Simple and Safe SQL Queries with C++ Templates

Joseph (Yossi) Gil  *   Keren Lenz
Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
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, carried out by our template library, assures, at compile-time, the correctness and safety of the generated SQL strings. That is to say that all SQL statements constructed by the system are syntactically correct, legal with respect to the database schema, and immune to injection attacks.

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

The system provides also initial support for the task of defining C++ data structures required for storing the results returned by database queries. An optional pre-processor can be used to define the database scheme to the C++ program.

Categories and Subject Descriptors D.1.2 [Programming Techniques]: Automatic Programming; D.3.3 [Programming Languages]: Language Constructs and Features; F.3.3 [Logics and Meanings of Programs]; Studies of Program Constructs

General Terms Design, Languages, Reliability, Security

Keywords C++, template programming, embedded languages, domain specific languages, databases, structural type equivalence, relational algebra

* Research supported in part by the IBM faculty award

1. Introduction
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 [39], a persistent (extended) version [2] of C, integration [12] of databases into SMALLTALK, the XJ [26] system integrating XML with JAVA, and many more.

This problem is important since all but the trivial software applications make use of a database system, and invest development resources on the recalcitrant problem of accessing the database from the high level language in which the application is written. The difficulty is that the interaction with a database is by the application constructing strings—statements in the database engine control language, which are then sent for execution to the database engine. 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.

In this work, we concentrate on C++ and its integration 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 [29], Microsoft SQL Server [32], MySQL [48], DB2 [35], and many more. Humans may find many advantages in SQL, including readability, well-defined expressive syntax, declarative semantics, flexibility, etc. But, for programs generating SQL statements, 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 logic of producing correct SQL strings may be complex, 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. These difficulties in producing SQL at runtime are discussed in brief below in Section 2) and in great detail in the literature (see, e.g., [8, 15]).

ARARAT, the system presented here, 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++.

There are two components to the system: a little language, henceforth called ARA, representing augmented relational algebra 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 relational algebra metaphor, rather than directly on SQL, (ii) tight integration with the host language using...
template programming, with minimal reliance on external tools. Some readers may appreciate a degree of elegance in overloading C++ operators.

The main contributions of this paper are in techniques for extending the 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 implementation of a 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. We are intrigued by the question of whether generics can be made powerful enough to support tasks such as symbol table lookup, without achieving full Turing completeness, a feature whose presence in C++ makes it undecidable to determine whether the compiler halts on a given input [24].

Use of Relational Algebra. Historically, SQL emerged from the seminal work of Codd [10] on relational algebra. In a sense, SQL makes it possible to encode relational algebra expressions in a syntax which is more readable to humans, and closer to natural language. These advantages are not as important when the queries are written by an application program; moreover, we believe that the less verbose relational algebra 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 relational algebra allows modular composition of queries, which is not possible in SQL.

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

```cpp
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.,

```sql
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 preprocesser as done in e.g., SQIL [16], SchemeQL [47] and Embedded SQL for C++. Advantages are in preserving flexibility and power of both languages. In contrast, this approach does not very well support 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 [31] whose structure reflects the constraints which the database scheme imposes, and allows production of correct SQL statements only. 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 [34] and the work of Van Wyk, Krishnan, Lijesh, Bodin and Johnson [50] 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 [44] 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: It is never possible to embed in full a foreign syntax in the host language, the result being that SQL is changed in many subtle and sometimes not so subtle ways, whereby generating a yet new SQL variant. For example, the following LINQ code snippet (drawn from Microsoft documentation)

```csharp
customers.Select(c => new {
    c.Name,
    TotalOrders = c.Orders.
        Where(o => o.OrderDate.Year == year).
        Sum(o => o.Total)
});
```

despite using SQL vocabulary (keywords `select` and `where` and function `sum`) and functions, requires some human pondering before it can be mentally expressed as an SQL statement.

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 [30] 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. Also, unlike LINQ and many other systems, ARARAT allows staged construction of query objects—that is, one can construct a query object, and then refine the query, without actually executing it (see Figure 3.3 below for an example).

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. We further admit that unlike LINQ, ARARAT is not industrial-strength, e.g., addition of user-defined functions to the core library is not as smooth as it ought to be, the support of vendor specific SQL dialects is not optimized, update operations are left for further research, etc.

We should also mention less systematic efforts in the form of C++ libraries, such as Ultimate++, Terimber, Postgress C library, IBPP, CQL, and SourcePro©-DB which provide a large set of functions that can be used for more reliable productions of SQL. Such libraries suffer from problems similar to SQL DOM. As explained above, ARARAT is implemented as a library, but its
learning curve is smaller thanks to the underlying theory and to the a high degree of language integration it offers.

**Environment Requirements of ARA\textsc{rat}**. Portability and simplicity were primary among our design objectives: ARA\textsc{rat} minimizes the use of external tools, and can be used (at its core) with any standard [42] C++ compliant compiler. For a more advanced features, we rely, as described in Section 5.4, on two small extensions: the typedef pseudo operator and the \_\_prepro\_\_ preprocessor macro. There is also a pre-processor, DB2\textsc{ara}, 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 [38]. However, their expressive power was soon employed by the community in serving other diverse tasks, from dimensional analysis [45], through a solution of the co-variation problem [43], to a framework for aspect oriented programming [51]. Czamecki 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 [8, 9] which manipulates types at compile-time much like STL manipulates data at runtime.

Most of the ARA\textsc{rat}’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 [33, 40].

This paper shows that the C++ templates mechanism is rich enough to support the complex algorithms required for this task. Note that there are several previous examples of embedding a little language in C++, including e.g., the Boost Spirit library [10] in which C++ is extended to include BNF syntax, while the compiler, as part of the compilation process generates an LL parser for the given syntax. Our system differs from this and (other and such examples) in that the type system of Boost Spirit is degenerate: the BNF symbols belong to one single “type”, in the sense that they may all take part in any production. In contrast, ARA\textsc{rat} features a non-trivial type system, allowing two queries to be combined only if they are compatible.

**Outline**. Section 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 Section 2 is revisited in Section 3, which shows how the same query finds its expression in ARA. Section 4 then gives a high level description of the entire ARA language. Section 5 describes in greater detail the \_\_R\_\_A\_\_R\_\_ architecture and the template programming techniques used there. The discussion in Section 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++.

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

<table>
<thead>
<tr>
<th>Relation</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE</td>
<td>ID(int), DEPTNUM(smallint),</td>
</tr>
<tr>
<td>FIRST_N(varchar), LAST_N(varchar),</td>
<td></td>
</tr>
<tr>
<td>SALARY(double), LOCATION(varchar)</td>
<td></td>
</tr>
<tr>
<td>DEPARTMENT</td>
<td>ID(smallint), MANAGER(int),</td>
</tr>
<tr>
<td>DESC(varchar), DIVNUM(smallint)</td>
<td></td>
</tr>
<tr>
<td>DIVISION</td>
<td>DIVNUM(smallint), CID(varchar)</td>
</tr>
</tbody>
</table>

There are three relations in this database, representing employees, departments, and divisions. Field DEPTNUM is used for identifying the department of an employee, i.e., typical join operation of EMPLOYEE and DEPARTMENT will involve renaming of field ID in DEPARTMENT to DEPTNUM. For simplicity, we used field DIVNUM which defines a direct (i.e., no renaming required) tie between the second relation and third. Also, field MANAGER in DEPARTMENT denotes the employee number of the manager of this organizational unit.

Observe that field ID has type int in EMPLOYEE and type smallint in DEPARTMENT. ARA\textsc{rat} type manager supports this.

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

```cpp
1 const char* get_employees(int dept, char* first) {
2    bool first_cond = true;
3    strings s = "new string";
4    "SELECT FIRST_N, LAST_N FROM EMPLOYES "
5    ;
6    if (dept > 0) | # valid department number
7    s.append( "WHERE DEPTNUM = ' ");
8    s.append(itoa(dept));
9    s.append("'"));
10   if (first == null)
11     return s.c_str();
12   if (first == null)
13     return s.c_str();
14   if (first == null)
15     return s.c_str();
16   if (first == null)
17     return s.c_str();
18   else
19     s.append("AND ");
20     s.append("FIRST_N = ' ");
21     s.append("'"));
22     s.append("'"));
23     return s.c_str();
24  }
```

**Figure 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\_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 4, by which EMPLOYEES 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 18, 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

```cpp
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,

```cpp
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 (Figure 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

```sql
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 relational algebra operators has a C++ counterpart. It is thus possible to write expressions of relational algebra, almost verbatim, in C++.

### 3.1 Composing Query Objects

Figure 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.

```cpp
#include "rat" // Global RAT declarations and macros
#include "employees.h"

// Primitive query objects and scheme of the EMPLOYEE database

DEF_P(FULL_N); DEF_F(EID);

// Define field names which were not defined in the input scheme
int main(int argc, char* argv[]) {
  const char * s = 10
  (EMPLOYEE / (DEPTNUM > 3 && SALARY < 1E5))
  // Selection of a tuple subset
  [12
  [13
  FIRST_N, LAST_N,
  FULL_N(cat(LAST_N,",",FIRST_N)),
  EID(ID)
  ]
  asSQL();
  // ... execute the SQL query in s using e.g., ADO.
  return 0;
}
```

**Figure 3.1: Writing a simple relational algebra expression in ARA.**

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

- **First (line 10), expression**
  
  ```cpp
  EMPLOYEE / (DEPTNUM > 3 && SALARY < 1E5)
  ```

  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 $100,000.
The syntax is straightforward: the selection criterion is written as a C++ boolean expression, and \texttt{operator /} is used for applying this criterion to \texttt{EMPLOYEE}.

- In lines 12–16 an array access operation, i.e., \texttt{operator []}, is employed to project these tuples into a relation schema consisting of four fields: \texttt{FIRST\_N}, \texttt{LAST\_N}, \texttt{FULL\_N} (computed from \texttt{FIRST\_N} and \texttt{LAST\_N}), and \texttt{EID} (which is just field \texttt{ID} renamed).

Note that expression \texttt{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 \texttt{FULL\_N} with the result of this computation. Similarly, expression \texttt{EID(\texttt{ID})} uses this operator for field renaming.

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

\begin{verbatim}
select FIRST\_N, LAST\_N,
concat(LAST\_N, "\", FIRST\_N ) as FULL\_N,
EID as EID
from EMPLOYEE where DEPT\_NUM > 3 and SALARY <= 100000;
\end{verbatim}

This statement is assigned, as a string, to variable \texttt{e}.

\textbf{Comment}. The exact content of string \texttt{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. Table 3.1 summarizes the ARA equivalents of the main operators of relational algebra.

\textbf{Table 3.1: ARA equivalents of relational algebra operators.}

<table>
<thead>
<tr>
<th>Relational Algebra Operator</th>
<th>ARA Operator</th>
<th>ARA Function</th>
<th>SQL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection (\sigma_{R}R)</td>
<td>\texttt{select}[\texttt{R},\texttt{c}]</td>
<td>\texttt{select * from \texttt{R}} where \texttt{c}</td>
<td></td>
</tr>
<tr>
<td>projection (\pi_{f_{1},f_{2}}R)</td>
<td>\texttt{select}[\texttt{f_{1}},\texttt{f_{2}}]</td>
<td>\texttt{select * from \texttt{R}} R</td>
<td></td>
</tr>
<tr>
<td>difference (R_{1}\setminus R_{2})</td>
<td>\texttt{subtract}(\texttt{R_{1}}, \texttt{R_{2}})</td>
<td>\texttt{R_{1} - R_{2}}</td>
<td></td>
</tr>
<tr>
<td>natural join (R_{1}\bowtie R_{2})</td>
<td>\texttt{union}(\texttt{R_{1}}, \texttt{R_{2}}), \texttt{R_{1}.join(\texttt{R_{2}})}</td>
<td>\texttt{R_{1} \bowtie R_{2}}</td>
<td></td>
</tr>
<tr>
<td>left join (R_{1}=\bowtie R_{2})</td>
<td>\texttt{left_join}(\texttt{R_{1}}, \texttt{R_{2}})</td>
<td>\texttt{R_{1}.left_join(\texttt{R_{2}})}</td>
<td></td>
</tr>
<tr>
<td>right join (R_{1}\bowtie R_{2})</td>
<td>\texttt{right_join}(\texttt{R_{1}}, \texttt{R_{2}})</td>
<td>\texttt{R_{1}.right_join(\texttt{R_{2}})}</td>
<td></td>
</tr>
<tr>
<td>rename (\rho_{a/b}R)</td>
<td>\texttt{rename}(\texttt{a/b})</td>
<td>\texttt{a as b}</td>
<td></td>
</tr>
</tbody>
</table>

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 \texttt{C++} (overloaded) operator: selection in relational algebra is represented by \texttt{operator \_\_select} in ARA, projection by \texttt{operator \_\_select} in ARA, union by \texttt{operator \_\_union}, difference by \texttt{operator \_\_select}, natural join by \texttt{operator \_\_union}, left join by \texttt{operator \_\_left_join}, and renaming by the function call operator \texttt{operator \_\_rename}.

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 relational algebra expression into C++ is quite straightforward. Figure 3.2 shows how a query object for finding the managers of departments of the divisions in the city of Haifa can be generated using overloaded operators, global functions and member functions.

\begin{Verbatim}
\begin{verbatim}
\#define DEF_V(x, EMPLOYEE);
const char* get_employees(short dept, char* first) {
  if (first != null) { // (a) Operator overloading version
    cout << "(";
    for (short i = 0; i < dept; i++) {
      if (i > 0) cout << ", ";
      cout << DEF_V(\texttt{\_\_select}(\texttt{DEPARTMENT}),);//, EMPLOYEE))
    };
    cout << ");"
  } else { // (b) Global functions version
    for (short i = 0; i < dept; i++) {
      if (i > 0) cout << ", ";
      cout << DEF_V(\texttt{\_\_select}(\texttt{DEPARTMENT}),);//, EMPLOYEE))
    };
    cout << ");"
  }
}
\end{verbatim}
\end{Verbatim}

Figure 3.2: Three alternatives C++ expressions to compute a query object that, when evaluated, finds the managers of department in divisions located in the city of Haifa: using (a) overloaded operators (b) global functions, and (c) member functions.

The composition of query objects with RA is "type safe", in the sense that an attempt to generate illegal queries results in a compile-time error. Thus, expressions \texttt{q1\_q2} and \texttt{q1\_q2} will fail to compile unless \texttt{q1} and \texttt{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.

\subsection{Storing Query Objects in Variables}

A single C++ statement was used in Figure 3.1 to generate the desired query object. But, as can be seen in this example, as well as in Figure 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, RA 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.

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

\begin{Verbatim}
\begin{verbatim}
int const char* get_employees(short dept, char* first) {
  DEF_V(e, EMPLOYEE);
  if (first != null) {
    e = (FIRST\_N == first);
    if (dept > 0) {
      e = (DEPT\_NUM == dept);
    } return e[FIRST\_N, LAST\_N].asSQL();
  }
}
\end{verbatim}
\end{Verbatim}

Figure 3.3: A rewrite of function get_employees (Figure 2.1).

In line 2 we define variable \texttt{e} which is initialized to the \texttt{EMPLOYEE} primitive query object. This line makes use of the macro \texttt{DEF} defined as \texttt{#define DEF_V(var_name, exp) typeof(exp) var_name = exp} to record both type and content of \texttt{EMPLOYEE} into variable \texttt{e}. Note that the second parameter to the macro is used twice: once for setting the type of the new variable by means of \texttt{Gnu C++ [17]} operator \texttt{typeof}, and again, to initialize this variable.

Hence, variable \texttt{e} can represent queries which return a relation with the same fields as \texttt{EMPLOYEE}. Initially, the evaluation procedure stored in this variable is nothing but an abstract encoding of the instruction to take all tuples of \texttt{EMPLOYEE}, but lines 3–6 modify it to specify a refined selection condition as dictated by \texttt{get_employees}'s parameters.

To complete the generation of the query object, we should evaluate \texttt{e[FIRST\_N, LAST\_N]} in representation of the step of eliminating all fields but \texttt{FIRST\_N} and \texttt{LAST\_N}. Obviously, the type of this
expression is not the same as the type of \( e \). Line 7 records both the type and value of this expression in a temporary variable.

Note that the example used abbreviated assignment 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, RA\( T \) offers abbreviated assignment operators for union and subtraction, so that expressions such as \( e_1 - e_2 \) or \( e_2 - e_1 \) are legal whenever the type of result of \( e_1 \) and \( e_2 \) is the same.

The example uses method \( \text{asSQL()} \) 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 \( o \), the following defines an array of pointers to object of \( e \) tuple type class, ready for storing the result of the query.

```cpp
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 who 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 relational algebra. Instead, Figure 3.4 shows the computation in three stages.

```cpp
def_F(M_SALARY);

const char* anomaly() { return (e * m)[FIRST_N,LAST_N] / (SALARY > M_SALARY);
}
```

Figure 3.4: Using ARA\( T \) to find the names of employees who earn more than their managers.

Line 5 demonstrates how renaming is used to prepare the field names in a relation for a join operation: field \( ID \) in DEPARTMENT is renamed to DEPTNUM so that the subsequent join operation will yield the manager number of each employee. (Note that EMPLOYEE \( \times \) DEPARTMENT[DEPTNUM,ID,MANAGER]) demonstrates how the common SQL phrase “\( T_1 \) join \( T_2 \) on \( T_1.f_1 = T_2.f_2 \)” is written in ARA\( T \).

Then, in line 7 we rename the fields in relation EMPLOYEE so that they can be used as descriptives of managers. Finally, line 9 does another join, to determine manager’s salary, projection of the desired fields, and then a selection based on the anomalous salary condition.

Observe that many non-useful fields are live in intermediate steps of the computation of Figure 3.4. No efficiency concern is to be made here: unlike LINQ, the computation is a query, rather that on its result. Only the final results is submitted to optimized execution by the database engine.

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 ARA\( T \) 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 relational algebra 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 Figure 4.1.

```
Definition ::= DEF_F(Field) | DEF_V(Var, Exp) | DEF_R(Relation, (Scheme))
Scheme ::= Field, Type [], Scheme
```

Figure 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 relational algebra. Each field name must be defined precisely once. For example, the following defines the two fields names used in the last relation of our running example (Table 2.1).

```
DEF_F(DIVNUM); DEF_F(CITY);
```

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(R, (DIVISION, [DIVNUM,SMALLINT,CITY,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 \( e \). The full syntax of such expressions is given below in Figure 4.2, but any relation defined by DEF\_R is also an expression. The statement

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

in Figure 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.

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

```
Statement ::= Exp; Var+=Exp; Var-=Exp; Var/=Cond;
Cond ::= Scalar; C++ variable | C++ literal | Field
```

Figure 4.2: The grammar of ARA, Part II: statements.

"The most important non-terminal in the grammar is \( e \) which denotes an expression of relational algebra obtained by applying any of the relational algebra operators to atomic relations. An Exp can be used from C++ in two main ways: first, such an expression responds to an \( \text{asSQL()} \) method; second, any such expression can
be passed to the `tuple` macro, which returns a `PODS`¹¹ type with all accessible fields in this relation.

An `Exp` may involve (i) relations of the database, (ii) C++ variables storing other `Exp`s, 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 subscription operators of relational algebra. 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 `RA` fields. Many logical, arithmetical and builtin functions can be used to build scalar expressions. Only a handful of these are presented in Figure 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 un initialized 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.

### 4.2 Semantics

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

For example, in a selection `a/c` expression, ARA makes sure that every field name used in the scalar expression `c` exists in the symbol table of `a`. `RA` 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 Figure 4.3.

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

### 5.1 Combining Compile-time and Runtime 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 operations applicable to a large vector [46]. The implementation of RAT resisted the temptation of employing this encoding as is for representing relational algebra expressions, or boolean and arithmetical expressions used in selection and in defining new fields. Table 5.1 compares the compiler architecture (so to speak) of RAT with that of the expression templates library and SEMT. (Table rows represent compilers’ main stages).

<table>
<thead>
<tr>
<th>Compiler Stage</th>
<th>Expression Templates / SEMT</th>
<th>RAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical Analysis</td>
<td>C++ Compiler</td>
<td>C++ Compiler</td>
</tr>
<tr>
<td>Parsing</td>
<td>C++ Compiler</td>
<td>C++ Compiler</td>
</tr>
<tr>
<td>Type Checking</td>
<td>Degenerate</td>
<td>Template Engine</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Template Engine</td>
<td>Program runtime</td>
</tr>
</tbody>
</table>

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 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 semantical 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

---

¹¹ Plain Old Data Structure, i.e., a C like `struct`, with `public` data members only.
Figure 5.1: A re-implementation of function get_employees (Figure 3.3) in which selection is applied after projection.

```
1 | const 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();
```

(line 6 of Figure 3.3), which would have been impossible if the type of e represented its evaluation procedure.

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 a concept, if the type satisfies this concept’s requirements. Thus, a concept defines a subset of the universe of all possible C++ types. The notation $C_1 \leq C_2$ (concept $C_1$ refines concept $C_2$) means that every type that models $C_2$ also models $C_1$.

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 [33, 40, 49] 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. Table 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.

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 relational algebra projection). The last cell in the first row of the table is to say that a C++ expression of the form $v_1 v_2$ (applying operator $\circ$ to values $v_1$ and $v_2$) where $v_1$ and $v_2$ belong in $F$, returns a value in $V$. The type of this returned value represents the set of types of $v_1$ and $v_2$. For example, expression $\text{FIRST}_N\ \text{LAST}_N$ belongs in $V$, and it records the set of these two symbols.

The concept $\mathcal{R}$ represents singleton types. In our running example, the value $\text{FIRST}_N$ is the unique instance of the type representing the symbol $\text{FIRST}_N$. The macro invocation $\text{DEF}_F(\text{FULL}_N)$ (line 6 Figure 3.1) defines a new type that models $F$ along with its single value. Henceforth, for brevity sake we shall sacrifice accuracy

<table>
<thead>
<tr>
<th>Concept Name</th>
<th>Purpose</th>
<th>Sample Operations $^a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$ Field</td>
<td>Symbolic field name.</td>
<td>$F, F : V$</td>
</tr>
<tr>
<td>$I$ Initialization</td>
<td>A symbolic field name, along with an initialization expression, $F \preceq I$.</td>
<td>$I : I - V$</td>
</tr>
<tr>
<td>$V$ Vocabulary</td>
<td>A set of (possibly initialized) symbolic field names.</td>
<td>$V, V : V$</td>
</tr>
<tr>
<td>$S$ Scalar</td>
<td>An expression evaluating to a scalar, e.g., string, boolean, integer, obtained by applying arithmetic, comparison and SQL-like functions to fields, literals and variables, $F \preceq S$.</td>
<td>$S : S$</td>
</tr>
<tr>
<td>$R$ Relation</td>
<td>An expression in enriched relational algebra evaluating to a relation.</td>
<td>$R : R$, $R \circ R$</td>
</tr>
</tbody>
</table>

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

in saying that a value belongs in a certain concept meaning that this value’s type models this concept. This convention will prove particular useful when talking about singleton types. Thus, we say that this macro invocation defines the value $\text{FULL}_N$ in concept $F$.

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

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

(line 14 in Figure 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., $\text{cat}(\text{LAST}_N, *, \text{FIRST}_N)$ 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 $\text{FULL}_N$ to $\text{cat}(\text{LAST}_N, *, \text{FIRST}_N)$ is in $I$.

Since $F \preceq S$ we have that the function call operator can be used in particular to do relational algebra-renaming, writing, e.g., $\text{RBD}(I)$ (line 15 in this figure).

Another instance of $S$ is the expression, showing in line 10, $(\text{DEPTNUM} > 3 \ \text{&&} \ \text{SALARY} < 1E5)$. Note again that scalar expressions may involve literals.

Concept $R$ is used for representing relational algebra 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 relational algebra. 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 have been enumerated, we turn to describing how these concepts are used both in compile-time and in runtime to realize the integration of relational algebra 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 $\text{S_TREE}$. Type $\text{S_TREE}$ is the root of a type hierarchy that does a rather standard text book [41, pages 279–290] implementation of an expression tree, with classes such as $\text{S_BINARY}$ (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 relational algebra.

The representation is untyped in the sense that the actual eval-
uation of each node in the tree may return any of the supported
types of scalar expressions, including booleans, strings and inte-
gers. 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 func-
tion is implemented in the inheriting classes, to return the SQL rep-
resentation 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 relational algebra expression, which contains the
scalar expression, to SQL. At the time of the translation, S_LIT
leaves are printed as SQL literals, while S_FIELDs are printed as SQL
field names.

Figure 5.2 shows what a class S modeling concept S looks like.

```
1 class S {
2  public:
3    const S_TREE *t; // Expression tree of S
4    typedef ... TYP; // Compile-time representation of t
5    typedef ... FLDS; // List of fields used in t
6  ...  }
```

Figure 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 TYP is a compile-time representation of the content
   of t, i.e., TYP is tree-structured type, with the same topology
   as t. However, instead of literal values and addresses of vari-
   ables, 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 RA. All these functions gen-
erate types modeling S. Each concrete function template obtained
by instantiating these templates computes the value of data mem-
ber t in the return value, and generates the typedefs TYP 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:TYP is a type tree, rooted a node representing
addition with two type subtrees S1:TYP and S2:TYP.

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 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 prop-
erties that describe the expression. Therefore, these types cannot be
reassembled 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 s. 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:TYP.

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 t with e. This integration affects
   only the runtime value of e and therefore reassigning the new
   value into the same variable e 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. RA verifies that
all the uninitialized fields are present in s and all initialized fields
are absent of R. RA also verifies that the initialized fields can be
calculated when bound to s using the same algorithm used in the
selection operation. In addition, the compile-time encoding of the
result scheme and the symbol table of s are updated, which means
that the result of a projection operation is an object of a new type
which cannot be assigned to e.

4.4 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
typedef pseudo operator. Thus, macro _DEF_V in Section 3 creates a
new variable var_name, which has the type of expr, and initializes
this variable with the content of expr.

The typedef operator is a pseudo operator, since instead of return-
ing a value, it returns the type of the argument, which can be
used anywhere a type is used. Like sizeof, this operator is evalu-
at compile-time. This operator is found in e.g., all Gnu im-
plementations[17] of the language; its significance was also recog-
nized by the C++ committee, which is considering a similar mech-
anism for querying the type of an expression, namely dectype op-
erator, as a standard language extension[12].

Without typedef, 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 defini-
tions are correct and consistent with the query objects.

Type ordering and structural type equivalence. In relational al-
gebra, 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 equal-
ity of two such sets without resorting to sorting. However, we still
require a method to sort such sets, for the purpose of the implemen-
tation of the Tuple_T macro. Consider for example the statements

```
TUPLE_T<DEPARTMENT * DIVISION ...>
```

which define variables x and y to be of a PODS whose field
names and types are the union of the field lists of DIVISION and
DEPARTMENT.

There is a great challenge in making x and y compatible, due to
the fact that the type system of C++ (just as JAVA, C#, and many
other languages) obey a nominal type equivalence rule: two types
are equivalent only if they are defined at precisely the same location
and carry the same name. Thus, it may appear impossible to make
the two variables to be the same type, unless the exponentially-

sized set of all possible types is pre-defined. The enabling obser-

vation is that despite nominal typing, C++ types generated by tem-
plate instantiation obey structural equivalence rule, in that two identi-
tical sequences of template applications generate the same type, even
if carried out at in different program locations. (Curiously, but
not incidentally, structural type equivalence of template instances
is found in JAVA and C# just as well.)

RAT relies on this observation in the implementation of macro
TUPLE_T: a recursive template call is made to create an inheritance
chain, where each class in this chain defines a single field. In or-

der to make the generated types equivalents, it is essential that this
inheritance chain is created at a predetermined order. To this end,
a total ordering relation is placed on fields. In our implementation,
this order is realized by a unique integral value associated with ev-
ery field name. The identifier is generated using another C++ lan-
guage 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 [36], and is scheduled to
be included in version 4.3 of g++ (pending approval on the GCC
steering committee 14). Using this macro ensures that each field has
a constant identifier and thus every relation has a unique represen-
tation.

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 the re-
lational algebra formalism with C++, which can be used to generate
data 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.

The compilation time of the program in Figure 2.1 is shorter than
that of Figure 5.1. The compilation of the program in Figure
2.1 took 1.04 seconds while the program in Figure 5.1, which
uses the RAT library that consists of about 3000 lines of C++ code,
compiled in 1.26 seconds. 15 The time difference does not consti-
tute a main consideration in this sort of applications. But of course,
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 execu-
tion time of queries. We venture to project and extrapolate from
the measurements we made that the finding will be that construc-
tion time is negligible, and that generating queries using template
mechanisms does not impose a significant performance penalty.

The current implementation support of dynamic queries is lim-
ited 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_H == first);
  ...}
where DEF_DYNAMIC_COND(c, EMPLOYEE, TRUE); encodes the sym-
bol table of EMPLOYEE into c. Also easy in principle is the defini-
tion of "prepared SQL statements", by storing memory addresses
of C++ variables instead of actual content. The actual value of

13 Even if this is done, there remains the challenge of referencing such a
previously made definition.

14 Ian Lance Taylor, private communication, March 2007

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

these variables is retrieved whenever the query is translated to SQL.
Moreover, as hinted in brief in Section 5.4, RAT can easily gener-
ate function members for these queries which would allow dynamic
changes of the parameters of a prepared statement.

Conversely, it should be stated that the extent of flexibility of in
dynamic queries is not as great as with programs not using
ARARAT. For example, such programs may be able to generate
a query that returns a user-specified subset of the fields of a given
relation, whereas in ARARAT, every such subset must be available
for at compile-time for inspection by the compiler.

In Section 5 we described non-trivial implementation tech-
niques, 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 semanti-
cal 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 par-
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 Figure 3.2(b) and Figure 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, but not
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 Figure 4.2.

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

The addition of support for group by clause is more of a chal-
lenge, 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 execu-
tion 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 state-
ments with C++. This research should strive for a smooth integra-
tion 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-
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 pro-
gramming line of research [6, 20, 27, 28, 37]. It is likely that other
lessons of this subdiscipline can benefit the community in its strug-
gle 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.

Acknowledgements

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

References


