, D., FELLEISEN, M., AND DUBA, B. 1986. Hygenic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, MA). ACM, New York, 151–159.

MELSKI, D., AND REPS, T. 1997. 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* (Amsterdam, The Netherlands). ACM, New York, 74–89.

MICROSOFT. 2004. ADO.NET. url: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/adonetanchor.asp`.

MICROSOFT. 2005. LINQ Project. url: `http://msdn.microsoft.com/netframework/future/linq/`.

ORACLE. 2006. Oracle TopLink. url: `http://www.oracle.com/technology/products/ias/toplink/index.html`.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, CA). ACM, New York, 49–61.

SANDHOLM, A., AND SCHWARTZBACH, M. 2000. A type system for dynamic web documents. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA). ACM, New York, 290–301.

SHIVERS, O. 1988. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, GA). ACM, New York, 164–174.

SQLJ. 1997. See `http://www.sqlj.org`.

SU, Z., FÄHNDRICH, M., AND AIKEN, A. 2000. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA). ACM, New York, 81–95.

SUN. 2003. Java Data Objects (JDO). url: http://java.sun.com/products/jdo/.

TAHA, W., AND SHEARD, T. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulations* (Amsterdam, The Netherlands). ACM, New York, 203–217.

VALLEE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. 1999. Soot–a Java optimization framework. In *Proceedings of the IBM Centre for Advanced Studies Conference, CASCON'99* (Mississauga, Ont., Canada). IBM, New York.

VIEGA, J., AND MCGRAW, G. 2001. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison Wesley, Boston, MA.

WASSERMANN, G., AND SU, Z. 2004. An analysis framework for security in Web applications. In *Proceedings of the 3rd FSE Workshop on the Specification and Verification of Component Based Systems (SAVCBS 2004)* (Newport Beach, CA). ACM, New York, 70–78.

WEISE, D., AND CREW, R. 1993. Programmable syntax macros. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Albuquerque, NM). ACM, New York, 156–165.

YELLIN, D. 1993. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Inf. 30,* 4 (July), 369–384.

Received August 2005; revised July 2006; accepted November 2006