Deep Typechecking and Refactoring *

Zachary Tatlock
UC San Diego
ztatlock@cs.ucsd.edu

Chris Tucker
UC San Diego
cjtucker@cs.ucsd.edu

David Shuffelton
UC San Diego
dshuffel@cs.ucsd.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

Abstract

Large software systems are typically composed of multiple layers, written in different languages and loosely coupled using a string-based interface. For example, in modern web-applications, a server written in Java communicates with a database back-end by passing in query strings. This widely prevalent approach is unsafe as the analyses developed for the individual layers are oblivious to the semantics of the dynamically constructed strings, making it impossible to statically reason about the correctness of the interaction. Further, even simple refactoring in such systems is daunting and error prone as the changes must also be applied to isolated string fragments scattered across the code base.

We present techniques for deep typechecking and refactoring for systems that combine Java code with a database back-end using the Java Persistence API [10]. Deep typechecking ensures that the queries that are constructed dynamically are type safe and that the values returned from the queries are used safely by the program. Deep refactoring builds upon typechecking to allow programmers to safely and automatically propagate code refactorings through the query string fragments.

Our algorithms are implemented in a tool called QUAIL. We present experiments evaluating the effectiveness of QUAIL on several benchmarks ranging from 3,369 to 82,907 lines of code. We show that QUAIL is able to verify that 84% of query strings in our benchmarks are type safe. Finally, we show that QUAIL reduces the number of places in the code that a programmer must look at in order to perform a refactoring by several orders of magnitude.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Reliability, Verification

Keywords JPA Query Analysis, Cross Language Typechecking, Cross Language Refactoring

1. Introduction

Large software systems are built in multiple layers, employing various systems, languages, and run-times. For example, companies like Amazon, Google, and Yahoo all produce software that has three basic tiers: a browser front-end using HTML and Javascript, a middle-tier running Java, .NET, or a similar stack of server technology, and a storage tier using a relational database or other long-lived storage platform. A successful integration of these layers must enjoy three properties: it must be *efficient*, in that it must be able to exploit the beneficial properties of the individual layers; it must be *usable*, in that it must allow engineers to easily utilize the full range of functionality available in the individual layers; and it must be *safe*, in that it must prevent errors due to interactions that cut across the different layers.

Unfortunately these three properties have hitherto resisted reconciliation. For example, consider the problem of interfacing between an object-oriented programming language and a relational database. A number of approaches exist for translating data to and from objects and relations, each of which attempts to deal with the so-called "impedance mismatch" [13] that exists between relational data and object-oriented data.

Orthogonal persistence solutions [1, 12] map the entire database into a collection of persistent objects that are directly manipulated and navigated in the programming language. This approach is safe, as the programming language's

* Supported in part by the NSF grants CCF-0427202, CNS-0541606, and CCF-0546170.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

type system is in full control, but is often hard to use and sacrifices opportunities for optimization of data access. In particular, the programming language and database are tightly coupled, resulting in a system whose data design is dependent on programming language decisions, which in turn leads to a less efficient and harder to use data store.

Language-based solutions [18, 15] extend programming languages like Java [23] or C♯ [2] with syntax inspired by functional programming constructs [24, 5] that can be directly compiled into database queries. Not only are these approaches safe, but they can also be efficient. In particular, the programming language and database can be more loosely coupled than in an orthogonal persistence solution, resulting in improvement efficiency of data storage and retrieval. However, the usability of these techniques remains to be seen. One barrier to adoption is that these techniques hide the structure of the generated queries, which can lead to unexpected performance bugs.

Finally, call-level interfaces [20] such as Toplink [6], EJB [14], JDO [17], and Hibernate [9] — recently consolidated under the Java Persistence API (JPA) [10, 11] standard for Java — allow developers to describe mappings of relational constructs onto Java types, which can then be queried using SQL-like query strings. This approach is both flexible and efficient because the query strings are expressed in a language specifically designed for data access, but are less safe as the host language is unaware of the content of the query strings, and is unable to apply its type constraints to those queries. This approach has the loosest coupling between programming language and database, and makes it easy for developers to efficiently query for and use data from the database without having any direct knowledge of the relational structures used to hold that data, but at the cost of a loss of safety.

In general the approach of loosely coupling different systems using a string-based interface with a domain-specific interfacing language is flexible, can leverage the properties of each layer effectively, does not force the developer to learn another language or paradigm, and hence is widely adopted by large-scale system developers.

Unfortunately, this approach is unsafe in that it greatly complicates two standard software engineering tasks: typechecking and refactoring. Large systems that are made of loosely coupled heterogeneous components have little support for performing typechecking at the boundaries of different languages. Furthermore, performing even simple refactoring in such systems is daunting and error prone, because the required changes cut across different layers, causing subtle bugs to slip between the cracks.

As an illustrative example, consider the Java code snippet from Figure 1. In this snippet, `Q0` is an abbreviation for the query string shown at the top of the figure. On line 2 the code creates a JPA query that returns the set of books owned by a given person `?1`. The person `?1` is a parameter to the query, and this parameter is set (on line 3) using a call to `setParameter`. Once the parameter is set, the query is executed (on line 4), and the result of the query, a list of `Book` objects, is stored in the `result` variable. The code then proceeds to print the name of each book in the `result` list (on lines 5-8). Note how in this code the `Book` and `Person` classes crosscuts both Java and the query language: they are meaningful in both contexts.

```
Q0:"SELECT b FROM Book b WHERE b.buyer = ?1"

1: Person user = getCurrentUser();
2: Query q = createQuery(Q0);
3: q.setParameter(1, user);
4: List<Object> result = q.getResultList();
5: for (Object b : result) {
6:   Book book = (Book) b;
7:   System.out.println(book.name);
8: }
```

Figure 1. Sample JPA Code

This code snippet illustrates two of the main difficulties in writing and maintaining JPA code:

Type Errors: As objects are passed to and from the database back-end through a single interface (namely `setParameter` and `getResultList`), these interfaces use `Object` for the type of parameters and results, which leads to the compiler missing type errors. For example, if a `Car` object were passed in to `setParameter` on line 3, the compiler would not be able to flag this as a type error. Furthermore, because `getResultList()` returns a list of `Objects`, the variable on line 4 is declared as `List<Object>`, not `List<Book>`, and as a result there is an additional cast on line 6, which at run-time may fail if the programmer uses the wrong type in the type cast.

Refactoring Errors: As names of classes appear in query strings, for example in `Q0`, changing the name of a class involves renaming all the queries that mention the class. Unfortunately, there is little support for helping the programmer make sure that all such strings have been renamed. The class-rename refactoring in Eclipse does not refactor such strings, and furthermore forgetting to rename a string does not lead to a type error, unlike renaming a class directly in Java, where forgetting to rename a mention of the renamed class leads to a type error.

Our broad research agenda is to address these problems, and thus reconcile safety, usability and efficiency, by developing techniques and tools for *deep typechecking* and *deep refactoring* – type-checking and refactoring that cut across different languages and systems. In this paper we present techniques for deep typechecking and refactoring for systems that combine Java code with a database back-end using JPA.

For deep typechecking, the main challenges lie in the fact that the problem cuts across Java and the JPA query languages in non-trivial ways. First, the Java code *manipulates* strings that *represent* JPA queries, making it difficult to identify the queries that are executed. Second, the Java code *sets the parameters* to the JPA queries, and *uses the results* that they produce, which requires tracking where these results flow to determine if they are used correctly.

We address these challenges by using different techniques for analyzing Java and JPA queries. We use a dataflow analysis on the Java side to compute the set of query strings that may be executed at a particular call site to `getSingleResult`, and to propagate the result type of executed queries through the code to make sure that downcasts will not fail at run-time. On the JPA side, we use a type system, augmented with some simple constructs to represent sets of queries, to determine if a query (or set of queries) typechecks. Working together, these two techniques allow us to identify (1) what query strings are executed by the Java code, (2) whether these query strings typecheck given the parameters that the Java code has set on them, and (3) whether the results of these queries are used correctly in the Java code.

For deep refactoring, the main challenge lies in updating raw strings that represent JPA queries or fragments of queries which are concatenated together by the Java code. Fragments are particularly difficult, not only for an automated tool, but also for humans, because there is no context available for understanding the query – it is difficult to tell if a given substring should be refactored without knowing the context in which the substring is typed.

We address this challenge by building upon deep typechecking as follows. First our algorithm tags each query fragment with an identifier that uniquely determines the Java string literal that the fragment originated from. The string dataflow analysis used for deep typechecking is used to compute the set of complete tagged query strings that reach an execution site. Next, our algorithm performs refactorings in the context of an entire query, using the deep typechecker as a subroutine to determine the types of identifiers. Finally, our algorithm propagates the changes back to the Java code using the tags.

In summary, this paper takes a step towards enabling developers to *safely* combine different layers using efficient and flexible call-level interfaces. In particular, we make the following concrete contributions:

- **Deep Typechecking.** We present techniques for checking that the parameters set by the Java code on JPA queries are of the correct type, that all parameters of a query are set before it is executed, and that the returned values of JPA queries are used correctly in the Java code (Section 3).

```
Q1 : "SELECT w FROM WeblogTemplate w
      WHERE w.website = ?1 AND w.link = ?2"

WeblogTemplate getPageByLink(Weblog site, String link){
1:  Query q = createQuery(Q1);
2:  q.setParameter(1, new Book());                //ERROR
3:  q.setParameter(1, site);                      //OK
4:  q.getSingleResult();                          //ERROR
5:  q.setParameter(2, link);                      //OK
6:  Book b = (Book) q.getSingleResult();          //ERROR
7:  return (WeblogTemplate) q.getSingleResult();//OK
}
```

Figure 2. Query Type Checking

- **Deep Refactoring.** We present techniques for performing class-rename and field-rename refactorings on classes and fields that may appear in query strings (Section 4). Our refactoring technique renames not only direct references from the Java code, but also references that appear in query strings that the Java code manipulates.
- **Experimental validation.** We implemented the above techniques in a tool called QUAIL, and present the results of running QUAIL on a variety of benchmarks (Section 5), the largest of which is `Roller`, a web blogging software system comprising 82,907 lines of code. We show that QUAIL is able to verify that 84% of query execution sites in the Java code use types correctly (the remaining cases are false-positives where we incorrectly report safe query executions as potential errors), and we show that our tool reduces the number of places in the code that a programmer must look at to perform refactorings by several orders of magnitude.

2. Overview

We begin with an overview of Deep Typechecking and Refactoring using simplified versions of examples taken from the `Roller` code base.

Deep Typechecking

Consider the Java method shown in Figure 2. For clarity, we write `Q1` as an abbreviation for the query string shown at the top of the Figure. For the moment, ignore the shaded lines 2,4,6. The string `Q1` corresponds to a `SELECT` query, which is used to find all `WeblogTemplate` objects `w` whose `website` and `link` fields respectively match the arguments `site` and `link` passed into the method. To this end, in line 1 the string `Q1` is used to create a query object `q`, and `setParameter` calls are used to bind the query parameters to the corresponding Java values on lines 3,5 respectively. Next, the query is executed by calling the `getSingleResult` method on line 7. The output of the method, is downcast to `WeblogTemplate` and returned.

Motivation. The goal of deep typechecking is to statically prove the absence of a variety of run-time errors (exceptions) that can arise in buggy programs, but which, due to the polylingual nature of the interaction, slip through the cracks of Java's type system.

- **Query Correctness:** First, suppose that the method contained the line 2, wherein the first query parameter is bound to a `Book` object. The Java type system would not flag any errors as the `setParameter` method takes any `Object` as input. However, the subsequent query execution will fail as the `website` field of a `WeblogTemplate` object is of type `Weblog` which cannot be compared against a `Book` object. Of course, errors can be caused independent of the ways that the parameters are set *e.g.* if `WeblogTemplate` objects do not have a `link` field. Our first goal is to ensure that in each query object, the parameters are set *safely*, *i.e.,* so that the subseqent queries execute without errors.
- **Query Completeness:** Second, suppose that the method contained the line 4, where the query is executed *before* the second parameter is bound. At run time, this line will cause the JPA implementation to throw an exception as the second query parameter is not bound. Our second goal is to ensure that in each query object, *all* the parameters have been set at the point where the query is executed.
- **Output Correctness:** Finally, suppose that the method contained the line 6 where the query is executed and its result downcast to type `Book`. As the query execution method `getSingleResult` returns an `Object` result, the line will typecheck but will cause a failed downcast exception to be thrown at run-time as the actual object returned is of the type `WeblogTemplate`. Our third goal is to ensure that the objects, or as we shall see, lists of objects returned by queries are safely used by the rest of the Java code, *i.e.,* do not cause failed downcast exceptions at runtime.

Algorithm. The key to connecting the Java code with the database back-end is the notion of a *bound query* which is a pair of a query object and a (partial) mapping from the parameters of the query to the *types* of the objects the parameters are bound to. For example,

$$\left(\begin{array}{l}\texttt{select w from Template w where w.website} = ?\texttt{1},\\ ?\texttt{1:Weblog}\end{array}\right)$$

represents a bound select query where the first parameter is bound to an object of type `Weblog`.

QUAIL uses bound queries to perform deep type checking via a three-step algorithm. In the first step, *Bound Query Analysis*, QUAIL performs a dataflow analysis to determine the set of bound queries that flow to each program point where a query is executed *e.g.* lines 6,7 from Figure 2. In the second step, *Type Analysis*, QUAIL checks that each of the bound queries that reach an execution point represent a well typed JPA query – namely that all the parameters are set to values of appropriate types. Further, in this step, QUAIL analyzes the structure of the query to infer the type of the query result. In the third step, *Result Analysis*, QUAIL propagates the type inferred for the query result to the points inside program where the result is downcast from `Object`, to verify that the downcast is safe.

QUAIL verifies the code in Figure 2 (ignoring the shaded lines), as exactly one bound query flows to the execution point on Line 7, and in this bound query, the two parameters ?1 and ?2 are set to objects of correct types. Further, the downcast succeeds as it is to the same type as that of `w` defined in the query.

If line 2 is used instead of line 3, QUAIL flags a warning as the bound query reaching line 7 fails to type check, as the comparison `w.website` $=$?1 fails to typecheck in a type environment where `w` has type `WeblogTemplate` (whose field `Website` has type `Weblog`) and ?1 has type `Book`. If line 4 is used, QUAIL flags a warning as the type of the parameter ?2 is undefined in the bound query flowing to line 4, causing the comparison `w.link` $=$?2 to not typecheck. If line 6 is used, QUAIL flags a warning as the inferred type of the output of the query reaching the execution point is not a superclass of `Book`.

```
Q2:"SELECT c.weblogEntry.website.name
    FROM WeblogEntryComment c
    WHERE c.weblogEntry.pubTime < ?1
      AND c.weblogEntry.pubTime > ?2"
Q3:"SELECT c.weblogEntry.website.name
    FROM WeblogEntryComment c
    WHERE c.weblogEntry.pubTime < ?1"

List getMostCommentedWebsites(Date start, Date end){
  Query q = null;
  if (end == null) end = new Date();
  if (start != null) {
    q = createQuery(Q2);
    q.setParameter(1, new Timestamp(end.getTime()));
    q.setParameter(2, new Timestamp(start.getTime()));
  } else {
    q = createQuery(Q3);
    q.setParameter(1, new Timestamp(end.getTime()));
  }
  List qs = q.getResultList();
  List results = new ArrayList();
  for (Iterator i = qs.iterator(); i.hasNext();){
    results.add((String) i.next());
  }
  return results;
}
```

Figure 3. Multiple Query Flow

Multiple Queries. The JPA makes it easy to use run-time information to build different kinds of query objects. Consider the method shown in Figure 3 which takes a `start` and `end` date and returns a list of websites that were published between those dates. The `start` parameter is optional – if it is `null` then the method should return all the sites published before the `end` date.

As before, we use `Q2` and `Q3` as abbreviations for the two query strings shown at the top of the Figure. The method checks if the `start` parameter is non-null. In this case, the method uses `Q2` to create a query with two parameters – a start and end, and sets the two parameters. If the case that `start` is `null`, the method uses `Q3` to create a query with only one parameter and sets the parameter appropriately. Finally, the method executes the query by calling `getResultList` and iterates over the list of objects returned, downcasting each object in the list to a `String` before adding it to the list of `results`.

QUAIL checks this method as follows. First, the bound query analysis deduces that *two* bound queries flow into `q` at the program point at line 11. The first bound query corresponds to `Q2` and has the parameters ?1 and ?2 bound to the type `Timestamp`. The second query corresponds to `Q3` and has the parameter ?1 bound to `Timestamp`. Second, each of these bound queries typechecks as in the type environment where `c` is of type `WeblogEntryComment`, the term `c.weblogEntry.pubTime` has type `Timestamp` and so the comparison(s) typecheck. Third, both queries return as output the field `c.weblogEntry.website.name` which has type `String`. By propagating this output type through the assignments on Lines 11,13,14, we infer that `i.next()` is of type `String` and so the downcast succeeds.

Deep Refactoring

Though query strings make for a simple and expressive interface between Java and the database back-end, they greatly complicate the task of modifying the code or database schemas as now the programmer must painstakingly go through each of the strings to make sure they are appropriately modified. For example, renaming (or deleting) a field of a persistent class would require sifting through each query string scattered across multiple files and appropriately renaming (or deleting) references to the modified field. Unfortunately, the Java type checker is of little help as the references are embedded within strings.

Motivation. The goal of deep refactoring is to simplify this task by using the machinery developed for deep typechecking to *automatically* and correctly update the query strings to reflect the changes made to the classes or the database. In particular, we consider two refactoring tasks – *Class Renaming* and *Field Renaming*. It is easy to extend our techniques to other tasks like automatically determining which query strings are *impacted* by the deletion of fields in persistent classes.

Recall the example from Figure 3. Suppose that the programmer wishes to change the name of the field `name` of the class `Weblog` to `blogName`. Unfortunately, one cannot simply perform a search-and-replace of the string `name` with

```
Q4:"SELECT w.name FROM WeblogAggregate w
    WHERE w.name = ?1"

boolean getTagComboExists(String tag, Weblog weblog){
    StringBuffer s = new StringBuffer();
    s.append(Q4);
    params.add(tag);
    if(weblog != null) {
      s.append(" AND w.weblog = ?2");
      params.add(weblog);
    } else {
      s.append(" AND w.weblog IS NULL");
    }
    Query q = createQuery(s.toString());
    List results = q.getResultList ();
    ...
}
```

Figure 4. Query String Construction

`blogName` as several other classes can also have fields with the same name.

To automatically refactor the query strings, QUAIL performs a modified version of the bound query analysis, where each query string is tagged with a unique identifier that indicates the location of the string in the program source. For the program in Figure 3, the set of tagged bound queries reaching the execution point at Line 11 is:

$$\{(\langle \mathtt{Q2} \rangle_4, \ldots), (\langle \mathtt{Q3} \rangle_8, \ldots)\}$$

Next, QUAIL refactors each tagged bound query by refactoring each term appearing in the query. For example, as `c` has the type `WeblogEntryComment`, the term `c.weblogEntry.website` has the type `Weblog` and hence, the term `c.weblogEntry.website.name` is changed to `c.weblogEntry.website.blogName`. A similar change is made to the second tagged query. Finally the tags are used to substitute the new query strings in place of the old ones in the program source.

Dynamic Query Construction. So far we have looked at queries built from contiguous strings. A common pattern is to dynamically construct queries by starting with a *base* string to which extra conditions are appended depending on run-time values. Consider the method shown in figure 4 which constructs a query from a base string `Q4`, to which, depending on whether or not the `weblog` variable is null, an extra condition is appended.

Suppose that the programmer wishes to rename the field `weblog` of the class `WeblogAggregate` to `blog`. Again, we cannot simply rename the substring `weblog` without knowing the type of the preceding identifier `w`.

QUAIL does refactoring in the presence of such dynamically created query strings as follows. First, we extend our notion of bounded queries to *extended bounded queries* which comprise a base query concatenated with a regular expression over query fragments that represent the set of possible extra conditions that can be appended to the end of the base query. Each query fragment appearing in the extension is also tagged with a unique identifier indicating the fragment's position in the source. As the variable declarations are in the query prefix string and not scattered over the regular extension, it is easy to extend both the type checking and refactoring algorithms to handle the extensions. Thus, QUAIL starts by using a standard string analysis algorithm [3] to compute the extended bounded queries that reach each query execution point, then it refactors the extended queries at the point, and finally replaces each tagged query fragment in the source with its refactored version.

Thus, for the code shown in Figure 4, QUAIL first determines that the extended query

$$\langle \texttt{Q4} \rangle_2[\langle \texttt{AND}\ \texttt{w.weblog} = ?2 \rangle_5 + \langle \texttt{AND}\ \texttt{w.weblog} = \texttt{NULL} \rangle_8]$$

reaches the execution point at line 10. Notice the common prefix with the declaration for the type of `w`, and the extension that includes the two possible extensions. Using the fact that `w` is of type `WeblogAggregate` obtained from the base query, QUAIL refactors the above to:

$$\langle \texttt{Q4} \rangle_2[\langle \texttt{AND}\ \texttt{w.blog} = ?2 \rangle_5 + \langle \texttt{AND}\ \texttt{w.blog} = \texttt{NULL} \rangle_8]$$

after which, the strings at lines 5,8 are refactored to: `"AND w.blog = ?2"` and `"AND w.blog = NULL"` respectively.

3. Deep Type checking

QUAIL performs deep type checking in three steps: a *Bound Query Analysis* over the Java code determines the set of bound queries that flow to each program point where queries are executed; a *Type Analysis* over the JPA query language then checks that each bound query is type-correct; finally a *Result Analysis* over the Java code checks that objects returned by the queries are used correctly in the Java code.

We start by presenting our type analysis (Section 3.1), which consists of a type system for the JPA query language. We then present the bound query analysis (Section 3.2), which computes the set of bound queries that the type system should be invoked with. Finally, we present the return analysis (Section 3.3), which propagates the return types from the type system through the Java code.

3.1 Type Analysis

Type Environment. A type environment Γ is a partial function from parameter names and identifier names to types. A type in our system can either be a base type or a class. Base types are primitives or classes that are not handled by the Java persistence API. As queries only refer to fields, we do not need to capture methods or inheritance in classes. Thus, we simply abstract a class as a record containing typed fields.

Syntax. The core syntax of JPA queries is shown in Figure 5. There are three kinds of queries: `select`, `update` and `delete`. Each of these queries contains a `where` clause that can be a conjunction, disjunction, or a comparison of two terms using an operator $\bowtie$ which ranges over comparison operators like $=$, $\leq$, $\geq$, *etc.*. Each term is either a query parameter, or an lvalue obtained by following the fields of some identifier declared in the query. We briefly and informally summarize the semantics of queries – the interested reader is referred to [10] for details.

- A `select` $t_1, \ldots, t_n$ `from` Γ `where` e query returns the set of tuples $t_1, \ldots, t_n$ chosen from the set of persistent classes described by the environment Γ, such that the condition e holds for each tuple.
- An `update` $x\!:\!C$ `set` $a_1, \ldots, a_n$ `where` e query updates each instance of C that satisfies the condition e, using the assignments $a_1, \ldots, a_n$.
- A `delete` $x\!:\!C$ `where` e deletes each instance of C that satisfies the condition e.

Tags. Observe that base part of the query, which consists of the `select`, `update` or `delete` clause, is tagged with an identifier i representing the location in the code where the base part originated from. For example, in Figure 3, the query created on line 4 in our syntax will be $\langle \texttt{Q2} \rangle_4$. These tags allow our refactoring algorithm to map changes in the query back to the strings in the Java code.

Extensions. To account for dynamically created queries, our syntax allows for *query extensions*. An extension r represents additions to the base query that have been made programatically using string concatenation. For example, $\langle \wedge e \rangle_i$ represents the fact that e has been added as a conjunct to the `where` clause of the base query, and $\langle \vee e \rangle_i$ represents the fact that e has been added as disjunct. Note that all the extensions apply to the `where` clause of the base query. In some cases, for example the method shown in Figure 4, the code that creates query strings contains control flow. To account for such control flow, query extensions can contain regular expression operators like concatenation and sum. QUAIL uses standard string analysis algorithms [3] to compute, for each site i in the code that calls `CreateQuery`, a query q_i in our syntax that overapproximates the set of queries that could be created at site i. Control flow constructs in the query-building code map directly to regular expression operators in our query syntax: linear code introduces concat operators, branches introduce $+$ operators. Like the base queries, the each query fragment appearing in an extension is tagged with the location in the code from which it originates.

For example, for the code shown in Figure 4, QUAIL would determine that the query created on line 10 is expressed in our syntax as follows:

$$q_{10} = \langle \texttt{Q4} \rangle_2[\langle \wedge\ \texttt{w.weblog} = ?2 \rangle_5 + \langle \wedge\ \texttt{w.weblog} = \texttt{NULL} \rangle_8]$$

That is, the query created at line 10 is an extended query with the base query `Q4` and an extension which is the sum of the

q	$::=$		*Queries:*
	$\mid$	$\langle \texttt{select}\ t_1, \ldots, t_n\ \texttt{from}\ \Gamma\ \texttt{where}\ e \rangle_i\ r$	select
	$\mid$	$\langle \texttt{update}\ x : \tau\ \texttt{set}\ a_1, \ldots, a_n\ \texttt{where}\ e \rangle_i\ r$	update
	$\mid$	$\langle \texttt{delete}\ x : \tau\ \texttt{where}\ e \rangle_i\ r$	delete
e	$::=$		*Expressions:*
	$\mid$	$t \bowtie t$	atom
	$\mid$	$e \wedge e$	conjunction
	$\mid$	$e \vee e$	disjunction
r	$::=$		*Extensions:*
	$\mid$	ϵ	empty
	$\mid$	$\langle \wedge e \rangle_i$	and
	$\mid$	$\langle \vee e \rangle_i$	or
	$\mid$	$r\ r$	concat
	$\mid$	$r + r$	sum
Γ	$::=$		*Type Envs:*
	$\mid$	$x : \tau, \Gamma$	identifier
	$\mid$	$?\texttt{p} : \tau, \Gamma$	parameter
τ	$::=$		*Types:*
	$\mid$	$C = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$	Record
	$\mid$	B	Base
a	$::=$	$lv{:=}t$	*Assignments*
t	$::=$		*Terms:*
	$\mid$	$?\texttt{p}$	parameter
	$\mid$	$\texttt{c}$	constant
	$\mid$	lv	lvalue
lv	$::=$		*Lvalues:*
	$\mid$	x	identifier
	$\mid$	$lv.f$	field-access

Figure 5. Query Syntax

two extensions of the then- and else- branch respectively. The QUAIL typechecker currently does not handle query strings that are created using a loop – such query strings will fail to typecheck. As our experiments will show in Section 5, JPA queries are rarely created in loops.

Type System. The type system for queries is shown in Figure 6. The system has five kinds of judgements:

- **Terms:** $\Gamma \vdash t : \tau$ which state that under the environment Γ the term t has type τ. The type of identifiers and parameters is found from the environment, and type of field expressions $lv.f$ is obtained via the type of the field f of the recursively computed (record) type of lv.
- **Assignments:** $\Gamma \vdash lv{:=}t$ which state that under the environment Γ the assignment $lv{:=}t$ typechecks. An assignment only type checks if the type of the value t being assigned is a subtype of the lvalue lv to which the assignment occurs.
- **Expressions:** $\Gamma \vdash e$ which state that under the environment Γ the expression e typechecks. An expression type checks if each atomic comparison within e is between values of comparable types, a notion made precise by the $\sim$ relation.
- **Extensions:** $\Gamma \vdash r$ which state that under the environment Γ the extension r typechecks. An extension typechecks if each of its constituent expressions typechecks.
- **Queries:** $\Gamma \vdash q$ which state that under the environment Γ the extended query q typechecks. Note that for the extended query to typecheck, its constituent `where` clause and assignments must typecheck. Further, notice that we use the type environment or declaration from the base query to type check the extension.

3.2 Bound Query Analysis

We now describe our *Bound Query Analysis*. The goal of this analysis is to compute the set of *bound queries* that could be executed at each `getSingleResult` call site (where a bound query is a query combined with the parameters that have been set on it)

Domain. As described in Section 3.1, a type environment Γ is a partial function from parameter names and identifier

Terms $\boxed{\Gamma \vdash t : \tau}$

$$\frac{ty(\mathtt{c}) = \tau}{\Gamma \vdash \mathtt{c} : \tau}\ [\text{T-Const}] \qquad \frac{?\mathtt{p} : \tau \in \Gamma}{\Gamma \vdash ?\mathtt{p} : \tau}\ [\text{T-Param}]$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\text{T-Id}] \qquad \frac{\Gamma \vdash lv : \{\ldots, f : \tau, \ldots\}}{\Gamma \vdash lv.f : \tau}\ [\text{T-Lval}]$$

Assignment $\boxed{\Gamma \vdash a}$

$$\frac{\Gamma \vdash lv : \tau \qquad \Gamma \vdash t : \tau' \qquad \tau' <: \tau}{\Gamma \vdash lv := t}\ [\text{Asgn}]$$

Expressions $\boxed{\Gamma \vdash e}$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash t' : \tau' \qquad \tau \sim \tau'}{\Gamma \vdash t \bowtie t'}\ [\text{E-Atom}]$$

$$\frac{\Gamma \vdash e \qquad \Gamma \vdash e'}{\Gamma \vdash e \wedge e'}\ [\text{E-And}] \qquad \frac{\Gamma \vdash e \qquad \Gamma \vdash e'}{\Gamma \vdash e \vee e'}\ [\text{E-Or}]$$

Extensions $\boxed{\Gamma \vdash r}$

$$\frac{\Gamma \vdash e}{\Gamma \vdash \langle \wedge e \rangle_i}\ [\text{R-And}] \qquad \frac{\Gamma \vdash e}{\Gamma \vdash \langle \vee e \rangle_i}\ [\text{R-Or}]$$

$$\frac{\Gamma \vdash r \qquad \Gamma \vdash r'}{\Gamma \vdash r\ r'}\ [\text{R-Cat}] \qquad \frac{\Gamma \vdash r \qquad \Gamma \vdash r'}{\Gamma \vdash r + r'}\ [\text{R-Sum}]$$

Queries $\boxed{\Gamma \vdash q}$

$$\frac{\begin{array}{c}\Gamma, \Gamma' \vdash t_i : \tau_i, i \in [1 \ldots n] \\ \Gamma, \Gamma' \vdash e \qquad \Gamma, \Gamma' \vdash r\end{array}}{\Gamma \vdash \langle \mathtt{select}\ t_1, \ldots, t_n\ \mathtt{from}\ \Gamma'\ \mathtt{where}\ e \rangle_i\ r}\ [\text{Q-Sel}]$$

$$\frac{\Gamma, x : \tau \vdash a_i, i \in [1 \ldots n] \quad \Gamma, x : \tau \vdash e \quad \Gamma, x : \tau \vdash r}{\Gamma \vdash \langle \mathtt{update}\ x : \tau\ \mathtt{set}\ a_1, \ldots, a_n\ \mathtt{where}\ e \rangle_i\ r}\ [\text{Q-Upd}]$$

$$\frac{\Gamma, x : \tau \vdash e \qquad \Gamma, x : \tau \vdash r}{\Gamma \vdash \langle \mathtt{delete}\ x : \tau\ \mathtt{where}\ e \rangle_i\ r}\ [\text{Q-Del}]$$

Figure 6. Rules for Type Checking Queries

names to types. As the types of identifiers are declared inside the query string, the bound query analysis needs only track the types of the values bound to the query parameters. Thus, we focus our attention on *parameter type environments* Γ, which map parameter names to types, not identifiers. We use P to denote the set of parameter names, T the set of types, and $TE = P \rightharpoonup T$ the set of all parameter type environments. We let Q be the set of queries.

A bound query is a pair (q, Γ) where $q \in Q$ and $\Gamma \in TE$. A bound query represents a query where some of the parameters have been set. We denote by $BQ = Q \times TE$ the set of all bound queries.

Our analysis will map each variable in the program to a set of bound queries. If we denote by V the set of Java variables, then the domain $\mathcal{D} = (D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ of our analysis is defined as:[1]

- $D \triangleq V \to 2^{BQ}$
- $\sqsubseteq$ is defined by $a \sqsubseteq b \triangleq \forall x \,.\, a(x) \subseteq b(x)$
- $\bot \triangleq \lambda x.\emptyset$ and $\top \triangleq \lambda x.BQ$
- $\sqcup$ is defined by $(a \sqcup b)(x) \triangleq a(x) \cup b(x)$
- $\sqcap$ is defined by $(a \sqcap b)(x) \triangleq a(x) \cap b(x)$

Flow Function. We use $F : \mathit{Stmt} \times D \to D$ to denote the flow function of our analysis, where Stmt represents the set of program statements. We now describe some representative cases of F. Given a map m, we denote by $m[a \mapsto b]$ the map where a has been made to map to b.

$F(v = \mathtt{CreateQuery}_k(s), in) = in[v \mapsto \{(q_k, \emptyset)\}]$
where q_k is the (extended) query computed by the string analysis for creation site k.

$F(v.\mathtt{SetParam}(p, x), in) = in[v \mapsto h(in(v))]$
where $h(R) = \{(q, \Gamma[p \mapsto \mathit{typeof}(x)] \mid (q, \Gamma) \in R\}$ and $\mathit{typeof}(x)$ is the type of the Java variable x.

For conditionals, the information coming into the conditional is propagated to both the true and the false successors. At merge points, the information from the two sides of the merge is joined using the lattice join operator $\sqcup$.

Checking. Now, consider a statement $y.\mathtt{getSingleResult}()$, and let bq be the final information computed by the above dataflow analysis right before the `getSingleResult` statement. Then we say that the given `getSingleResult` is *param-type-correct* iff the following condition holds:

$$\forall (q, \Gamma) \in bq(y) \,.\, \Gamma \vdash q$$

This condition states that for each bound query that flows to the `getSingleResult` statement, all the parameters of the query have been set using correct types. Note that this check will catch type errors due to improper comparisons and assignments and also errors due to parameters not being set.

3.3 Return Analysis

For clarity of exposition, we assume that the only way to execute a query is via a call to `getSingleResult` which re-

[1] Throughout the paper we use the abstract interpretation convention that $\bot$ represents no behaviors of the program and $\top$ represents all possible behaviors. Thus, $\bot$ is the most optimistic information, and $\top$ is the most conservative information.

turns a single object. Our algorithms can be easily extended to handle methods like `getResultList` which return lists of objects, and our implementation handles these cases. As all objects returned from JPA queries pass through a single API, namely `getSingleResult`, the type of these objects is the most general Java type possible, namely `Object`. Consequently, to access specific fields of objects returned from the database, the programmer must downcast to a more specific type. The goal of our return analysis is to guarantee that such downcasts, which are performed on values originating from a JPA query, will not fail at runtime, or dually, pointing out at compile-time, the downcasts that may fail.

Domain. The algorithm PropagateTypes for checking return types, shown in Figure 7, computes at each program point a map that stores for each variable x the set of types that x may contain which could have originated from queries. Thus domain $\mathcal{D}$ of the analysis is $\mathcal{D} = (D, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ where $D \triangleq V \to 2^T$, $\perp \triangleq \lambda x.\emptyset$, $\top \triangleq \lambda x.T$, and $\sqsubseteq, \sqcup, \sqcap$ are defined as in Section 3.2.

Algorithm PropagateTypes**.** The algorithm maintains a global *worklist* of methods to be analyzed, and a global map GF from field names to the set of types stored in those fields. As this map is global, fields are treated in a flow insensitive manner. We also store in the maps Sum_{in} and Sum_{out} an input and output summary for each method. For simplicity of exposition, we assume that methods only take one parameter. The input summary of a method is the set of types that flow into its parameter, and the return summary is the set of types that the method returns.

Below the declaration of globals in Figure 7, we define a simple construct that will help describe our algorithm. For a map s, we define $[s(i) := s(i) \cup v$ **if changed add** $m]$ to perform two tasks: first, it performs the update $s(i) := s(i) \cup v$; second it adds m to the global worklist of methods if the update to $s(i)$ has changed the value of $s(i)$.

The algorithm PropagateTypes starts by initializing *worklist* with the set of methods that contain calls to `getSingleResult`, after which it processes method from the worklist. In particular, while the worklist is not empty, PropagateTypes removes a method from the worklist and calls $\mathsf{FixPoint}(F, m.cfg, Sum_{in}(m))$. The FixPoint procedure (not shown here) uses standard techniques to compute the intraprocedural dataflow-analysis fixpoint of the flow function $F : Stmt \times D \to D$. In particular, FixPoint starts with $\perp$ at every program point, and then iteratively calls F until a fixed point is reached. Once the intraprocedural fixpoint has been computed, PropagateTypes uses ret_info to extract from the fixpoint δ the set of types computed for the return value of the method. This set, which is stored in r, is merged into the output summary of the current method, and if the output summary changes, then the method's callers – $callers(m)$ – are added to the worklist using the "if changed add" construct.

Globals
  $worklist : 2^M$ (worklist of methods)
  $GF : Field \to 2^T$ (map from fields to sets of types)
  $Sum_{in} : M \to 2^T$ (method input summaries)
  $Sum_{out} : M \to 2^T$ (method return summaries)

Notation
  For map s, define $[s(i) := s(i) \cup v$ **if changed add** $m]$ as:
    let $n := s(i) \cup v$
    if $s(i) \neq n$ **then**
      $s(i) := n$
      $worklist.\mathsf{Add}(m)$

Procedure PropagateTypes()
  $worklist :=$ new empty worklist of methods
  for each m that contains a call to `getSingleResult` **do**
    $worklist.\mathsf{Add}(m)$
  while $worklist$ not empty **do**
    let $m := worklist.\mathsf{Remove}$
    let $\delta := \mathsf{FixPoint}(F, m.cfg, Sum_{in}(m))$
    let $r := ret\_info(\delta)$
    $Sum_{out}(m) := Sum_{out}(m) \cup r$ **if changed add** $callers(m)$

Flow Function

$F(x = y, in) = in[x \mapsto in(y)]$

$F(x = y.\texttt{getSingleResult}(*), in) =$
$$in[x \mapsto \{\tau_q \mid (q, \Gamma) \in bq(y)\}]$$
  where τ_q is the return type of query q, and bq is the computed information for the incoming program point by the bound query analysis from Section 3.2.

$F(x = (\tau)y, in) =$
  if $\exists \tau' \in in(y)\ .\ \neg\tau' <: \tau$ **then**
    raise "Warning: JPA downcast may fail"
  return in

$F(x = *.f, in) = in[x \mapsto GF(f)]$

$F(*.f = y, in) =$
  $GF(f) := GF(f) \cup in(y)$ **if changed add** $readers(f)$
  return in

$F(x = y.m(a), in)$
  $Sum_{in}(m) := Sum_{in}(m) \cup in(a)$ **if changed add** m
  return $in[x \mapsto Sum_{out}(m)]$

Figure 7. Propagating `getSingleResult` Return Types

Flow Functions. The cases for the flow function $F : Stmt \times D \to D$ are shown at the bottom of Figure 7. For a simple assignment $x = y$, the flow function maps x to the set of types that y is mapped to. For a statement $x = y.\mathtt{getSingleResult}_i(*)$, the flow function maps x to the set of all types that could be returned by the query. The flow function for a type cast makes sure that the type cast is correct, and if not displays a warning. For a field read $x = *.f$, the flow function maps x to what the current flow-insensitive information is for the field f. For a field write $*.f = y$, the flow-insensitive information for f is updated, and if this information changes for f then all the methods that read f – $readers(f)$ – are added to the worklist. Finally, for a method call $x = y.m(a)$, the input summary for method m is updated with the set of types that a contains, and if m's input summary changes, then m is added to the worklist (using the 'if changed add" construct). The flow function also maps x to the current output summary of m (note that this output summary may not account for the new information that is coming into m, but if the new incoming information to m causes the analysis of m to change m's output summary, then m's callers, including the current method, will be reanalyzed).

4. Deep Refactoring

The goal of our refactoring algorithm is to automatically identify all the places in a JPA application that need to be updated if a class or field is renamed. Java references to the renamed entities can easily be handled using Eclipse's built-in refactorings for renaming a class or a field. The difficulty lies in updating raw strings that represent JPA queries or fragments of queries which are concatenated together by the Java code, due to the lack of a context within which to understand the query.

While building an extended query, QUAIL tags each query fragment with an identifier that uniquely determines the Java string literal that the fragment originated from. Once an extended query is constructed, QUAIL refactors the *entire* query, which makes it is easy to understand identifiers and their types. Finally, QUAIL propagates the changes back to the Java code using the tags.

We now describe how we refactor an extended query q. Once the query is refactored, propagating changes back to the Java strings using the tags is straightforward. QUAIL currently handles two refactoring tasks – class- and field-renamining.

Algorithm $\mathsf{RF}(\Gamma, \Delta, q)$**.** Our refactoring algorithm, shown in Figure 8, takes three parameters:

- Γ, a type environment, which for the top-level call to RF is empty,
- Δ, a refactoring specification, which is either a class renaming, written $C \mapsto C'$, or a field renaming, written $C.f \mapsto C.f'$, and
- q, an extended query.

Refactoring Function

$\mathsf{RF}(\Gamma, C \mapsto C', q) = q[C'/C]$

$\mathsf{RF}(\Gamma, \Delta, \mathtt{c}) = \mathtt{c}$

$\mathsf{RF}(\Gamma, \Delta, ?\mathtt{p}) = ?\mathtt{p}$

$\mathsf{RF}(\Gamma, \Delta, x) = x$

$\mathsf{RF}(\Gamma, C.f \mapsto C.f', lv.f'') =$
  if $f = f''$ **and** $\Gamma \vdash lv : C''$ **and** $C'' <: C$
  then $lv.f'$ **else** $lv.f''$

$\mathsf{RF}(\Gamma, \Delta, t \bowtie t') = \mathsf{RF}(\Gamma, \Delta, t) \bowtie \mathsf{RF}(\Gamma, \Delta, t')$

$\mathsf{RF}(\Gamma, \Delta, e \wedge e') = \mathsf{RF}(\Gamma, \Delta, e) \wedge \mathsf{RF}(\Gamma, \Delta, e')$

$\mathsf{RF}(\Gamma, \Delta, e \vee e') = \mathsf{RF}(\Gamma, \Delta, e) \vee \mathsf{RF}(\Gamma, \Delta, e')$

$\mathsf{RF}(\Gamma, \Delta, lv{:=}t) = \mathsf{RF}(\Gamma, \Delta, lv){:=}\mathsf{RF}(\Gamma, \Delta, t)$

$\mathsf{RF}(\Gamma, \Delta, \langle \wedge e \rangle_i) = \langle \wedge \mathsf{RF}(\Gamma, \Delta, e) \rangle_i$

$\mathsf{RF}(\Gamma, \Delta, \langle \vee e \rangle_i) = \langle \vee \mathsf{RF}(\Gamma, \Delta, e) \rangle_i$

$\mathsf{RF}(\Gamma, \Delta, r + r') = \mathsf{RF}(\Gamma, \Delta, r) + \mathsf{RF}(\Gamma, \Delta, r')$

$\mathsf{RF}(\Gamma, \Delta, r\ r') = \mathsf{RF}(\Gamma, \Delta, r)\ \mathsf{RF}(\Gamma, \Delta, r')$

$\mathsf{RF}(\Gamma, \Delta, \langle \mathtt{select}\ t_1, \ldots, t_n\ \mathtt{from}\ \Gamma'\ \mathtt{where}\ e \rangle_i\ r) =$
  let $\Gamma'' := \Gamma, \Gamma'$
  let $t'_1, \ldots, t'_n := \mathsf{RF}(\Gamma'', \Delta, t_1), \ldots, \mathsf{RF}(\Gamma'', \Delta, t_n)$
  let $e' := \mathsf{RF}(\Gamma'', \Delta, e)$
  let $r' := \mathsf{RF}(\Gamma'', \Delta, r)$
  $\langle \mathtt{select}\ t'_1, \ldots, t'_n\ \mathtt{from}\ \Gamma'\ \mathtt{where}\ e' \rangle_i\ r$

$\mathsf{RF}(\Gamma, \Delta, \langle \mathtt{update}\ x{:}\tau\ \mathtt{set}\ a_1, \ldots, a_n\ \mathtt{where}\ e \rangle_i\ r) =$
  let $\Gamma' := \Gamma, x{:}\tau$
  let $a'_1, \ldots, a'_n := \mathsf{RF}(\Gamma', \Delta, a_1), \ldots, \mathsf{RF}(\Gamma', \Delta, a_n)$
  let $e' := \mathsf{RF}(\Gamma', \Delta, e)$
  let $r' := \mathsf{RF}(\Gamma', \Delta, r)$
  $\langle \mathtt{update}\ x{:}\tau\ \mathtt{set}\ a'_1, \ldots, a'_n\ \mathtt{where}\ e' \rangle_i\ r'$

$\mathsf{RF}(\Gamma, \Delta, \langle \mathtt{delete}\ x{:}\tau\ \mathtt{where}\ e \rangle_i\ r) =$
  let $\Gamma' := \Gamma, x{:}\tau$
  let $e' := \mathsf{RF}(\Gamma', \Delta, e)$
  let $r' := \mathsf{RF}(\Gamma', \Delta, r)$
  $\langle \mathtt{delete}\ x{:}\tau\ \mathtt{where}\ e' \rangle_i\ r'$

Figure 8. Refactoring

Given these three parameters, the RF function returns a refactored version of q as follows.

Class-Renaming. The first case in the definition of RF handles the class-rename refactoring $C \mapsto C'$. In this case RF simply replaces all occurrences of the class name C with the new class name C'. We use the notation $q[C'/C]$ to represent q with class C replaced with C'. This substitution is syntactic and is performed by traversing the abstract syntax tree (AST) of the query, and replacing each node in the AST that represents class C with C'.

Field-Renaming. The remaining cases in the definition of RF handle the field-rename refactoring $C.f \mapsto C.f'$. For this refactoring, RF performs a traversal of the query's AST, maintaining in Γ the set of accumulated bindings from the base query's type declarations. Using these bindings, the fifth case in the definition of RF renames a field access $lv.f$ to $lv.f'$ if lv's type is a subtype of C.

Loops. Whereas the QUAIL typechecker fails to typecheck query strings that are created using a loop, a simple insight allows QUAIL to handle such query strings in the context of refactoring: for the purposes of refactoring, QUAIL simply needs to statically identify all query fragments, and this can be done by simply assuming that the loop body creating the query executes once or zero times. This approach is guaranteed to refactor query fragments correctly if there are no type errors to begin with.

5. Evaluation

To evaluate our deep typechecking and deep refactoring algorithms, we implemented these algorithms in an Eclipse plugin called QUAIL. For bound query analysis and return analysis, we use Eclipse's libraries to walk over the Java instructions. For the type analysis of queries, we built our own JPA query parser and type checker. The type checker uses Eclipse's libraries to identify the types of lvalues (for example `w.website` and `w.link` in Figure 2). In total, QUAIL consists of 5700 lines of Java code.

Our goal is to evaluate QUAIL along two dimensions:

- **Precision of deep typechecking.** Can QUAIL prove a large fraction of query executions type safe?
- **Utility of deep refactoring.** Can QUAIL help programmers perform refactorings that cross-cut Java and JPA?

To answer these questions, we ran QUAIL on several benchmarks, which are listed in Table 1. These benchmarks were executed on a 2.6 GHz Core2Duo™ machine with 4GB of RAM running Ubuntu 7.10, Eclipse 3.3, and Sun JDK 1.6. Table 1 shows for each benchmark the number of lines of Java code, the number of calls to `CreateQuery`, the number of calls to `SetParam`, and the number of sites that execute queries. This last number is further split into the number of calls to `getSingleResult`, `getResultList`, and `executeUpdate`. The `executeUpdate` procedure is a query execution statement used for updates and delete queries. The largest of our benchmarks is `Roller`, a web blogging software system that comprises 82,907 lines of code. Named queries, a feature of JPA that allows queries to be named in the object-relational model and then invoked later by name, are currently handled by inlining the query string at the query execution site. However, we believe it would be straightforward to extend QUAIL to handle these directly in the object-relational mapping.

Deep typechecking. Table 2 shows the results of deep type checking on our benchmarks. For each benchmark, we count the number of query execution statements that pass each of the following QUAIL checks:

- param: parameters passed to the executed query using `SetParam` are of the correct type.
- completeness: parameters are set on all paths leading to the query execution statement.
- result: typecasts on objects originating from the query execution statement will not fail at runtime.

The "total" column refers to the total number of query execution statements that were considered. For result checking, the total is smaller than for param and completeness because result checking does not check calls to `executeUpdate` (which does not return any objects from the database). The "time" column, which lists the number of seconds to perform typechecking on each benchmark, demonstrates the scalability of deep typechecking.

The results show that overall QUAIL succeeds in 84% of the checks that it performs. The correctness of 84% of the execution sites follows from the soundness of QUAIL. To increase our confidence, we manually examined all 84% to make sure that they are indeed type correct. Furthermore, although the benchmarks presented do not contain query type errors, our regression test suite includes an array of erroneous queries which we check for failure.

The query execution sites that QUAIL cannot analyze involve either construction of the query string by iterating over heap-based data structures or reflection to obtain the name of an object's class at run time. Our dataflow analysis cannot currently handle such cases, but we believe that with some extra precision, QUAIL could be extended to deal with these programming patterns. Furthermore, although QUAIL's analysis is neither path sensitive nor interprocedural, neither of these would increase precision across the 100,000 lines of deployed Java code in our benchmarks. That is, neither of these would eliminate any of the false positives. Furthermore, while imprecise aliasing information can lead to imprecision for any static analysis, we have found that in these benchmarks there are unique, method-local references to query objects, and so a precise alias analysis is not required.

Benchmark	LOC	`CreateQuery`	`SetParam`	Number of query execution sites						
				Single	+	List	+	Update	=	Total
`CaveatEmptor`	3,369	6	5	1	+	5	+	0	=	6
`PetStore`	5,435	15	20	0	+	15	+	0	=	15
`Planet`	14,525	14	9	4	+	10	+	0	=	14
`Roller`	82,907	128	165	19	+	65	+	17	=	101
Total	106,236	163	199	24	+	95	+	17	=	136

Table 1. Benchmark statistics

Benchmark	Kind of check	# pass	# fail	Total	Time (seconds)
`CaveatEmptor`	param	3 (50%)	3 (50%)	6 (100%)	2.8
	completeness	3 (50%)	3 (50%)	6 (100%)	
	result	3 (50%)	3 (50%)	6 (100%)	
`PetStore`	param	11 (73%)	4 (27%)	15 (100%)	5.7
	completeness	11 (73%)	4 (27%)	15 (100%)	
	result	10 (67%)	5 (33%)	15 (100%)	
`Planet`	param	11 (79%)	3 (21%)	14 (100%)	6.1
	completeness	11 (79%)	3 (21%)	14 (100%)	
	result	12 (86%)	2 (14%)	14 (100%)	
`Roller`	param	89 (88%)	12 (12%)	101 (100%)	26.5
	completeness	89 (88%)	12 (12%)	101 (100%)	
	result	74 (88%)	10 (12%)	84 (100%)	
Total		327 (84%)	64 (16%)	391 (100%)	41.1

Table 2. Number of checks that succeed in deep type checking

Deep refactoring. To evaluate QUAIL's utility for performing deep refactoring, we performed a variety of deep refactoring tasks using QUAIL. Table 3 summarizes the results of doing these tasks. We picked those refactorings of fields/classes that occurred most often in query strings and query string fragments. Our refactoring is implemented within the Eclipse refactoring framework, which makes sure that the appropriate preconditions are satisfied before performing any field/class rename refactoring. All the results in Table 3 are for the part of the refactoring that deals with query strings – refactoring references in Java is handled by Eclipse's' refactorings, and we do not provide statistics for this part of the refactoring.

For each refactoring, the column labeled "# changes" in Table 3 shows the number of changes that QUAIL performed. The column labeled "# refacstr" shows the number of refactorable strings, which are string literals occurring in the Java code that represent queries or fragments of queries. These are the strings that a programmer using existing tools would have to consider to perform the refactoring. Note that in some cases, the number of changes is larger than the number of refactorable strings, because there can be multiple changes per string. The column labeled "# refacloc" shows the number of places in the refactorable strings that are refactorable: for a class-rename, this is the number of class references occurring in refactorable strings; for a field-rename, this is the number field references occurring in refactorable strings. The "# refacloc" column is an estimate of number of places in the code that a programmer using existing tools would have to consider to perform the refactoring. Finally, the column labeled "# warnings" shows the number of locations in the code that QUAIL determined may need refactoring, but that QUAIL was not able to refactor because of imprecision in the QUAIL string analysis. These are left to the programmer to refactor. As Table 3 shows, the "# warnings" column is much smaller than the "# refacloc", showing that QUAIL drastically reduces the number of places in the code that the programmer needs to look at in order to perform a refactoring. The time to perform these refactorings is within 5% of the corresponding typechecking time.

Comparison with hand-refactoring. Performing refactorings by hand is difficult for two reasons. First, since some identifiers like `name` are very common, it is not sufficient to simply look at identifier names when refactoring field names – one has to also look at the type of the receiver, and doing this inside query fragments is tedious and error prone, because type information inside these query strings is not readily available. This may lead the programmer to incorrectly change a field having the correct name, but on the wrong class. Second, it is often hard to find all strings that represent

Benchmark	Refactoring performed	# changes	# refacstr	# refacloc	# warnings
CaveatEmptor	`Bid` ↦ `Offer`	4	3	5	0
	`Item` ↦ `Product`	1	3	5	0
	`Bid.amount` ↦ `Bid.amt`	4	3	7	0
	`Item.id` ↦ `Item.SerialNo`	2	3	7	0
PetStore	`Product` ↦ `Pet`	4	11	13	0
	`Item` ↦ `PetStoreItem`	5	11	13	0
	`Product.productID` ↦ `Product.id`	2	11	24	0
	`Item.productID` ↦ `Item.id`	5	11	24	0
Planet	`Planet` ↦ `World`	2	23	14	0
	`Subscription` ↦ `Feed`	6	23	14	0
	`Planet.handle` ↦ `Planet.alias`	3	23	17	0
	`Feed.feedURL` ↦ `Feed.url`	2	23	17	0
Roller	`WeblogEntry` ↦ `Entry`	13	262	119	4
	`User` ↦ `Blogger`	20	262	119	4
	`AutoPing` ↦ `APing`	5	262	119	4
	`Weblog` ↦ `Blog`	6	262	119	4
	`WeblogEntry.pubTime` ↦ `WeblogEntry.time`	18	262	337	4
	`User.userName` ↦ `User.name`	7	262	337	4
	`User.enabled` ↦ `User.valid`	9	262	337	4
	`AutoPing.pingTarget` ↦ `AutoPing.target`	3	262	337	4
	`Weblog.name` ↦ `Weblog.blogName`	6	262	337	4

Table 3. Statistics for deep refactoring

query fragments, especially when the refactored field is a common identifier like `name`, for which a grep will flood the programmer with false positives. This may lead the programmer to miss changes that should be made. Neither of these two kinds of errors can be found using the Java type system. QUAIL will not only avoid these errors altogether, but if the programmer for some reason chooses to perform JPA query refactoring by hand, the QUAIL type checker would be able to find such errors.

As anecdotal evidence, a software developer who works with Hibernate [9] in a production environment has described a problem he faces with a persistent class given the unfortunate name `Test`. The class needs to be renamed to something more descriptive, but doing so is practically impossible due to the frequency with which the string "Test" occurs in the code-base: fixing occurrences by hand is too time consuming and error prone, and a search and replace will result in the incorrect replacement of far too many strings. The developer expressed confidence that using a tool like QUAIL the decision may be made to perform the refactoring.

Unit Testing. In a realistic scenario, a programmer who would perform refactorings of query strings by hand would make use of unit tests in order to make sure that the refactoring is performed correctly, and also to identify places that need to be refactored further. However, even such safeguards can easily fail. For example, in the case of `Roller`, we found that refactoring `Weblog.handle` causes 15 query strings to be changed, but despite the many unit tests in `Roller`, only 8 of these queries are covered by unit tests. The remaining 7 are slight variations on the ones that are tested. If a programmer inadvertently makes a typo while refactoring one of the 7 queries that are not covered by unit tests, then the typo will not be caught at the unit level, and may even make it into a production system.

6. Related work

The research most closely related to our work can be partitioned into several categories.

Static Analyses for checking multi-lingual software are the most closely related line of work. Of these, we have drawn inspiration from [21] which gives a static technique for typechecking the SQL queries dynamically generated by programs using JDBC [8]. The technique is based on using a string analysis [3] to compute an automaton overapproximating the set of strings that can be sent as queries, and an algorithm for typechecking the automaton. In contrast to QUAIL, this work focuses on the typechecking the SQL queries against the database schema, and unlike QUAIL, ignores parameterized queries and return value type checking. Further, as JDBC provides lower-level access to the database (in contrast to the higher level Object-Relational Map of JPA), the question of deep refactoring does not arise.

Further afield, there have been several recently proposed static analyses for checking multilingual software. Examples include [7] which presents a type system and dataflow analysis for checking the correctness of the OCaml/C foreign function interface, and [19] which presents a system for ensuring the type safety of programs that combine C and Java.

Orthogonal Persistence is an approach to integrating programming languages and databases in a manner that entirely sidesteps the need for deep typechecking and refactoring. Here, the entire database is exposed to the programmer as a collection of objects that can be navigated. Examples include the PJAMA [1] and THOR [12] projects. In this tight coupling, the string based interface is eliminated and hence one can directly use the language's typechecking and refactoring tools. The drawback with this approach is that it sacrifices the bulk-access optimizations possible by using SQL. To recover some of the benefits of these optimizations [22] proposes the use of abstract interpretation to statically extract SQL queries from the code. It remains to be seen whether in practice the query extraction yields systems with competitive performance.

Programming Languages supporting SQL allow programmers to efficiently interact with the database while staying within a single programming language. This is achieved by providing syntax to describe queries in a manner that permits efficient compilation to SQL. Two classical examples are the DBPL [18] and Tycoon [15] languages. More recently proposals include the functional languages Kleisli [24], and Links [5] which provides a single unified language for Web-programming. A more backwards-compatible approach are special extensions to existing languages, like Linq [2] for C♯, and [23] for Java. The most compatible approach is to embed queries as instances of special classes within Java [4] or C++ [16]. As for orthogonal persistence, each of these approaches eliminate the need for deep typechecking and refactoring. However, we conjecture that adoption has been limited by the fact query strings provide a more flexible and readable way to harness all of SQL, in contrast to these approaches which only incorporate restricted subsets. Moreover, there is a large amount of legacy code that would have to be rewritten to benefit from these techniques.

7. Conclusions and Future Work

In this paper we presented QUAIL, a tool for deep typechecking and refactoring Java code that uses query strings to interact with databases, thereby allowing programmers to use the efficient and flexible string-based interface in a safe manner. Based on our experiences building and evaluating QUAIL, we believe there are several avenues for future work. First, we would like to improve the precision of our technique to eliminate the few false positives where we incorrectly flag safe query executions as potential errors. For example, we would like to extend QUAIL so that it can typecheck queries with dynamically generated parameters, *i.e.,* where the parameters are created and set by iterating over a list or array. Second, we would like to enrich the type system to prove more properties statically. For example, verifying that queries which are called with `getSingleResult` indeed determine at most one object. This would require a more precise modelling of the constraints of the database schema, *e.g.* primary keys, to verify that the query returned a unique result. Further, we would like to do some basic semantic sanity checking on the queries, *e.g.* that the `WHERE` clause is not a tautology or contradiction. Finally, we would like to extend our techniques to obtain deep versions of other software engineering tasks that are standard within a single language. Examples include deep code completion, where the tool would run in an online manner and suggest class and field completions for query string fragments, and deep impact analysis, where the tool would alert the programmer to the string fragments that may change due to alterations in the database schema or vice versa.

References

[1] Malcolm P. Atkinson, Laurent Daynès, Mick J. Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent java. *SIGMOD Record*, 25(4):68–75, 1996.

[2] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to c#. In *OOPSLA*, pages 479–498, 2007.

[3] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *SAS*, pages 1–18, 2003.

[4] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE*, pages 97–106, 2005.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, LNCS 4709. Springer, 2006.

[6] Jacques-Antoine Dub, Rick Sapir, and Peter Purich. Oracle application server toplink application developers guide, 10g (9.0.4). Oracle Corporation, 2003.

[7] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *PLDI*, pages 62–72, 2005.

[8] G. Hamilton and R. Cattell. Jdbc: A java sql api. Sun Microsystems, 1997.

[9] Hibernate reference documentation, may 2005. `http://www.hibernate.org/hib_docs/v3/reference/en/html`.

[10] Java persistence api faq. `http://java.sun.com/javaee/overview/faq/persistence.jsp`.

[11] Glassfish implementation of the java persistence api. `https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html`.

[12] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and efficient sharing of persistent objects in thor. In *SIGMOD Conference*, pages 318–329, 1996.

[13] David Maier. Representing database programs as objects. In M. Atkinson, editor, *Advances in Database Programming Languages*, pages 377–386. Springer, 1990.

[14] V. Matena and M. Hapner. Enterprise java beens specification 1.0. Sun Microsystems, 1998.

[15] F. Matthes, G. Schroder, and J. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M. Atkinson, editor, *Fully Integrated Data Environments*, LNCS. Springer-Verlag, 1995.

[16] Russell A. McClure and Ingolf H. Krüger. Sql dom: compile time checking of dynamic sql statements. In *ICSE*, pages 88–96, 2005.

[17] C. Russell. Java data objects (jdo) specification jsr-12. Sun Microsystems, 2003.

[18] Joachim W. Schmidt and Florian Matthes. The dbpl project: Advances in modular database programming. *Inf. Syst.*, 19(2):121–140, 1994.

[19] Gang Tan and Greg Morrisett. Ilea: inter-language analysis across java and c. In *OOPSLA*, pages 39–56, 2007.

[20] Murali Venkatrao and Michael Pizzo. Sql/cli - a new binding style for sql. *SIGMOD Record*, 24(4):72–77, 1995.

[21] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar T. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[22] Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL*, pages 199–210, 2007.

[23] Darren Willis, David J. Pearce, and James Noble. Efficient object querying for java. In *ECOOP*, pages 28–49, 2006.

[24] Limsoon Wong. Kleisli: a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.