Eliminating Impedance Mismatch in C++

Joseph (Yossi) Gil * and Keren Lenz
Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
yogi, lkeren@cs.technion.ac.il

ABSTRACT

Recently, the C# and the VISUAL BASIC communities were tantalized by the advent of LINQ [18]—the *Language INtegrated Query* technology from Microsoft. LINQ represents a set of language extensions relying on advanced (some say hard to understand) techniques drawn from functional languages such as type inference, λ-expressions and most importantly, monads. The 3^{rd} edition of C# just as the 9^{th} of VISUAL BASIC allow programmer to directly access relational and XML-based databases from within the programming language. We show that very similar capabilities can be achieved in the C++ programming language without relying on any language extensions, compiler modifications, external processing tools, or any other vendor specific machinery: ARARAT is a C++ template library whose objective is type safe generation of SQL statements for access relational database systems. Learning curve is minimal since ARARAT resembles relational algebra, which is at the core of SQL.

1. INTRODUCTION

Most software applications, arguably, all but the trivial, use a database for data management and storage. These applications are usually written in high level programming languages such as C++, JAVA [2] and C#. To use such a database, applications must produce strings containing statements in the database language (usually SQL) that are sent to the database engine for execution.

A direct use of this naive approach results in tremendous waste of precious developer resources on the recalcitrant problem [5, 8] of producing such accurate output. The generation of SQL (or XPath for that matter) output are nothing but *computer programs produced by computer programs*. It is well known that computer programming is a difficult, costly, and error prone task for humans. This is the reason that it is doubly more difficult to write computer programs that generate other programs.

The seam between the programming language and database language, that is the programmatic production of SQL commands, is a source for many potential problems. The generated statements are usually checked for correctness only at runtime, when executed against the database, which might lead to late discovery of malfunctioning, sometimes even after the product is shipped to customers. Further, precisely at this seam, exploiting small errors occurring during this tedious work, SQL injection attacks occur.

The ARARAT system sets its goal at automatic and safe generation of SQL code from C++. In one way, ARARAT is a C++ language enhancement, the first of its kind to offer immunity to these errors. In another, almost magically, this enhancement is achieved without any modifications to the compiler.

There are many C++ database access libraries, free, shareware and commercial including Ultimate++[1], Postgress [2], IBPP[3], DTL[4], libodbc++[5], MySQL C++[6], OTL[7], SQLAPI++[8] SOCI[9], Terimber[10], CQL[11], SourcePro DB[12], but invariably, they are all wrappers around the SQL server API, taking charge of issues such as connection management. Security issues and smooth integration with the host language, as what is offered by ARARAT are not handled.

2. A SHORT EXAMPLE

Table 2.1 introduces a database schema that will be used as a running example. There are two relations in this database, representing employees and departments.

Tab. 2.1: A database schema, to be used as running example.

Relation	Fields
EMPLOYEE	ID(**int**), FIRST_N(**varchar**), LAST_N(**varchar**), DEPTNUM(**smallint**), LOCATION(**varchar**)
DEPARTMENT	ID(**smallint**), NAME(**varchar**), DIVNUM(**smallint**)

Figure 2.1 shows a simple C++ function that might be used in an application that uses the database described in Table 2.1: Function `get_employees` receives a short integer `dept` and a string `first`, and returns an SQL statement in string format which evaluates to the set of all employees who work in department `dept` bearing the first name `first`. The function presented in the figure also takes care of the special cases that `first` is `null` or `dept` is not a valid department number.

*Research supported in part by the IBM faculty award

VLDB '07, September 23-28, 2007, Vienna, Austria.

[1] http://http://www.ultimatepp.org/
[2] http://http://www.postgresql.org/docs/8.1/interactive/libpq.html
[3] http://http://www.ibpp.org/
[4] http://http://dtemplatelib.sourceforge.net/index.htm
[5] http://http://libodbcxx.sourceforge.net/
[6] http://http://mysqlcppapi.sourceforge.net
[7] http://http://otl.sourceforge.net/home.htm
[8] http://http://www.sqlapi.com/index.html
[9] http://http://soci.sourceforge.net
[10] http://http://www.terimber.com
[11] http://http://www.cql.com/
[12] http://http://www.roguewave.com/sourcepro/sourcepro.cfm

```
1 char* get_employees(short dept, char* first) {
2    bool first_cond = true;
3    string s(
4       "SELECT FIRST_N, LAST_N FROM EMPLOYES "
5    );
6    if (dept > 0){ //valid dept number
7       s.append( "WHERE DEPTNUM = ' ");
8       s.append(itoa(dept));
9       s.append("'");
10      first_cond = false;
11   }
12   if (first == null)
13      return s;

15   if (first_cond)
16      s.append("WHERE ");
17   else
18      s.append("AND");
19   s.append("FIRST_N= ' ");
20   s.append(first);
21   s.append("'");

23   return s;
24 }
```

Fig. 2.1: Function returning an erroneous SQL query string.

The *expected* returned value of this function is a string such as

```
SELECT * FROM EMPLOYEES
   WHERE
       DEPTNUM = '3'
   AND
       FIRST_NAME = 'John'
```

The equivalent function using in ARARAT, is shown in Figure 2.2.

```
1 char* get_employees(short dept, char* first) {
2    DEF_V(e,EMPLOYEE[FIRST_N,LAST_N]);
3    if (first != null) e /= (FIRST_N == first);
4    if (dept > 0) e /= (DEPTNUM == dept);
5    return e.asSQL();
6 }
```

Fig. 2.2: A re-implementation of `get_employees` (Figure 2.1), using ARARAT.

Comparing the two figures, we see that the ARARAT version is much shorter. To understand the ARARAT code all we need to know is that selection (in the relational algebra sense) is represented by C++ division operator `/`, while projection into a set of fields is carried out by the square brackets operator (`[]`). Having learned that, we believe many will agree that the ARARAT code is also more elegant.

Further, as it turns out, the ARARAT version solves the many errors found in Figure 2.1, including a *misspelled name* in line 4, a *syntax error* in the case that both parameters are non-nulls (see line 18), *vulnerability* to code injection and more. These errors either cannot happen with ARARAT (SQL code produced by it is always syntactically correct), or are detected at compile-time (schema changes or misspelling will result in a compilation error.)

A reformulation of Figure 2.2 in LINQ is shown in Figure 2.3. Examining the LINQ version, we see that in contrast with the ARARAT implementation, the query code uses keywords which are reminiscent of SQL, but the query code is more complicated, and one must use an iteration variable to specify the query. Further, the fact that LINQ has no query objects which can be stored in variables, does not make it possible to define the query in stages. Instead, one needs to refine the query result, which can result in loss of optimization opportunities. Further, the projection operation must take place after the selection, since it eliminates fields that are used in the selection criterion. As a result the intermediate query results contain unnecessary fields and more data than necessary is transferred and processed.

```
1 void get_employees(short dept, string first){
2   var emps = (from e in employees select e);
3   if (dept != 0)
4      emps = (from e in emps
5              where e.dept == dept select e);
6   if (first != null)
7      emps = (from e in emps
8          where e.firstName.Equals(first)
9              select e);
10  var results = (from e in emps
11       select new {e.firstName, e.lastName});
12  ...
13 }
```

Fig. 2.3: The LINQ equivalent of Figure 2.2.

3. A BRIEF TUTORIAL

A programmer wishing to execute a database query must create first a *query object*, which encodes both the specification of the query and the scheme of its result. This query object can then be used for purpose such as

1. *Query Execution.* method `asSQL()` sent to a query object returns a valid injection-proof SQL statement representing the query execution.
2. *Type Definition.* Macro `TUPLE_T`, when applied to a query object returns the type of the tuples that the query object may return. This type is a simple C++ **`struct`**, with a standard mapping of SQL types to C++ types. Further, this generated type is assignment compatible with all tuples which include the same field set. This type can be used as a parameter to collection and data structures libraries, typically STL.
3. *Composing Compound Query Objects.* Two query objects can be combined using relational algebra operators to generate a compound query objects. A query object can be compound with a boolean *selection expression* to restrict its result set, or projected to restrict the columns set. Other composition operators include sorting and limit selection as in SQL.

A *primitive* query object for each of the relations in the input database must be prepared, either manually (the encoding is simple and straightforward) or by a simple tool. The content of each primitive query object is an encoding of the pseudo-code instruction: "return all fields of the relation". The C++ statement

```
EMPLOYEE.asSQL()
```

(for example) will return the following SQL statement

```
select * from EMPLOYEE;
```

A programmer may compose more interesting query objects out of the primitives. For this composition, our library provides a number of functions and overloaded operators. Each of the relational algebra operators has a C++ counterpart. It is thus possible to write relational algebra expressions, almost verbatim, in C++.

Figure 3.1 shows a simple C++ program demonstrating how a compound query object is put together in ARARAT. This query object is then converted to an SQL statement ready for execution.

In lines 11–16 of this figure, a compound query object is generated in two steps:

- First (line 11), expression

  ```
  EMPLOYEE / (DEPTNUM > 3 && SALARY <= 3.14))
  ```

  evaluates to the query object representing a selection of these tuples of relation `EMPLOYEE` in which `DEPTNUM` is greater than 3 and `SALARY` is no greater than 3.14.

  The syntax is straightforward: the selection criterion is written as a C++ Boolean expression, and **`operator /`** is used for applying this criterion to `EMPLOYEE`.

```
1 #include "rat" // Global RAT declarations and macros
2 #include "employees.h"
3  // Primitive query objects and scheme of the EMPLOYEE database

5 DEF_F(FULL_N); DEF_F(NUMBER);
6 // Define field names which were not defined in the input scheme

8 int main(int argc, char* argv[]) {
9  const string s = (
10   (EMPLOYEE / (DEPTNUM > 3
11      && SALARY < 3.14)) // Selection of a tuple subset
12   [
13    FIRST_N, LAST_N,
14    FULL_N(cat(LAST_N,", ", FIRST_N)),
15    NUMBER(ID)
16   ]
17  ).asSQL();
18  //... execute the SQL query in s using e.g., ADO.
19  return 0;
20 }
```

Fig. 3.1: Writing a simple relational algebra expression in ARARAT.

- Then, (lines 12–16), an array access operation, i.e., `operator []`, is employed to project these tuples into a relation schema consisting of four fields: `FIRST_N`, `LAST_N`, `FULL_N` (computed from `FIRST_N` and `LAST_N`), and `EMPNUM` (which is just field `ID` renamed).

  Note that the expression `cat(LAST_N,", ", FIRST_N)` produces a new (anonymous) field whose content is computed by concatenating three strings. The function call operator is then used to associate field name `FULL_N` with the result of this computation. Similarly, expression `EMPNUM(ID)` uses this operator for field renaming.

After this query object is created, its function member `asSQL()` is invoked (in line 17) to convert it into an equivalent SQL statement ready for execution:

```
select  FIRST_N, LAST_N,
        concat(LAST_N,", ", FIRST_N ) as FULL_N,
        EMPNUM as ID
from EMPLOYEE
        where DEPTNUM > 3 and SALARY <= 3.14;
```

This statement is assigned, as a string, to variable `s`.

Figure 3.2 shows another function, which uses ARARAT join operator to combine two query objects.

```
1 DEF_F(EMPNUM); DEF_F(DEPT_NAME);

3 char* emp_nums_and_depts() {
4  return (
5    (EMPLOYEE[EMPNUM(ID), ID(DEPTNUM)]*DEPARTMENT)
6    [EMPNUM, DEPT_NAME(NAME)]).asSQL();
7 }
```

Fig. 3.2: Using join operator in ARARAT.

In this function, the `operator *` is used for joining two relations. Note that field `DEPTNUM` in relation `EMPLOYEE` corresponds to field `ID` in relation `DEPARTMENT`. In order to perform a natural join these fields must have the same name. Therefore, we perform the appropriate renaming before the join operation.

It is also worth noting that both relations contain a field named `ID`, but its type is different. This is made possible due to the late binding of a field name to a relation.

As we saw, the usual C++ operators including comparisons and logical operators may be used in selection condition *and* in making the new fields. Table 3.1 summarizes the ARARAT equivalents of the main operators of relational algebra.

As can be seen in the table, the operators of relational algebra can be written in C++, using either a global function, a member function, or (if the user so chooses) with an intrinsic C++ (overloaded) operator: selection in relational algebra is represented by

Tab. 3.1: ARARAT equivalents of relational algebra operators.

Relational Algebra Operator	ARARAT Operator	ARARAT Function	SQL equivalent
selection $\sigma_c R$	R/c	`select(`R`,`c`)` R`.select(`c`)`	**select** $*$ **from** R **where** c
projection $\pi_{f_1,f_2} R$	R`[`f_1`,`f_2`]`	`project(`R`,(`$f1$`,`$f2$`))` R`.project((`$f1$`,`$f2$`))`	**select** f_1, f_2 **from** R
union $R_1 \cup R_2$	R_1`+`R_2	`union(`R_1`,`R_2`)` R_1`.union(`R_2`)`	R_1 **union** R_2
difference $R_1 \setminus R_2$	R_1`-`R_2	`subtract(`R_1`,`R_2`)` R_1`.subtract(`R_2`)`	R_1 - R_2
(natural) join $R_1 \bowtie R_2$	R_1`*`R_2	`join(`R_1`,`R_2`)` R_1`.join(`R_2`)`	R_1 **join** R_2
left join $R_1 =\times R_2$	R_1`<<`R_2	`left_join(`R_1`,`R_2`)` R_1`.left_join(`R_2`)`	R_1 **left join** R_2
right join $R_1 \times= R_2$	R_1`>>`R_2	`right_join(`R_1`,`R_2`)` R_1`.right_join(`R_2`)`	R_1 **right join** R_2
rename $\rho_{a/b} R$	$b(a)$	`rename(`a`,`b`)` a`.rename(`b`)`	a **as** b

```
    ((EMPLOYEE * DEPARTMENT)
/
    (DIVNUM == 2))[LOCATION]
```

(a) Operator overloading version

```
project(
   select(
      join(EMPLOYEE,DEPARTMENT),
      (DIVNUM == 2)
   ),LOCATION)
```

(b) global functions version

```
EMPLOYEE
   .join(DEPARTMENT)
      .select(DIVNUM == 2)
         .project(LOCATION)
```

(c) member functions version

Fig. 3.3: Three alternatives C++ expressions to compute a query object that, when evaluated, finds the locations of employees in division 2: using (a) overloaded operators (b) global functions, and (c) member functions.

`operator /`, projection by `operator []`, union by `operator +`, difference by `operator -`, natural join by `operator *`, left join by `operator <<`, right join by `operator >>`, and renaming by the function call operator `operator ()`.

ARARAT does not directly support Cartesian product. Since the join of two relations with no common fields is their cross product, this operation can be emulated (if necessary) by appropriate field renaming followed by a join.

The translation of any relational algebra expression into C++ is quite straightforward. Figure 3.3 shows how a query object for finding the locations of employees in division 2 can be generated using overloaded operators, global functions and member functions.

The composition of query objects with ARARAT is "type safe", in the sense that an attempt to generate illegal queries results in a compilation error. Thus, expressions `q1+q2` and `q1-q2` fail to compile unless `q1` and `q2` are query objects with the same set of fields. Similarly, it is illegal to project onto fields which do not exist in the relation, or select upon conditions which include such fields.

In the demonstration, we shall explain why the inherent delayed execution semantics of query objects avoids the hurdles of the monads semantics as found in LINQ. We will also show how query objects can be stored in variables, returned by functions, etc.

4. RELATED WORK AND DISCUSSION

LINQ probably represents the culmination of long research effort on the problem of seamless integration of database processing with high-level application languages (see, e.g., surveys in [3,4] as well

as Pascal-R [19], a persistent (extended) version of C [15] [1], integration of databases into SMALLTALK [10]- [7], the XJ [13] system integrating XML with JAVA, and many more. Other approaches of integrating SQL with an application language, include embedding, e.g., SQLJ [9], SchemeQL [20] and Embedded SQL for C[13], libraries such as SQL DOM system [17] whose structure reflects the constraints which the database scheme imposes, and allows production of only correct SQL statements, and static analyzer that checks that the program only produces correct SQL statements [6, 11, 12]. (The hosts of plain SQL wrappers mentioned above will not be discussed further.)

In comparison to these, and thanks to the template-based implementation, ARARAT achieves a high-level of integration with the host language without using a software library tailored to the database. In this respect, ARARAT is somewhat similar to the HaskellDB [16] system, a database oriented extension of HASKELL [14]. However, unlike HaskellDB, ARARAT supports dynamic queries, and by relying on C++ is more accessible to application programmers.

Another issue common to all approaches is the integration of the SQL type system with that of the host language. ARARAT automatically defines a record type for each possible fields combination, with a fixed mapping of SQL types to C++ types. This type can be used for data retrieval and manipulation.

Admittedly, just like many other systems mentioned above, ARARAT is language specific. The extension to languages such as JAVA requires modification of the host language to support C++ like templates; alternatively, one can modify JAVA, just as any other language, to support the ARARAT syntax.

Also, just like other research systems it is not a full blown database solution. The current implementation demonstrates the ideas with a safe generation of queries. ARARAT extensions for integration of query execution are left for future research, or for a commercialized implementation.

Finally, a few word are in place regarding the the adaptation of ARARAT to concrete implementation of SQL. It is well known that commercial vendors introduce their own (some may say idiosyncratic) dialect in adoption of the SQL standard. Simple statements, such as the ones used here will be understood by all such implementations. However, realization of more advanced features, such as nested queries may be very different in different database implementation. The procedure of translating the content of a query object into an SQL statement must be cognizant of these differences. This demand can be compared to the requirement that an implementation of a high level programming language, specifically a compiler, must be aware of the target language. Writers of multi-targets compilers address this difficulty by producing object code in intermediate form, which is then translated to the specific machine code. The same idea applies to ARARAT: the internal form of a query object is an arbitrarily complex expression in relational algebra. The familiar post order traversal of such an expression yields an evaluation procedure whose elements are simple SQL queries, each of which operates on previously computed intermediate results, yielding another such result. The final such operation computes the entire query result. This process lends itself to a general purpose process of translation into an arbitrary SQL implementation. The cost is of course is in that less-sophisticated database implementation may miss optimization opportunities, when given such a sequence of operations. A specialized translator may be in place for such implementations.

[13]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/esqlforc/ec_6_epr_01_3m03.asp,2004.

5. REFERENCES

[1] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA'87*.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[3] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[4] M. P. Atkinson and R. Welland. *Fully Integrated Data Env.: Persistent Prog. Lang., Object Stores, and Prog. Env.* Springer, 2000.

[5] T. Bloom and S. B. Zdonik. Issues in the design of object-oriented database programming languages. In *OOPSLA'87*.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS'03*.

[7] G. Copeland and D. Maier. Making Smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, 1984.

[8] J. E. Donahue. Integrating programming languages with database systems. In *Data Types and Persistence (Appin), Scotland, 1985*.

[9] A. Eisenberg and J. Melton. SQLJ Part 1: SQL routines using the Java programming language. *SIGMOD Rec.*, 28(4):58–63, 1999.

[10] A. Goldberg. *Smalltalk-80: The Interactive Prog. Env.* Addison-Wesley, 1984.

[11] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE'04*.

[12] W. G. J. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *ASE'05*.

[13] M. Harren, B. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML processing into Java, 2003.

[14] S. P. Jones. *Haskell 98 Language and Libraries: The Revisited Report*. Cambridge University Press, 2003.

[15] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 2nd ed., 1988.

[16] D. Leijen and E. Meijer. Domain specific embedded compilers. In *USENIX'99*.

[17] R. A. McClure and I. H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In *ICSE'05*.

[18] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *ICMD'06*.

[19] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. on Database Sys.*, 2(3):247–261, 1977.

[20] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Workshop on Scheme and Functional Programming (London, UK, 2002)*.