Simple and Safe SQL Queries with C++ Templates *

Joseph (Yossi) Gil † Keren Lenz

Technion—Israel Institute of Technology

yogi, lkeren@cs.technion.ac.il

Abstract

Most software applications use a relational database for data management and storage. Interaction with such a database is often done by letting the program construct strings with valid SQL statements, which are then sent for execution to the database engine. The fact that these statements are only checked for correctness at runtime is a source for many potential problems such as type and syntax errors and vulnerability to injection attacks.

The ARARAT system presented here offers a method for dealing with these predicaments, by coercing the host C++ compiler to do the necessary checks of the generated strings. A library of templates (and preprocessor directives) effectively extends C++ with a little language representing an augmented relational algebra formalism. Type checking of this language extension, as done by the template library, assures, at compile-time, the correctness of the generated SQL strings. All SQL statements constructed by the system are immune to injection attacks.

Standard techniques (e.g., "*expression templates*") for compile time representation of symbolic structures, are enhanced by our system to support a type system and a symbol table lookup of the symbolic structure. Our work may also open the way for embedding other domain specific languages in C++.

1. Introduction

ARARAT is a system that employs C++ templates mechanism to address the common problem of integrating a database language into a programming language. A library of templates extends C++ with a little language [7] representing an augmented relational algebra formalism. The proposed mechanism is general and can be used to embed other little languages in C++. Some readers may appreciate the degree of elegance in overloading C++ operators.

Much research effort was invested in the search for the holy grail of seamless integration of database processing with high-level application languages (see, e.g., surveys in [3,4]). Fruits of this quest include e.g., the work on Pascal-R [38], a persistent (extended) version of C [2], integration of databases into SMALLTALK- [12], the XJ [26] system integrating XML with JAVA, and many more. In this paper, we concentrate on integration of C++ with the *Structured Query Language*, better known as SQL, which is (still) the lingua franca of database processing, with adoptions of the standard [23], in all major database engines including Oracle [28], Microsoft SQL Server [31], MySQL [47], DB2 [34], and many more.

To use these databases, the application program must recognize that SQL engines (just as XML- and other database engines) expose a string based interface. Humans find many advantages in such an interface, including readability, well-defined- expressive syntax, declarative semantics, flexibility, etc. Application programs however are faced with the difficulty of producing *at runtime* SQL statements, feeding these to the SQL engine, and standing ready to process the results. Thus, those parts in the application which interact with the database can be thought of as programs that do the non-meager task of composing other programs, but with no aid of supportive, type-safe language environments and CASE tools.

The difficulties in producing SQL at runtime are discussed in detail in the literature (see, e.g., [8,15], or in brief below in Sec. 2). In a nutshell, these difficulties are rooted at the fact that the logic to produce correct SQL strings is complex, that correctness is relative to the scheme of the database (e.g., arithmetical operations are allowed only on fields which the scheme declares as numeric) and that errors in the process are not detected until runtime; further, certain errors may even invite *injection attacks* which compromise the safety and integrity of the entire database.

The ARARAT system presented here, designed for safe production of SQL queries from C++, has two components: a little language, henceforth called ARA, representing augmented *relational algebra* (RA) and a C++ templates and pre-processor directives library, nicknamed RAT, which realizes this language as a C++ extension. Thus, in a sense, RAT is the ARA compiler. Unique features of ARARAT include: (i) reliance on a RA metaphor, rather than directly on SQL. (ii) tight integration with the host language using *template programming*, with minimal reliance on external tools.

The main contributions of this paper are in techniques for extending C++ language, using the templates mechanism and without modifications to the compiler, to enable embedding of little languages within the language. Specifically, we describe the integration of the ARA little language that allows the generation of type safe SQL queries using existing C++ operators. On a broader perspective, our work may be used as a case study by designers of genericity mechanisms in future languages. Hopefully, from our experience may language designers gain intuition of the different features that genericity should offer. For example, we believe our work makes part of the case for including a `typeof` like operator. Also, language designers may consider favorably making generics powerful enough for doing operations such as symbol table lookup, yet being careful not to achieve full Turing completeness, which will bring the question of whether the compiler halts on a given input, to the verge of undecidability.

Relational Algebra. Historically, SQL emerged from the seminal work of Codd [10] on relational algebra. In a sense, SQL makes it possible to code RA expressions in a syntax which is more readable to humans, and closer to natural language. These advantages are

* The support work of Marina Rahmatulin, even on the eve of her wedding day, is gratefully acknowledged.

† Research supported in part by the IBM faculty award

[copyright notice will appear here]

not as important when the queries are written by an application program; moreover, we believe that the less verbose RA syntax is more natural and integrates better with imperative languages such as C++ which make extensive use of a rich set of operators. The concise and regular syntax of RA allows modular composition of queries, which is not always possible in SQL.

To appreciate the level of integration of ARARAT with C++, consider the following statement,

```
dbcon << (EMPLOYEE[SALARY] / (DEPTNUM == 3));
```

which sends to the stream `dbcon` an SQL statement that computes the salaries of all employees in the third department, i.e.,

```
select SALARY from EMPLOYEE where DEPTNUM = 3;
```

Moreover, if `q` is a C++ variable storing a query to a database, then `TUPLE_T(q)` is a C++ type which can be used for storing a tuple of the result of this query.

Comparison with Related Work. There are a number of approaches to the problem of integrating SQL with an application language. First, it is possible to process the embedded SQL statements by a dedicated external preprocessor as done in e.g., SQLJ [16], SchemeQL [46] and Embedded SQL for C[1]. Advantages are in preserving flexibility and power of both languages. In contrast, this approach does not support very well dynamic generation of statements— all possible SQL statements must be available to the preprocessor for inspection.

Second, it is possible to use a software library, as done in the SQL DOM system [30] whose structure reflects the constraints which the database scheme imposes, and allows production of only correct SQL statements. However, such systems trade expressive power for usability and require the user to learn a non-trivial library; further, an external tool must be used to generate this library.

Third, in systems such as LINQ [33], the host language compiler is modified to support SQL like syntax. The advantage is the quality of integration, and the minimal learning curve. An important variation of this approach is offered by Cook and Rai's *Safe Query Objects* [11] which rely on the reflective features of OpenJava [43] to achieve such syntactical changes and to encode constraints imposed by the scheme. Concerns with this approach include portability, expressive power, and quality of integration of the foreign syntax with other language features.

Finally, there is the approach of using an external static analyzer that checks that the program only produces correct SQL statements [9, 21, 25]. Again, usability is limited by the necessity of invoking an external tool. Also, unlike the other approaches, this approach does not simplify the engineering work of coding the production of SQL queries. Since the analysis is only partial, it does produce false alarms.

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 [29] system, a database oriented extension of HASKELL. 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 class for each possible fields combination, with a fixed mapping of SQL types to C++ types. Such a class can be used for data retrieval and manipulation.

Admittedly, just like many other systems mentioned above, ARARAT is language specific. 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.

***Environment Requirements of* ARARAT .** Portability and simplicity were primary among our design objectives: ARARAT minimizes the use of external tools, and can be used (at its core) with any standard [41] C++ compliant compiler. For a more advanced features, we rely, as described in Sec. 5.4, on two small extensions: the **`typeof`** pseudo operator and the `__COUNTER__` preprocessor macro. There is also a pre-processor, DB2ARA, which translates a database schema into ARA declarations. This process, which can be thought of as writing type definitions in a header file, is not difficult and can be carried out by hand.

Template Programming. C++ templates were initially designed for generic data structures and algorithms, as done in STL [37]. However, their expressive power was employed by the community in serving other diverse tasks, from dimensional analysis [44], through a solution of the co-variance problem [42], to a framework for aspect oriented programming [49]. Czarnecki and Eisenecker, in their work on application of template programming [13], marked the emergence of a research trend on generative programming.

Template specialization offering conditionals and *template recursive invocation* make this mechanism Turing complete [24]. Also, **`typedef`** definitions inside a class are used in letting one type store what can be thought of as "pointer" to another type. With this mechanism, lists of types, trees of types, and other data structures have become tools of the trade, and there is even a library of data structures and algorithms, Boost Mpl[2] which manipulates types at compile-time much like STL manipulates data at runtime.

Most of the ARARAT's work is carried out by the C++ compiler itself, which is mercilessly exploited, using template programming to ensure that the compiled program will generate at runtime only correct and safe SQL. Our implementation started from the established techniques of template programming [1, 13], and extended these to meet the challenges of realizing a little language. For example, the seduction of the C++ compiler to produce (relatively) meaningful and short compilation errors in response for errors in use of the template library is borrowed from [32, 39].

Thus, this paper shows that the C++ templates mechanism is rich enough to support the complex algorithms required for this task, including e.g., symbol table management, and that SQL type system can be encoded by the hosting type system, and delegated to the compiler to process. The template programming techniques developed here should also be usable (Sec. 6 discusses some the issues involved in embedding SQL rather than RA in C++.)

Outline. Sec. 2 motivates this work by explaining some of the many difficulties raised by the manual process of writing programs that produce SQL statements. The running example presented in Sec. 2 is revisited in Sec. 3, which shows how the same query finds its expression in ARA. Sec. 4 then gives a high level description of the entire ARA language. Sec. 5 describes in greater detail the RAT architecture and the template programming techniques used there. The discussion in Sec. 6 highlights the advantages and limitations of our solution and draws some directions for further research.

2. Preliminaries: Problem Definition

This section motivates our work by demonstrating the intricacies of the existing string based database access from C++.

Tab. 2.1 introduces a database schema that will be used as a running example throughout the paper.

There are three relations in this database, representing employees, departments and divisions. Fields `DEPTNUM` and `DIVNUM` define

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

[2] http://www.boost.org/libs/mpl/doc/index.html, July 2002.

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

Relation	Fields
EMPLOYEE	EMPNUM(**int**), DEPTNUM(**smallint**), FIRST_N(**varchar**), LAST_N(**varchar**), SALARY(**double**), LOCATION(**varchar**)
DEPARTMENT	DEPTNUM(**smallint**), MANAGER(**int**), DESC(**varchar**), DIVNUM(**smallint**)
DIVISION	DIVNUM(**smallint**), MANAGER(**int**), DESC(**varchar**)

(respectively) the ties between the first relation and second, and between the second and the third. Also, field MANAGER in DEPARTMENT and in DIVISION denotes the employee number of the manager of this organizational unit.

Fig. 2.1 shows a simple C++ function that might be used in an application that uses the database described in Tab. 2.1.

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

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

21   return s;
22 }
```

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

Function get_employees receives an 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.

Note that this is only a small example. In a typical database application, there are many similar functions that are often much more complex. Furthermore, every change to the application functionality will necessarily result in change to those functions logic.

Further scrutiny of the body of get_employees reveals more errors and problems:

1. *Misspelled name.* A typo lurks in line 3, by which EMPLOYES is used (instead of EMPLOYEE) for the table name. This typo will go undetected by the compiler.
2. *Syntax error.* If both parameters are non-nulls, there is no space after the AND keyword in line 16, and thus the produced statement is syntactically incorrect.
3. *Type mismatch.* The SQL data type of DEPTNUM column is **smallint**, while the corresponding input parameter is of type **int**. Such errors might result in unexpected behavior. The sources of this kind of problems, what is sometimes called in the literature *impedance mismatch*, an inherent problem of integrating database engines and programming languages. There is no single type system including the programming language and the database language, and thus very little checking can be done on the junction of the two languages.
4. *Security vulnerability.* The code is vulnerable to SQL script injection attacks. A malicious user can construct a value of the first parameter that executes unexpected statement that harms the database.
5. *Code coverage.* Every SQL statement in the code should be executed during testing to validate its correctness. Providing a full code coverage of all execution paths is a demanding task.
6. *Maintenance cost.* The database structure must be kept in sync with the source code. Changes to the database schema might require changes to the SQL queries in the application, which makes the maintenance of the code base harder.

Which of these problems are solved by AraRat? First, name misspelling will result in compile-time error (we do assume that the database scheme was fed correctly into the system). Other syntax errors will be caught even without this assumption. Also, Rat takes care of the correct conversion of C++ literals and values to SQL.

Strings generated by AraRat are immune to injection attacks. Rat defense against injection attack is twofold: (i) For non-string types, RAT and C++ type system prevent such attacks. Consider e.g., the query

```
select * from users where name='$name' and pin=$pin
```

that is vulnerable to the attack of injecting the string "1 or 1=1" (without the quotes) into variable pin. The injected value becomes part of the generated SQL statement,

```
select * from users where name='$name' and pin=1 or 1=1
```

i.e., the filter becomes a tautology. These kinds of attacks are not possible with AraRat, since pin must be an integer. (ii) Rat's protection against injections into string variables, e.g., into variable $name in the above example, is carried out at runtime. As we shall see below (Fig. 4.2), Ara allows user string variables only as part of scalar expressions, used for either making a selection condition, or for defining new field values. Therefore, Rat's runtime can, and indeed does, validate the contents of string variables taking, escaping as necessary all characters that might interfere with the correct structure of the generated SQL statement.

The maintenance cost is minimized by AraRat. A change to the scheme requires a re-run of the DB2Ara tool (or a manual production of its output), but after this is done, mismatches of the generated SQL to the scheme are flagged with compilation errors.

3. Relational Algebra Queries

An Ara 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 executing the query, defining variables for storing its result, and for other purposes.

Just like all C++ objects, query objects have two main properties:

1. *Type.* The type of a query object that represents a certain query encodes in it the scheme of the result of this query. The Rat library computes at compile-time the data type of each tuple in the result. If two distinct queries return the same set of fields, then the two query objects representing these queries will encode in them the same result type.
2. *Content.* The content of a query object, which is computed at runtime by the Rat library, is an abstract encoding of the procedure by which the query might be executed. All query objects have an asSQL() method which translates this abstract encoding into a string with the SQL statement which can be used to execute the query. Also, query objects have a conversion into string operator that invokes asSQL(), so query objects can be used anywhere a **const char** * can be used.

The DB2Ara pre-processor tool generates a *primitive* query object for each of the relations in the input database. The content

of each primitive query object is an encoding of the pseudo-code instruction: "return all fields of the relation". In the running example, header file `employees.h` defines three such primitives: `EMPLOYEE`, `DEPARTMENT` and `DIVISION`. Thus, the C++ expression `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, the RAT library provides a number of functions and overloaded operators. Each of the RA operators has a C++ counterpart. It is thus possible to write RA expressions, almost verbatim, in C++.

3.1 Composing Query Objects

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

```
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(ID);
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 && SALARY < 3.14))
11    // Selection of a tuple subset
12    [
13      FIRST_N, LAST_N,
14      FULL_N(cat(LAST_N,", ", FIRST_N)),
15      ID(EMPNUM)
16    ]
17  ).asSQL();
18  // ... execute the SQL query in s using e.g., ADO.
19  return 0;
20 }
```

Fig. 3.1: Writing a simple RA expression in ARA.

In lines 10–16 of this figure, a compound query object is generated in two steps:

- First (line 10), 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`.

- 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 `ID` (which is just field `EMPNUM` renamed).

  Note that 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 `ID(EMPNUM)` 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
    ((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. 3.2: 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.

```
where DEPTNUM > 3 and SALARY <= 3.14;
```

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

Comment. The exact content of string `s` may be implementation-dependent, yet it is guaranteed to be a syntactically SQL correct statement, whose content accurately reflects the query encoded in the query object.

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

Tab. 3.1: ARA equivalents of relational algebra operators.

Relational Algebra Operator	ARA Operator	ARA Function	SQL equivalent
selection $\sigma_c R$	R/c	`select(`R,c`)` R`.select(`c`)`	**`select`** $\star$ **`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

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 **`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 ()`**.

ARA 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 RA expression into C++ is quite straightforward. Fig. 3.2 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 RAT is "type safe", in the sense that an attempt to generate illegal queries results in a compile-time error. Thus, expressions `q1+q2` and `q1-q2` will 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.

3.2 Storing Query Objects in Variables

A single C++ statement was used in Fig. 3.1 to generate the desired query object. But, as can be seen in this example, as well as in Fig. 3.2, a single C++ expression for generating a complex query objects might be a bit cumbersome. Moreover, there are cases in which a query object *must* be created incrementally.

For the purpose of creating a query object in several steps, RAT makes it possible to record intermediate objects in variables. In this recording, it is important to remember the significance of type: A query object can be assigned to a variable only if the type of this variable represents the same type of results as the query object.

Fig. 3.3 makes use of query variables assignment to recode our motivating example, Fig. 2.1, in RAT.

```
1 char* get_employees(short dept, char* first) {
2    DEF_V(e,EMPLOYEE);

4    if (first != null)
5       e /= (FIRST_N == first);
6    if (dept > 0)
7       e /= (DEPTNUM == dept);
8    return e[FIRST_N,LAST_N].asSQL();
9 }
```

Fig. 3.3: A rewrite of function `get_employees` (Fig. 2.1).

In line 2 we define variable `e` which is initialized to the `EMPLOYEE` primitive query object. This line makes use of the macro `DEF_V`, defined as

```
#define DEF_V(var_name, exp) typeof(exp) var_name = exp
```

to record both type and content of `EMPLOYEE` into variable `e`. Note that the second parameter to the macro is used twice: once for setting the type of the new variable by means of Gnu C++ [17] operator **`typeof`**, and again, to initialize this variable.

Hence, variable `e` can represent queries which return a relation with the same fields as `EMPLOYEE`. Initially, the evaluation procedure stored in this variable is nothing but an abstract encoding of the instruction to take all tuples of `EMPLOYEE`, but lines 4–7 modify it to specify a refined selection condition as dictated by `get_employees`'s parameters.

To complete the generation of the query object, we should evaluate `e[FIRST_N,LAST_N]` in representation of the step of eliminating all fields but `FIRST_N` and `LAST_N`. Obviously, the type of this expression is not the same as the type of `e`. Line 8 records both the type and value of this expression in a temporary variable.

Note that the example used abbreviated assignment operator (**`operator /=`**) to modify the query variable `e`. This what may be thought of as an extension of RA, is legal, since selection does not change the type of the result. Similarly, RAT offers abbreviated assignment operators for union and substraction, so that expressions such as `e1+=e2` or `e2-=e1` are legal whenever the type of result of `e1` and `e2` is the same.

The example uses method `asSQL()` of the query object to get an SQL statement representing the query. Another piece of information that can be obtained from a query object is the tuple type, a class that can store one tuple of the result relation. Given a query object `e`, the following defines an array of pointers to object of `e` tuple type class, ready for storing the result of the query.

```
TUPLE_T(e) **result = new TUPLE_T(e)*[db.result_size()];
```

3.3 An Elaborate Example

Finally, we show a simple program to compute the names of employees which earn more than their managers. (This example was used e.g., in the work on Safe Query Objects [11].) The code could have been written as a single expression of RA. Instead, Fig. 3.4 shows the computation in three stages.

```
1 DEF_F(M_SALARY);

3 char *anomalous() {
4   DEF_V(e, (EMPLOYEE*DEPARTMENT));
5   DEF_V(m,
6         EMPLOYEE[MANAGER(EMPNUM),M_SALARY(SALARY)];
7   return (
8     (e * m)[FIRST_N,LAST_N] / (SALARY > M_SALARY)
9   ).asSQL();
10 }
```

Fig. 3.4: Using ARARAT to find the names of employees who earn more than their managers.

Line 4 first uses relational algebra join to compute the manager number of each employee. Many non-useful fields remain in the result, but this does not pose an efficiency problem, since we only compute an SQL query that is later submitted to optimized execution by the database engine. Then, in line 6 we rename the fields in relation `EMPLOYEE` so that they can be used as descriptives of managers. Finally, line 8 does another join, to determine manager's salary, projection of the desired fields, and then a selection based on the anomalous salary condition.

4. The ARA Little Language

Having seen a number of examples, it is time for a more systematic description. Instead of a software manual, we use a language metaphor to explain how the ARARAT system works: we ask the user to think of the system as an extension of the C++ programming language with a new little language, ARA, designed for composing SQL queries. ARA augments the mathematical formalism of RA with features designed to enhance usability, and for integration with the host programming language. This section first describes the grammar and then highlights some of the semantics of this little language.

The next section describes the techniques we used for realizing the grammar and the semantics within the framework of C++ templates, and hints on how these techniques might be used for realizing other little languages.

4.1 Syntax

ARA extension of C++ is in allowing a C++ programmer to include any number of ARA *definitions* and *statements* within the code. The BNF grammar of definitions is given in Fig. 4.1.

```
Definition ::=DEF_F(Field) | DEF_V(Var,Exp) | DEF_R(Relation,(Scheme))
  Scheme ::=Field/Type [,Scheme]
    Type ::=INT | SMALLINT | BOOL | STRING | ...
```

Fig. 4.1: The grammar of ARA. Part I: Schema Definition.

The first line of the figure indicates that there are three kinds of definitions:

1. *Field definitions* are used to define the field labels space of RA. Each field name must be defined precisely once. For example, the following defines the three fields names used in the last relation of our running example (Tab. 2.1).
   `DEF_F(DIVNUM); DEF_F(MANAGER); DEF_F(DESC);`
   Field names are untyped until bound to a relation.
2. *Relation definitions*, which are similar to the *Data Definition Language* (DDL) statements of SQL, are used for specifying a database schema, by declaring relation names and the list of field-type pairs included in each. For example, the scheme of the last relation of our running example, is specified by the following definition
   `DEF_R(DIVISION, (DIVNUM/SMALLINT,MANAGER/INT,DESC/STRING))`
   (Such definitions are normally generated by the DB2ARA tool but evidently can be easily produced by hand.)

3. *Variable definitions*, made with the help of a DEF_V call, create a new C++ variable initialized to a relational algebra expression Exp. The full syntax of such expressions is given below in Fig. 4.2, but any relation defined by DEF_R is also an expression. The statement

```
DEF_V(e,EMPLOYEE)
```

in Fig. 3.3 thus defines e to the (trivial) relational algebra expression EMPLOYEE. Variable definition encodes in the type of variable the set of accessible fields in the expression.

Fig. 4.2 gives a (partial) syntax of ARA statements.

```
Statement ::=Exp; | Var+=Exp; | Var-=Exp; | Var/=Cond;
      Exp ::=Var | Relation | Exp+Exp | Exp-Exp | Exp*Exp
              | Exp<<Exp | Exp>>Exp | Exp/Cond | Exp[Vocabulary]
     Cond ::=Scalar
   Scalar ::=C++ variable | C++ literal | Field
              | Scalar && Scalar | Scalar || Scalar | !Scalar
              | Scalar + Scalar | Scalar * Scalar | -Scalar
              | Scalar > Scalar | sin(Scalar) | cat(Scalar,Scalar)
Vocabulary ::=FieldOptInit [,Vocabulary]
FieldOptInit ::=Field | Field(Scalar)
```

Terminals include Relation, Field and Var which are C++ identifiers, respectively representing a name of a relation in the data base, a field name, and a variable storing a query object.

Fig. 4.2: The grammar of ARA. Part II: statements.

The most important non-terminal in the grammar is Exp, which denotes an expression of RA obtained by applying any of the RA operators to atomic relations. An Exp can be used from C++ in two main ways: first, such an expression responds to an asSQL() method; second, any such expression can be passed to the TUPLE_T macro, which returns a *PODS*[3] type with all accessible fields in this relation.

An Exp may involve (i) relations of the database, (ii) C++ variables storing other Exps, or (iii) field names. All three must be previously defined.

An ARA statement can be used anywhere a C++ statement is legal. It can be an Exp, or it may modify a C++ variable (defined earlier by a DEF_V) by applying to it the union or substraction operators of RA. Similarly, a statement may apply a selection operator to a variable based on a condition Cond. It is not possible to use the join and projection operators to change a variable in this fashion, since these operators change the list of accessible fields, and hence require also a change to the variable type.

An Exp is composed by applying relational algebra operators, union, substraction, selection, projection and the three varieties of join to atomic expressions. Atomic expressions are either a C++ variable defined by DEF_V or a relation defined by DEF_R. An Exp may appear anywhere a C++ expression may appear, but it is used typically as receiver of an asSQL() message, which translates the expression to SQL.

Cond is a Scalar expression which evaluates to an SQL truth value. The type system of ARA is similar to that of SQL, i.e., a host of primitive scalar types, including Booleans, strings, integers, and no compound types. Scalar expressions of ARA must take one of these types. They are composed from C++ literals, C++ variables (which must be of one of the C++ primitive types or a "**char** *"), or RAT fields. Many logical, arithmetical and builtin functions can be used to build scalar expressions. Only a handful of these are presented in Fig. 4.2.

Finally, note that the projection operation (**operator** []) involves a Vocabulary, which is slightly more general than a simple list of field names. As in SQL, ARA allows the programmer to define and compute new field names in the course of a projection. Accordingly, a Vocabulary is a list of both uninitialized and *initialized* fields. An uninitialized field is simply a field name while an initialized field is a renamed field or more generally, a field initialized by a scalar expression.

[3] Plain Old Data Structure, i.e., a C like **struct**, with **public** data members only.

4.2 Semantics

RAT defines numerous semantical checks on the ARA little language. Failure in these triggers an appropriate C++ compilation error. In particular, RAT applies *symbol table lookups* and *type checking* on every scalar expression.

For example, in a selection e/c expression, RAT makes sure that every field name used in the scalar expression c exists in the symbol table of e; RAT then fetches the type of these fields, and applies full type checking of c, i.e., that the type signature of each operator matches the type of its operands; finally, if c's type is not boolean, then the selection is invalid.

Other checks are shown in Fig. 4.3.

1. In union e1+e2, and in e1-e2, the sets of fields of e1 and e2 must be the same.
2. In e1+e2, e1-e2, e1*e2, e1<<e2 and e1>>e2, if a field is defined in e1 with type τ, then either this field is not defined in e2, or it is defined there with the same type τ.
3. In a selection, e/c, expression c is a properly valid expression of boolean type.
4. There exists an evaluation sequence for a vocabulary, i.e., the initializing expression of any initialized field does not depend, directly or indirectly on the field itself.
5. In using a Vocabulary in a projection it is required that
   (a) all initialized fields are *not* found in the symbol table of the projected relation;
   (b) all uninitialized fields exist in this symbol table;
   (c) If an initializing expression uses a field which does not occur in the vocabulary, then this field exists in the projected relation.

Fig. 4.3: Some semantical checks applied by RAT.

5. A Look Into RAT Internals

In contrast with other embedded languages, the ARA little language is implemented solely with C++ template mechanism, and without modifications to the compiler of the host language, nor with additional pre- or post-processing stages. This section explains how this is done.

In Sec. 5.1, we explain the general technique of representing the executional aspects of a RA expression as a C++ runtime value, without compromising type safety. Sec. 5.2 then discusses some of the main components of the RAT architecture. Sec. 5.3 demonstrates the technique, showing how these components cooperate to achieve compile-time assurance that only boolean expressions are used for selection. Finally, Sec. 5.4 discusses the two compiler extensions which RAT uses.

5.1 Combining Compile-time and Run-time Representations

There is a rather standard encoding of symbolic expressions as types e.g., for symbolic derivation (SEMT [18]) or for emitting efficient code after gathering all operation applicable to a large vector [45]. The implementation of RAT resisted the temptation of employing this encoding as is for representing RA expressions, or boolean and arithmetical expressions used in selection and in defining new fields. Tab. 5.1 compares the compiler architecture (so to speak) of RAT with that of the expression templates library and SEMT.

It is an inherent property of template-based language implementation that the lexical analysis is carried out by host compiler. Similarly, since no changes to the host compiler are allowed, the syntax of the little language is essentially also that of C++, although

Tab. 5.1: Realizing compiler stages with template libraries.

Compiler Stage	Expression Templates / SEMT	RAT
Lexical Analysis	C++ Compiler	C++ Compiler
Parsing	C++ Compiler	C++ Compiler
Type Checking	Degenerate	Template Engine
Code Generation	Template Engine	Program runtime

```
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. 5.1: A re-implementation of function `get_employees` (Fig. 3.3) in which selection is applied after projection.

both expression templates and ARA make extensive use of operator overloading to give a different semantics to the host syntax to match the application needs.

The main objective of expression templates and SEMT is *runtime* efficiency. Accordingly, the type system in these is degenerate, and code is generated at compile-time by the templates engine. Conversely, ARA is designed to maximize compile-time safety, and includes it own type- and semantic-rules. ARA has a non-degenerate type system, and non-trivial semantical rules, which are all applied at the time the host C++ language is compiled. This design leads to the delay of code generation (production of the desired SQL statement) to runtime. An advantage of this delay is that the same code fragment may generate many different statements, depending on runtime circumstances, e.g., use C++ parameters. To make this possible, the structure of a scalar expression is stored as a runtime value. Types (which can be thought of as compile-time values) are used for recording auxiliary essential information, such as the list of symbols which this expression uses. These symbols are bound later (but still at compile-time) to the data types dictionary of the input scheme.

This technique (together with the delayed execution of semantics of query objects) makes it possible to use the same field name, possibly with distinct types, in different tables. But, the main reason we chose this strategy is to allow assignments such as

```
e /= (DEPTNUM == dept);
```

(line 7 of Fig. 3.3), which would have been impossible if the type of `e` represented its evaluation procedure.

Curiously, this technique supports what may seem paradoxical at first sight: a selection based on fields which were projected out, making it possible to rewrite Fig. 3.3 as Fig. 5.1.

Observe that in line 4 we apply a selection criterion which depends on field `DEPTNUM`, which was projected out in line 2. This is possible since the type of each query entity encodes two lists:

1. *Active Fields.* this is a list with names and types of all fields in the tuples computed by evaluating the query; and
2. *Symbol Table.* This includes the list of all fields against which a selection criterion may be applied. In particular, this list includes, in addition to the active fields, fields which were projected out.

Comment. This crisp distinction between runtime and compile-time representation does not apply (in our current implementation of RAT) to scalar expressions. Consequently, it is not possible to define a C++ variable storing a selection condition, and then modify this expression at runtime. Each boolean expression in ARA has its own type.

5.2 Concepts in RAT

A *type concept* [5] (or for short just a concept) is defined by a set of requirements on a C++ type. We say that a type *models* the concept, if the type satisfies these concepts' requirements. Thus, a concept defines a subset of the universe of all possible C++ types. The notation $\mathcal{C}_1 \preceq \mathcal{C}_2$ (concept $\mathcal{C}_2$ *refines* concept $\mathcal{C}_1$) means that every type that models $\mathcal{C}_1$ also models $\mathcal{C}_2$.

Concepts are useful in the description of the set of types that are legible parameters to a template, or may be returned by it. However, since the C++ template mechanism is untyped (in the sense that the suitability of a template to a parameter type is checked at application time), concepts are primarily a documentation aid. (Still, there are advanced techniques [32, 39, 48] for realizing concepts in the language in such a way that they are checked at compile-time.) A language extension to support concepts [22] is a candidate for inclusion in the upcoming revision of the ISO C++ standard.

The RAT architecture uses a variety of concepts for representing the different components of a query, including field names, field lists, conditions etc. Tab. 5.2 summarizes the main such concepts. Comparing this table with the language grammar (Figs 4.1 and 4.2), we see that concepts (roughly) correspond to non-terminals of the grammar.

Tab. 5.2: The main concepts in the RAT architecture.

Concept Name	Purpose	Sample Operations [a]
F Field	Symbolic field name.	F, $F:V$
I Initialization	A symbolic field name, along with an initialization expression, $F \preceq I$.	I, $I:V$ $F(S):I$
V Vocabulary	A set of (possibly initialized) symbolic field names.	V, $I:V$ I, $V:V$
S Scalar	An expression evaluating to a scalar, e.g., string, boolean, integer, obtained by applying arithmetical, comparison and SQL-like functions to fields, literals and variables, $F \preceq S$.	S+$S:S$ S*$S:S$ cat(S,S) : S S>=$S:S$ S || $S:S$
R Relation	An expression in enriched RA evaluating to a relation.	R*$R:R$ R+$R:R$ R-$R:R$ R[V] : R $R/S:R$

[a] The notation $\mathcal{A} \diamond \mathcal{B} : \mathcal{C}$ where $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are concepts and $\diamond$ is a C++ operator means that the library defines a function template overloading the operator $\diamond$, such that the application of $\diamond$ to values in kinds $\mathcal{A}$ and $\mathcal{B}$ returns a value of concept $\mathcal{C}$. This notation is naturally extended to unary operators and to the function call operator.

The most fundamental concept is F, which represents symbolic field names. The vocabulary concept V represents a set of fields (such sets are useful for RA projection). The last cell in the first row of the table means that a C++ expression of the form v_1`,`v_2 (applying `operator ,` to values v_1 and v_2) where v_1 and v_2 belong in F, return a value in V. The type of this returned value represents the set of types of v_1 and v_2. For example, expression `FIRST_N,LAST_N` belongs in V, and it records the set of these two symbols.

Types that model F are singletons. In our running example, the value `FIRST_N` is the unique instance of the type representing the symbol *FIRST_N*. The macro invocation `DEF_F(FULL_N)` (line 6 Fig. 3.1) defines a new type that models F along with its single value. Henceforth, for brevity sake we shall sacrifice accuracy in saying that a value belongs in a certain concept meaning that this value's type models this concept. This convention will prove particulary useful when talking about singleton types. Thus, we say that this macro invocation defines the value `FULL_N` in concept F.

The concept I represents initialized fields, necessary for representing expressions such as

```
FULL_N(cat(LAST_N, ", ", FIRST_N))
```  (5.1)

(line 14 in Fig. 3.1). A type modeling I has two components: a field name and an abstract representation of the initializing expression.

Concept S represents scalar expressions used in an initialization expression and in selection, e.g., `cat(LAST_N, ", ", FIRST_N)` is

in S. In writing $F(S) : I$ in the table we indicate that expression (5.1) which applies the overloaded function call operator of field `FULL_N` to `cat(LAST_N, ", ", FIRST_N)` is in I.

Since $F \preceq S$ we have that the function call operator can be used in particular to do RA-renaming, writing, e.g., `ID(EMPNUM)` (line 15 in this figure).

Another instance of S is the expression, showing in line 10, `(DEPTNUM > 3 && SALARY < 3.14)`. Note again that scalar expressions may involve literals.

Concept R is used for representing RA expressions. The rightmost cell in the last row of the table specifies the semantics of union, substraction, cross product, selection, and projection. We can see that RAT enriches the semantics of RA. For example, vocabularies may include initialized fields. The initialization sequence of such fields specifies the process by which this field is computed from other fields during projection.

5.3 Type Safety of Selection and Projection Expressions

Now that the main concepts of the RAT architecture were enumerated, we turn to describing how these concepts are used both in compile-time and in run-time to realize the integration of RA into C++. The description is as high-level as possible, although some of the technical details do pop out.

5.3.1 Managing Scalar Expressions

At runtime, a scalar expression is represented as a value of type `S_TREE`. Type `S_TREE` is the root of a type hierarchy that does a rather standard text book [40, pages 279–290] implementation of an expression tree, with classes such as `S_BINARY` (for binary operators), and `S_UNARY` (for unary operators), to represent internal nodes. Leaves of the tree belong to one of two classes: (i) `S_LIT`, which store the value of C++ values participating in the expression, and (ii) `S_FIELD` representing a name of one of the fields in the underlying RA.

The representation is untyped in the sense that the actual evaluation of each node in the tree may return any of the supported types of scalar expressions, including booleans, strings and integers. In difference with standard representation of expression tree, the nodes of this tree do not have an evaluation function. Instead, class `S_TREE` has a pure virtual function `char *asSQL()`. This function is implemented in the inheriting classes, to return the SQL representation of the expression by a recursive traversal of the subtree.

Thus, the evaluation of an `S_TREE` is carried out by translating it to SQL. This translation is always performed in the context of translating a RA expression, which contains the scalar expression, to SQL. At the time of the translation, `S_LIT` leaves are printed as SQL literals, while `S_FIELD` are printed as SQL field names.

Fig. 5.2 shows what a class `S` modeling concept S looks like.

```
1 class S {public:
2   const S_TREE *t; // Expression tree of S
3   typedef ... TYPS; // Compile−time representation of t
4   typedef ... FLDS; // List of fields used in t
5     ... }
```

Fig. 5.2: The main ingredients of a type modeling concept S.

As seen in the figure, each such type has a data member named `t` which stores the actual expression to be evaluated at runtime.

For the purpose of compile-time type checking of the scalar expression, when using it for projection or selection, each such type has two compile-time properties, realized by a **`typedef`**:

1. property `TYPS` is a compile-time representation of the content of `t`, i.e., `TYPS` is tree-structured type, with the same topology as `t`. However, instead of literal values and addresses of variables, which are difficult to represent in the type system, this compile-time representation stores just their types.
2. property `FLDS` is a type which represents the list of field names that take part in this scalar expression. Each node in this list is a type modeling concept F.

A number of function templates (including many that overload the standard operators) are defined in RAT. All these functions generate types modeling S. Each concrete template function obtained by instantiating these templates computes the value of data member `t` in the return value, and generates the **`typedef`**s `TYPS` and `FLDS` of the type of the result.

More specifically, if `S1` and `S2` are types that model S, then the return type of **`operator`** `+(const &S1, const &S2)` is a type `S` modeling S such that (i) `S::FLDS` is the merge of `S1::FLDS` and `S2::FLDS`, and (ii) `S::TYPS` is a type tree, rooted a node representing and addition with two type subtrees `S1::TYPS` and `S2::TYPS`.

The actual value returned by a call **`operator`** `+(x, y)` (where the class of `x` is `S1` and that of `y` is `S2`), is such that its `t` field is of class `S_PLUS` and has two subtrees `x.t` and `y.t`.

Note that types that model concept S have compile-time properties that describe the expression. Therefore, these types cannot be reassigned with a different expression.

5.3.2 Managing Relational Algebra Expressions

As already mentioned, types that model concept R have a runtime encoding of the procedure for evaluating the query and two main compile-time properties: an encoding of the scheme of the result of a query, and a list of fields which can be used in a selection criterion.

When a selection operation on relation `r` of type `R` with scalar expression `s` of type `S` is encountered, `S` is bound to `R`. Steps in this binding include:

1. A check that all fields in `S::FLDS` are present in `R`.
2. Binding each field of `S` to its type in `R` to analyze the types of `S::TYPS`.
3. If there is a type mismatch between an operator and its operands or if the result type is non boolean, a compilation error is issued.
4. Integration of the content of `s.t` with `r`. This integration affects only the runtime value of `r` and therefore reassigning the new value into the same variable `r` is possible.

A projection operation is defined on relation `r` of type `R` and field list `v` of type `V` modeling concept V. There are two kinds of elements in `v`: uninitialized fields, each modeling concept F, and initialized fields, each modeling concept I. RAT verifies that all the uninitialized fields are present in `R` and all initialized fields are absent of `R`. RAT also verifies that the initialized fields can be calculated when bound to `R` using the same algorithm used in the selection operation. In addition, the compile-time encoding of the result scheme and the symbol table of `R` are updated, which means that the result of a projection operation is an object of a new type which cannot be assigned to `r`.

5.4 Compiler Extensions and the `TUPLE_T` macro

Extracting a variable's type. An incremental generation of query object requires that intermediate objects are stored in variables. It is necessary to record both the type and the content of these objects. ARARAT extracts the type of a query object with the non-standard **`typeof`** pseudo operator. Thus, macro `DEF_V` in Sec. 3 creates a new variable `var_name`, which has the type of `exp`, and initializes this variable with the content of `exp`.

The **`typeof`** operator is a pseudo operator, since instead of returning a value, it returns the *type* of the argument, which can be used anywhere a type is used. Like **`sizeof`**, this operator is evaluated at compile-time. This operator is found in e.g., all Gnu implementa-

tions [17] of the language; its significance was also recognized by the C++ committee, which is considering a similar mechanism for querying the type of an expression, namely `decltype` operator, as a standard language extension [4].

Without **`typeof`**, ARARAT can still produce query objects, but in order to store these in variables the user must manually define appropriate types. The compiler still checks that these type definitions are correct and consistent with the query objects.

Type ordering. In RA, the order of fields in a relation has no meaning, i.e., two schemes represent the same relation if they consist of the same set of fields. The Boost Mpl library makes it possible to test the equality of two such sets without resorting to sorting. However, we still require a method to sort such sets, for the purpose of the implementation of the `TUPLE_T` macro. Consider for example the statement

```
TUPLE_T(EMPLOYEE * DEPARTMENT) emp1;
TUPLE_T(DEPARTMENT * EMPLOYEE) emp2;
```

which defines variables `emp1` and `emp2` to be of a PODS whose field names and types are the union of the field lists of `EMPLOYEE` and `DEPARTMENT`. To generate such a field, RAT uses a recursive template call to create an inheritance chain, where each class in this chain defines a field name. We need this chain to be of a predetermined order, so that types of `emp1` and `emp2` are the same.

To this end, an ordering relation must be placed on types. In our implementation, this order is realized by a unique integral value associated with every field name. The identifier is generated using another C++ language extension, the `__COUNTER__` macro. This macro is a compile-time counter, evaluates to an integer which is incrementing in each use. It is supported by Microsoft compiler [35], and is scheduled to be included in version 4.3 of g++ (pending approval on the GCC steering committee [5]). Using this macro ensures that each field has a constant identifier and thus every relation has a unique representation.

Without `__COUNTER__`, ARARAT must resort to the standard `__LINE__` macro. The limitation placed on the user is that all fields are defined in different lines in the same source file.

6. Discussion and Further Research

The ARARAT system demonstrates a seamless integration of RA language with C++, which can be used to generate safe SQL queries (using method `asSQL()`). Also, ARARAT makes it possible to generate a PODS type for storing query results. The challenges in the implementation were in the restriction on use of external tools, without introducing cumbersome or long syntax.

Admittedly, the compilation time of the program in Fig. 2.1 is shorter than that of Fig. 5.1. The compilation of the program in Fig. 2.1 took 1.04 seconds while the program in Fig. 5.1, which uses the RAT library that consists of about 3000 lines of C++ code, compiled in 1.26 seconds [6]. We believe that the compilation time is not a main consideration in this sort of applications. A comprehensive benchmark is required for comparing the runtime performance of the two alternatives over a wide range of queries. The benchmark should measure both the construction and execution time of queries. We estimate that the construction time is negligible, and therefore generating queries using ARARAT does not impose a significant performance penalty.

The current implementation support of dynamic queries is limited to modifications of a query object by applying selection, union and substraction to it. It is mundane to add support to dynamically created conditions, allowing thus to define a selection condition with code such as

```
DEF_DYNAMIC_COND(c, EMPLOYEE, TRUE);
if (dept > 0);
   c &= (DEPTNUM == dept);
if (first != null)
   c &= (FIRST_N == first)
...
```

where `DEF_DYNAMIC_COND(c, EMPLOYEE, TRUE);` encodes `EMPLOYEE`'s symbol table into `c`. Also easy in principle is the definition of "prepared SQL statements", by storing memory addresses of C++ variables instead of actual content. The actual value of these variables is retrieved whenever the query is translated to SQL. Moreover, as hinted in brief in Sec. 5.4, RAT can easily generate function members for these queries which would allow dynamic changes of the parameters of a prepared statement.

In Sec. 5 we described non-trivial implementation techniques, by which both compile-time and runtime values are used for realization of the language extension. These techniques, and the experience of managing symbol tables, type systems, and semantical checks with template programming, can be used in principle for introducing many other little languages to C++. Further research will probably examine the possibility of doing that, and in particular in the context of a little language for defining XML data. Presumably, the same ARA representation can be used to generate not only SQL, but also directives for other database systems.

Another prime candidate for a little language to be added thus in C++ is the SQL language itself. Indeed, a user preferring explicit function names, as in Fig. 3.2(b) and Fig. 3.2(c) will find the resulting code similar to SQL. We contemplate extending this to support a more SQL-like syntax as done in e.g., LINQ. Still, it should be remembered that, as evident by the LINQ experience, support of even the **`select`** statement can be only partial, mainly because the syntax is too foreign to that of the host language. This is the reason we believe that the advantage of building upon user's familiarity with SQL is not as forceful as it may appear.

On the other hand, it should be clear how to extend ARA to support more features offered by the **`select`** statement, without imposing an SQL syntax. For example, we can easily extend the ARA syntax to support sorting and limits features of **`select`**, by adding e.g., the following rules to Fig. 4.2.

Exp ::= Exp`.asort(`Field`)` | Exp`.dsort(`Field`)` | Exp`.limit(`Integer`)`

The addition of support for **`group by`** clause is more of a challenge, since it requires a type system which allows non-scalar fields, i.e., fields containing relations.

Our work concentrated on selection and queries since these are the most common database operations. We demonstrated how these queries can be generated in a safe manner, and showed how RAT can define a receiver data type. The extension to support the generation of other statements in the Data Manipulation sub-Language (DML) of SQL does not pose the same challenges as those of implementing queries.

We note that ARARAT does not take responsibility on the execution of statements, although it is possible to use it to define the data types which take part in the actual execution. We leave it to future research to integrate the *execution* of queries and other DML statements with C++. This research should strive for a smooth integration of the execution with STL. For example, an **`insert`** statement should be able to receive an STL container of the items to insert. Interesting, challenging and important in this context is the issue of integration of database error handling, transactions, and locking with the host language.

A fascinating direction is the application of ARA to C++ own data structures instead of external databases. Perhaps the best in-

[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/

[5] Ian Lance Taylor, private communication, March 2007

[6] On a 2.13GHz Intel(R), Pentium(R) M processor with 2 GB of RAM.

tegration of databases with C++ is achieved by mapping STL data structures to persistent store.

Finally, we note that in a sense, work on integration of SQL with other languages can be viewed as part of the generative programming line of research [6,19,20,27,36]. It is likely that other lessons of this subdiscipline can benefit the community in its struggle with the problem at hand. For example, it may be possible to employ the work on *certifiable program generation* [14] to prove the correctness of RAT.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

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

[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] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[6] D. S. Batory, C. Consel, and W. Taha, eds. *Proc. of the 1st Conf. on Generative Prog. and Component Eng.*, vol. 2487 of *LNCS*. Springer, 2002.

[7] J. Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

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

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

[10] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[11] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE'05*.

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

[13] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[14] E. Denney and B. Fischer. Certifiable program generation. In *GPCE'05*.

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

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

[17] R. M. S. *et al. Using GCC: The GNU Compiler Collection Reference Manual for GCC 3.3.1*. Gnu Press, 2003.

[18] J. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *USENIX C++'98*.

[19] R. Glück and M. R. Lowry, eds. *Proc. of the 4th Conf. on Generative Prog. and Component Eng.*, vol. 3676 of *LNCS*. Springer, 2005.

[20] R. Glück and M. R. Lowry, eds. *Proc. of the 5th Conf. on Generative Prog. and Component Eng.*, vol. 3676 of *LNCS*. Springer, 2006.

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

[22] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: First-class language support for generic programming in C++. In *OOPSLA'06*.

[23] J. R. Groff and P. N. Weinberg. *SQL, the complete reference*. Osborne/McGraw-Hill, 1999.

[24] Z. Gutterman. Symbolic pre-computation for numerical applications. Master's thesis, Technion, 2004.

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

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

[27] G. Karsai and E. Visser, eds. *Proc. of the 3rd Conf. on Generative Prog. and Component Eng.*, vol. 3286 of *LNCS*. Springer, 2004.

[28] G. Koch and K. Loney. *Oracle: The Complete Reference: Electronic Edition*. Osborne, 1997.

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

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

[31] B. McGehee. *Using Microsoft SQL Server 7.0*. Que, 1999.

[32] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *Workshop on C++ Template Programming (Erfurt, Germany, 2000)*.

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

[34] R. B. Melnyk and P. C. Zikopoulos. *DB2: The Complete Reference*. McGraw-Hill Companies, 2001.

[35] C. Pappas and W. H. Murray. *Visual C++.Net: The Complete Reference*. McGraw-Hill Companies, 2002.

[36] F. Pfenning and Y. Smaragdakis, eds. *Proc. of the 2nd Conf. on Generative Prog. and Component Eng.*, vol. 2830 of *LNCS*. Springer, 2003.

[37] P. J. Plauger. The standard template library. *C/C++ Users J.*, 13(12):10–20, 1995.

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

[39] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Workshop on C++ Template Programming (Erfurt, Germany, 2000)*.

[40] P. Smith. *Applied Data Structures with C++*. Jones & Bartlett, 2004.

[41] A. Stevens and C. Walnum. *Standard C++ Bible*. Wiley, 2000.

[42] V. Surazhsky and J. Y. Gil. Type-safe covariance in C++. In *SAC'04*.

[43] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for Java. In *OOPSLA'99 Workshop*.

[44] Z. D. Umrigar. Fully static dimensional analysis with C++. *SIGPLAN Not.*, 29(9):135–139, 1994.

[45] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

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

[47] M. Widenius and D. Axmark. *MySQL Reference Manual*. O'Reilly, 2004.

[48] J. Willcock, J. G. Siek, and A. Lumsdaine. Caramel: A concept representation system for generic programming. In *Workshop on C++ Template Programming (Tampa, Florida, 2001)*.

[49] Z. Yao, Q. long Zheng, and G.-L. Chen. AOP++: A generic aspect-oriented programming framework in C++. In *GPCE'05*.