Science of Computer Programming 75 (2010) 573–595

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

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

ARTICLE INFO

Article history:
Received 11 February 2008
Received in revised form 29 December 2009
Accepted 19 January 2010
Available online 7 February 2010

Keywords:
C++
Template programming
Embedded languages
Domain specific languages
Databases
Structural type equivalence
Relational algebra

ABSTRACT

Most large software applications rely on an external relational database for storing and managing persistent data. Typically, such applications interact with the database by first constructing strings that represent SQL statements, and then submitting these for execution by the database engine. The fact that these statements are only checked for correctness at runtime is a source for many potential defects, including type and syntax errors and vulnerability to injection attacks.

The AraRat system presented here offers a method for dealing with these difficulties by coercing the host C++ compiler to do the necessary checks of the generated strings. A library of templates and preprocessor directives is used to embed in C++ a little language representing an augmented *relational algebra formalism*. Type checking of this embedded language, carried out by our template library, assures, at compile-time, the correctness and safety of the generated SQL strings. All SQL statements constructed by AraRat are guaranteed to be syntactically correct, and type safe with respect to the database schema. Moreover, AraRat statically ensures that the generated statements are immune to all injection attacks.

The standard techniques of "*expression templates*" and "*compile-time symbolic derivation*" for compile-time representation of symbolic structures, are enhanced in our system. We demonstrate the support of a type system and a symbol table lookup of the symbolic structure. A key observation of this work is that type equivalence of instantiated nominally typed generics in C++ (as well as other languages, e.g., Java) is structural rather than nominal. This makes it possible to embed the structural type system, characteristic to persistent data management, in the nominal type system of C++.

For some of its advanced features, AraRat relies on two small extensions to the standard C++ language: the `typeof` pseudo operator and the `__COUNTER__` preprocessor macro.

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 [5,4]). Fruits of this quest include e.g., the work on Pascal-R [54], a persistent (extended) version [2] of C [37], integration [14] of databases into Smalltalk [23], the XJ [29] system integrating XML with Java [3], and many more.

This problem is important since most large software applications make use of a database system. Development resources are invested in 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 carried out in a two step process: First, the application constructs strings — statements in the database engine control language. Then, these strings are sent for execution by the

☆ Research supported in part by the IBM faculty award.
* Corresponding author.
E-mail addresses: yogi@cs.technion.ac.il (J. Gil), lkeren@cs.technion.ac.il (K. Lenz).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.01.004

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. An added difficulty with the indirect, meta-programming process is that it is carried out with no aid of supportive, type-safe language environments and CASE tools.

In this work, we concentrate on the generation of SQL statements from C++ [60] programs. SQL is still the lingua franca of database processing, with adoptions of the standard [26], in all major database engines including Oracle [38], Microsoft SQL Server [42], MySQL [69], DB2 [46], and many more. Humans may find many advantages in SQL, including readability, well-defined expressive syntax, declarative semantics, flexibility, etc. Most of these advantages are lost however when instead of writing SQL directly, a human is required to write application programs that write SQL. One of the major difficulties in writing such programs, is that the SQL statements they generate are only checked for correctness at runtime. The logic of producing correct SQL strings may be complex. Further, coding errors in the process may invite *injection attacks* which compromise the safety and integrity of the entire database. These and other difficulties in producing SQL at runtime are discussed in brief below in Section 3 and in great detail in the literature (see, e.g., [9,17]).

AraRat, the system presented here, employs C++ templates to address the common problem of integrating a database language into a programming language. Type checking of this embedded language, carried out by our template library, assures, at compile-time, the correctness and safety of the generated SQL strings. All SQL statements constructed by AraRat are guaranteed to be syntactically correct, and type safe with respect to the database schema. Moreover, AraRat statically ensures that the generated statements are immune to all injection attacks.

A library of templates is used to embed in C++ a little language [8] representing a relational algebra formalism. In our language, C++ operators replace the mathematical operators of relational algebra, e.g., `+` is union ($\cup$), `/` is selection (σ). Our language augments relational algebra with abbreviated assignments, e.g., `+=`, and `/=`. 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 an embedded language in C++. Thus, Rat is the Ara compiler. Key features of AraRat include: (i) tight integration with the host language using *template programming*, with minimal reliance on external tools and non-standard language extensions, (ii) direct reliance on relational algebra, rather than through SQL. (Some readers may appreciate a measure of elegance in the resulting language.)

We describe techniques for embedding little languages within the C++ language, using the template mechanism and without modifications to the compiler. 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 generic mechanisms in future languages. Hopefully, from our experience language designers may gain intuition of the different features that generic mechanisms should offer. For example, we believe our work makes part of the case for including a mechanism for eliciting the type of an expression, such as the **`typeof`** operator found in some contemporary compilers, or, the more advanced proposed additions to the language's standard.

AraRat also provides support for the task of defining C++ data structures required for storing the results returned by database queries. The challenge in doing that is significant: the type of a query result is inherently *structural* in that two queries returning the same set of fields (i.e., named–typed-columns in a relation) must have the same type, regardless of the method by which these fields were collected. In contrast, the types found in mainstream application programming languages are typically *nominal*: each record type in (say) C++ must be defined and named separately, and once such a definition is made, no other record type is equivalent to that type, even if it has the same set of fields. To bridge the gap between the two systems, AraRat relies on the observation that the instantiation of generics in C++ (as it is in many other nominative programming languages) is structural. We demonstrate how this unique property of generics can be used to support not only *structural equivalence*, but also *structural subtyping* in the nominative type system of C++, that is, we show how template instantiation can be used to define record types in such a way that each such record type is a subtype of all record types which contain a subset of its fields.

Use of relational algebra

Historically, SQL emerged from the seminal work of Codd [12] on relational algebra. 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; 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 always possible in SQL.

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

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

The expression `EMPLOYEE[SALARY]` *projects* the `EMPLOYEE` relation, selecting only the `SALARY` field. The division operator, "`/`", *restricts* the projected records to those in which `DEPTNUM == 3`. Finally, the left shift operator, "`<<`", sends to the stream `dbcon` the resulting SQL statement i.e.,

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

Moreover, if `q` is a C++ variable storing a query to a database, writing `TUPLE_T(q) r;` defines `r` to be of the type of records that `q` returns.

Fig. 1.1 A LINQ code snippet.

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

Comparison with related work

A number of constructs of SQL address some of the difficulties of programmatical composition: a systematic use of *prepared statements* adds a layer of safety at the cost of excluding dynamic composition of statements; *command parameters* defend against injection attacks, but not against syntax errors arising from dynamical composition of statements. Our focus here is in more than that: a closer integration, as seamless as possible, of SQL with the host language, an issue which has engaged the research community for at least a decade.

A first approach to this problem is in processing of embedded SQL statements by a dedicated external preprocessor as done in e.g., SQLJ [18], SchemeQL [68] 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 [41] whose structure reflects the constraints which the database schema imposes, and allows production of correct SQL statements only. Such systems trade expressive power for usability and require the user to use a large schema-specific library, with e.g., a class for each field, a class for each condition on each field, etc.

Third, in LINQ [44], Xen [45] and the work of Van Wyk, Krishnan, Lijesh, Bodin and Johnson [73] the host language compiler is modified to support SQL- (or XML-)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 work entitled *Safe Query Objects* [13] which relies on the reflective features of OpenJava [63] to achieve such syntactical changes and to encode constraints imposed by the schema. 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. Consider for example, the LINQ code snippet in Fig. 1.1 which is drawn from Microsoft documentation. Despite using SQL vocabulary (keywords **`select`** and **`where`**) and functions (`Sum`), the code in the example may need some pondering before it is revealed that it relies on the non-standard nested relational model. The translation to plain SQL, although guaranteed to be possible (the nested and the non-nested relational models are known to be equivalent) may not be straightforward. Worse, although LINQ offers textual resemblance to SQL statements, this resemblance may be misleading, since not all SQL statements have an equivalent LINQ representation; the programmer thus need to learn a new variant of SQL in order to use LINQ. In contrast, AraRat is isomorphic to relational algebra, the only differences being operator names and AraRat's support for assignment.

Finally, there is the approach of using an external static analyzer that checks that the program only produces correct SQL statements [28,24,11]. 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 [39] system, an SQL library for Haskell [35]. In HaskellDB, just like in AraRat, columns of a query's result are carried in the type of the query, and therefore the support of dynamic queries is limited to changes to the conditions of the **`where`** clause. However, unlike HaskellDB, by relying on C++ AraRat 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. Given a description of a database schema, which can be obtained using the DB2Ara tool or manually, 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, AraRat allows for a manipulation of SQL query objects at runtime — that is, one can construct a query object, and then refine the query, without actually executing it (see Fig. 4.3 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.

[1] msdn.microsoft.com/library/default.asp?url=/library/enus/esqlforc/ec_6_epr_01_3m03.asp, 2004.

We should also mention C++ libraries, including e.g., *Ultimate++*,[2] *Terimber*,[3] *Postgress C library*,[4] *IBPP*,[5] *CQL*,[6] and *SourcePro©-DB*[7] which provide an extensive set of functions that can be used for the production of SQL. The support for the generation of SQL statements using these libraries ranges from none to elaborate mechanisms found in *Ultimate++* allowing partial emulation of the SQL syntax with C++ functions calls, as in

```
SqlSet borrowed = Select(BOOK_ID).From(BORROW_RECORD).Where(IsNull(RETURNED));
```

However, none of these libraries apply static type checking to the produced SQL statements. On the other hand, these libraries excel in many aspects of database management which AraRat neglects, including the managements of connections, passwords, transactions, etc.

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 [58] C++ compliant compiler. For more advanced features, we rely, as described in Section 6.4, on two small extensions: the `typeof` operator and the `__COUNTER__` 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.

Outline. We start with two preliminary sections: Section 2 discusses the distinction between nominal and structural types and reviews some techniques of C++ template programming, and then Section 3 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 3 is revisited in Section 4, which shows how the same query is expressed in Ara. Section 5 then gives a high-level description of the entire Ara language. Section 6 describes in greater detail the Rat architecture and the template programming techniques used there. Section 7 demonstrates how AraRat supports structural equivalence and structural subtyping in a nominative language. The discussion in Section 8 highlights the advantages and limitations of our solution and draws some directions for further research.

2. Preliminaries I: Typing and templates

2.1. Nominal vs. structural typing

This section should serve as a reminder of the differences between nominal and structural typing, and explain why we think of the instantiation of C++ templates as structural even though the language is largely nominal. As it turns out, such a mixup between the paradigms, is not C++ specific. Indeed, B. Pierce stated that '*A few nominal languages have been extended with such "generic" features but the results of such extensions are no longer pure nominal systems, but somewhat complex hybrids of the two approaches.*" [51, p. 254].

Our implementation makes an important use of the observation that template instantiation in C++ is structural. (Still, there are other issues in extending our technique to other programming languages.) Section 2.2 below explains why dealing with external data requires structural typing, and provides further intuition of the reasons why SQL is structurally typed. Take note however, that our perspective on the structural typing properties of C++ templates may be objectional to some, as we shall explain.

A nominal type system, as found e.g., in Pascal [71] dictates that a type is equivalent only to itself, and that subtyping is explicit rather than implicit. In contrast, in structural type systems, found e.g., in Haskell, type compatibility and equivalence are determined based on the structure of the type, i.e., two types are considered compatible if they have identical structure.

Fig. 2.1 demonstrates the difference between the two. Fig. 2.1(a) shows a Pascal program fragment, in which the two types `T1` and `T2` are distinct, even though they are both arrays of integers whose indices are in the range 1 . . . 100, because they refer to two distinct type definitions. Thus, variable `a1` of type `T1` is not assignment compatible with variable `b1` of type `T2`, even though the two variables have structurally the same type.

For the same reason, variables `a2` and `b2` are not assignment compatible. The anonymous type of each of these two vanishes after the variable was declared, and cannot be used after the declaration terminated. (In contrast, variables `a3` and `b3` refer to the same anonymous type and are therefore assignment compatible with each other.)

Fig. 2.1(b) demonstrates structural typing in C: Although variables `a` and `b` are defined in distinct statements, they are of the same type (an array of 100 elements of type `int`), and are assignment compatible. The same type also receives the names `T1` and `T2` by means of the `typedef` declarations. But, these type names do not generate new types. All four variables defined in this short code excerpt, `a`, `b`, `c1` and `c2`, have the same exact type and are assignment compatible.

2 http://www.ultimatepp.org.
3 http://www.terimber.com.
4 http://www.postgresql.org/docs/8.1/interactive/libpq.html.
5 http://www.ibpp.org.
6 http://www.cql.com.
7 http://www.roguewave.com/sourcepro/sourcepro.cfm.

Fig. 2.1 Nominal typing in PASCAL (a) vs. structural typing in C (b).

```
TYPE
    T1 = array[1..100] of Integer;
    T2 = array[1..100] of Integer; (* incompatible with T1 *)
VAR
    a1: T1;
    b1: T2; (* incompatible with a1 *)
    a2: array[1..100] of Integer;
    b2: array[1..100] of Integer; (* incompatible with a2 *)
    a3, b3: array[1..100] of Integer; (* a3 and b3 are mutually compatible *)
```

(a) Nominal typing in PASCAL.

```
int a[100];
int b[100]; /* compatible with a */
typedef int T1[100];
typedef int T2[100]; /* same type as T1 */
T1 c1; /* compatible with a and b */
T2 c2; /* compatible with a, b and c1 */
```

(b) Structural typing in C.

Fig. 2.2 Non-equivalent `struct` types in C.

```
struct {
   int id;
   char* name;
} p1;
```

```
struct {
   int id;
   char* name;
} p2;
```

Unlike arrays, pointers, and other C type constructors which are structurally typed, equivalence of aggregate type constructors, that is `struct` and `union` types in C is determined nominally, as demonstrated in Fig. 2.2. Each of the variables `p1` and `p2` in the figure refers to its own anonymous type which is discarded after the variables are defined.

Nominal typing of aggregate types carries through from C to C++'s `class` type constructor. Moreover, *subtyping* of class types is nominal rather than structural; one class is a subtype of another only if it is declared as such. Such a declaration also dictates that the set of members of the subtype is a super set of the set of members of the supertype. In contrast, in structural typing system, a type is a super type of another if every member found in the former is also found in the latter.

Although type names play a less crucial role in structural type systems, such systems usually allow naming types; for example, the following defines two named types whose equivalence is structural.

```
typedef void(*g)(int a, int b);
typedef void(*f)(int i, int j);
```

The structural equivalence implies that a value of type `f` may be assigned to a variable of type `g`.

2.2. Structural typing in SQL

The many advantages of nominal type systems (see e.g., [40,51] for a discussion of the relative merits of the two approaches), make these the rule in mainstream programming languages, including JAVA, C# [30], SMALLTALK, and C++ (aggregate types). However, there is one crucial point in which nominal type systems fail, that is, the interaction with external data: Data stored on permanent storage, data received from communication lines, and all other external data cannot be nominally typed, since all nominal types declared within a program vanish when the program terminates. Nominal types defined in distinct programs are necessarily distinct, and therefore data serving for program communication must be structurally typed.

As SQL deals with permanent data, it must therefore be structurally typed. Other reasons for adopting this typing rule include: the reliance on relational algebra, which is structural in its essence, and the need to apply a union of tables of the same structure generated by different sequence of operations.

A fundamental rule of SQL is that two tables are mutually compatible if they have the same set of fields. This rule applies not only to the basic relations as they are found in the database but also to interim relations obtained in the course of computing queries: a union (say) of two subqueries is possible if they both contain the same set of fields. The difficulty that ARARAT faces is in using the nominal type system, with types particular to the programs they are defined in, for manipulating external data which was not created using these types.

What made ARARAT possible is the observation that template instantiations (just as arrays) are structurally typed in the following sense. If `Stack` is a C++ `template` taking a single type parameter, then variables `a` and `b` defined by

```
Stack<int> a;
Stack<int> b;
```

Fig. 2.3 Computing factorials at compile-time with C++ templates.

```
1 template<int N>
2    class factorial {
3       public: enum {result = N*factorial<N-1>::result}  };
4    };

6 class factorial<1> {
7    public: enum {result = 1};
8 };
```

are compatible.

As shall be explained below, the compatibility of variables `x` and `y` defined by

```
DEF_V(x, DIVISION*DEPARTMENT); // Variable x has the type of the join of two relations.
DEF_V(y, DEPARTMENT*DIVISION); // Variable y has the type of the same join, but in opposite order.
```

is achieved by using two structurally identical sequences of template instantiations, that generate the same type. (The macro `DEF_V(v, E)` defines a variable `v` whose type is the same as that of relational expression `E`).

The single instantiation rule. Consider again an example such as

```
Stack<int> a;
Stack<int> b;
```

The fact that `a` and `b` are of the same type can be explained not by structural equivalence of template instantiation, but rather by what is known as the "single instantiation rule" (which may be thought of as lazy evaluation of type functions). From this perspective, *only* the first occurrence of `Stack<int>` leads to the generation of a new type. All subsequent occurrences refer to *exactly* that type. Further, the fact that `a` and `b` are of the same type is explained by the compiler fetching the mangled name of that type from its symbol table, rather than by taking its structure into account. Thus, by this perspective we do not have structural typing at all in template instantiation, but rather a single type, with a single name, which is referenced from different code locations. In the example above, this name is `Stack<int>`, or whatever mangled name the compiler chooses to call it.

Issues with this alternative perspective may be the perceived promotion of mangling, an internal compilation strategy, into the language type system, and the fact that type equivalence occurs even when there is no instantiation at all, as in

```
Stack<int> *p; // No instantiation of Stack<int> is required here,
Stack<int> *q; // neither here.
// ...
p = q; // p and q are compatible
```

Another concern is that identical instantiations of a template in distinct compilation units may still generate distinct, yet identical types in some compilers.[8]

Note that structural typing of **`template`** instantiation is in accordance with C++'s "single instantiation rule". Without structural typing, multiple instantiations of a class template with the same arguments, would have always resulted in multiple copies of that class in the generated binary. If this was the case, the acute code bloat problem due to template instantiation, could not be solved by clever techniques such as those described by B. Stroustrup in chapters 15.6.3 and 15.10.4 of the "The Design and Evolution of C++" [59]. Still, structural equivalence of generic instantiation is found in other nominally typed languages in which the problem of code bloat is minimal (e.g., C#) or non-existent (e.g., JAVA).

2.3. Programming with C++ templates

C++ templates were initially designed for generic data structures and algorithms, as used in STL [52]. The community, however, has observed that this language mechanism is powerful enough to serve many other tasks. This section reviews some of the techniques of template programming, giving the intuition why templates can be used to encode all algorithms, and how this encoding may be achieved. Readers who are familiar with template programming techniques may skip forward to Section 3.

Fig. 2.3 shows a simple example which, relying on integer constant parameters to templates, computes factorials at compile-time.

The first template in the figure makes the recursive definition. The *template specialization* (lines 6–8) is the recursion base. The figure thus demonstrates two important techniques of template programming: using recursive calls for iteration and specialization for conditions. Together, the two techniques make C++ templates (unlike JAVA generics) Turing complete (see e.g., [10] and Gutterman's [27] emulation of two-stacks Turing machine with templates). Turing completeness means that there *exists* a C++ templates implementation of any algorithm that can be implemented in (say) C++. Deriving this implementation is sometimes difficult; the literature offers many techniques for making this task easier.

8 See for example, Marshall Cline famous C++ FAQ, questions 35.13, 35.15.

Fig. 2.4 Implementing a list of types data structure with templates.

```
template<typename First, typename Rest>
  class Cons { // Generate a list whose first element is First while the remaining elements are Rest.
  public:
    typedef First CAR; // Use a LISP like notation,
    typedef Rest CDR; // also for the CDR component.
 };
```

A basic technique is of using templates to represent data structures. Fig. 2.4 for example shows how **typedef** instructions inside a class make it possible for one type to store what can be thought of as "pointer" to another type.

The template `Cons` in the figure demonstrates how LISP-like lists can be implemented. Recursive, LISP-like processing of lists, lists-of-lists, is common in the field of template programming, with the recursion being realized by template specialization, (much as in Fig. 2.3). The generalization of lists to trees of types, lookup-tables and other advanced data structures, is not difficult and has become part of the tools of the trade. There is even a library of data structures and algorithms, Boost MPL,[9] which manipulates types at compile-time much like STL manipulates data at runtime. Boost even includes an implementation of unnamed functions, in the form of λ-expressions [32].

An important technique, to which we will refer later, is the standard encoding of symbolic expressions as types e.g., for symbolic derivation (SEMT [20]) or for emitting efficient code after gathering all operations applicable to a large vector [67]. Three principles are applied by such an encoding:

(i) Literals are encoded either as simple types, or as templates receiving the literal value.

```
class EMPTY_SET { // the ∅ literal, used e.g., in set-theoretical symbolic expressions.
};
template<int N> class NUMBER { // an integral value, used e.g., in arithmetical symbolic expressions.
   public: enum {value = N};
};
template<char C> class VARIABLE { // a symbolic variable, denoted by character C
   public: enum {name = C};
};
```

(ii) An n-ary operator is encoded by a template taking n-arguments. For example, binary addition operator may be encoded by

```
template<typename L, typename R> class PLUS {
   public: typedef L left; typedef R right;
};
```

(iii) The type encoding of a non-atomic symbolic expression is defined recursively as follows. Apply the template of the top most operator, to the types of the arguments of this operator. So, the type encoding of $e_1 + e_2$, is `PLUS<T1,T2>` where `T1` is type encoding of e_1 and `T2` is the type encoding of e_2.

At a later point in time, but still at compile-time, this type structure can be processed, often by a recursive traversal, to generate other compile time structures, or to generate actual code.

Some of the diverse applications in which template programming is used include dimensional analysis [65], a solution of the co-variance problem [61], and even for implementing a framework for aspect oriented programming [74]. Czarnecki and Eisenecker, in their work on application of template programming [15], marked the emergence of an important research trend on generative programming [7,50,36,22]. RAT uses many of the techniques developed in this research, including methods for different overloading of the same operator for different sets of types, traits [48], etc.

Most of the ARARAT's work is carried out by the C++ compiler itself, which is driven by template programming to make sure that the compiled program will generate at runtime only correct and safe SQL. Our implementation started from the established techniques of template programming [15,1], and extended these to meet the challenges of realizing a little language. For example, the techniques for producing (relatively) meaningful and short compilation errors in response for errors in use of the template library are borrowed from [43,55].

RAT representation of relational algebra expressions is hybrid, straddling the compile-time and runtime worlds. The C++ type of an ARA expression reflects the schema of the associated query, allowing type checking the query at compile-time, while the actual evaluation procedure, that is, the Abstract Syntax Tree of the query, is represented as a runtime value. This hybrid representation allows repeated assignment of different queries that share the same result schema, to a single C++ variable, whose type encodes that schema. The compile-time encoding of the result schema is used for generation of a data type for the query result, which represents a single row in the result relation.

This paper shows that the C++ template 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

[9] www.boost.org/libs/mpl/doc/index.html, July 2002.

Table 1
A database schema, to be used as running example.

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

Fig. 3.1 Function returning an erroneous SQL query string.

```
1 const char* get_employees(int dept, char* first) {
2   bool first_cond = true;
3   string& 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     first_cond = false;
11  }
12  if (first == null)
13     return s.c_str();

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

23  return s.c_str();
24 }
```

library[10] in which BNF syntax is embedded within C++, while the compiler, as part of the compilation process generates an LL parser for the given syntax. Our system differs from this (and other such examples) in that the type system of Boost Spirit is degenerate in the sense that it has a fixed small set of types: grammars, actors, closures, etc. All symbols in a BNF belong to a single type. In contrast, AraRat features a multi-sorted type system, allowing two queries to be combined only if they are compatible.

A specific challenge in this management is that the type system of queries is structural rather than nominal. The observation that made it possible to meet this challenge is that instantiations of generic classes obeys structural equivalence rules, even in nominally typed languages. We further demonstrate that non-trivial symbol table management, can be carried out as part of template based computation.

3. Preliminaries II: Problem definition

This section motivates our work by demonstrating the intricacies of the existing string based database access from C++. Table 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. Field DEPTNUM is used in relation EMPLOYEE as a foreign key for referencing relation DEPARTMENT, i.e., a typical join operation of EMPLOYEE and DEPARTMENT will involve renaming of field ID in DEPARTMENT to DEPTNUM. For simplicity, we used field DIVNUM both as a primary key in the third relation and a foreign key in the second one. 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. The AraRat type manager supports this.

Fig. 3.1 shows a simple C++ function that might be used in an application that uses the database described in Table 1.

Function `get_employees` receives an integer `dept` and a string `first`, and returns an SQL statement in string format that evaluates to the set of all employees who work in department `dept` bearing the first name `first`.

10 spirit.sourceforge.net/.

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:

(i) *Misspelled name.* A typo lurks in line 4, by which `EMPLOYES` is used (instead of `EMPLOYEE`) for the table name. This typo will go undetected by the compiler.
(ii) *Syntax error.* There is no space after the `AND` keyword in line 18, and thus, if both parameters are non-null, the produced statement is syntactically incorrect.
(iii) *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 source of this kind of problem, which is sometimes called in the literature *impedance mismatch* [72], an inherent problem of integrating database engines and programming languages. One aspect of the difficulty is that the type systems of the underlying programming language (C++ in this case) and the database language (SQL) are distinct – one may even say, foreign to each other – and thus very little checking can be done at the junction of the two languages.
(iv) *Security vulnerability.* The code is vulnerable to SQL script injection attacks [31] as, a malicious user can construct a value of the `first` parameter that executes unexpected statement that harms the database.
(v) *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.
(vi) *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 schema 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 the C++ type system prevent such attacks. Consider e.g., a query generated by the following simple C++ code:

```
char *makeSelectUserQuery(char *name, char *pin) {
   char* result = (char*)malloc(1000 + strlen(name) + strlen(pin));
   sprintf(result, "select * from users where name='%s' and pin=%s", name,pin);
   return result;
}
```

This simple function is vulnerable to attacks in which a malicious user supplies the string "`1 or 1=1`" (without the quotes) into variable `$pin`. This injected value becomes part of the generated SQL statement,

```
select * from users where name='administrator' 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 Fig. 5.2, 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, 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 schema are flagged with compilation errors.

4. Relational algebra queries

An Ara programmer wishing to execute a database query must create first a *query object*, encoding both the specification of the query and the schema of its result. The 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:

(i) *Type.* The type of a query object encodes in it the schema of the result of the 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.
(ii) *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 operator to string, that invokes `asSQL()`, so query objects can be used anywhere a value of type **`const char *`** can be used.

Fig. 4.1 Writing a simple relational algebra expression in ARA.

```
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);
6 DEF_F(EID); // Define field names which were not defined in the input scheme

8 int main(int argc, char* argv[]) {
9  const char *s = (
10   (EMPLOYEE / (DEPTNUM > 3 && SALARY <= 100000))  // Selection of a tuple subset
11   [
12    FIRST_N, LAST_N,
13    FULL_N(cat(LAST_N,", ", FIRST_N)),
14    EID(ID)
15   ]
16 ).asSQL();
17 // ... execute the SQL query in s using e.g., ADO.
18 return 0;
19 }
```

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

4.1. Composing query objects

Fig. 4.1 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.

In lines 10–15 of this figure, a compound query object is generated in two steps:

- First (line 10), the expression `EMPLOYEE / (DEPTNUM > 3 && SALARY <= 100000 )` 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 `operator /` is used for applying this criterion to `EMPLOYEE`.
- In lines 11–15 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 `EID` (which is just field `ID` renamed).

  Note that the ARARAT 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 `EID(ID)` uses this operator for field renaming.

After this query object is created, its function member `asSQL()` is invoked (in line 16) to convert it into an equivalent SQL statement ready for execution:

```
select
   FIRST_N, LAST_N,
   concat(LAST_N,", ", FIRST_N ) as FULL_N,
   ID as EID
from EMPLOYEE where DEPTNUM > 3 and SALARY <= 100000;
```

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

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 2 summarizes the ARA equivalents of the main operators of relational algebra.

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

Table 2
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** * **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

Fig. 4.2 Three alternative 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.

```
(
  (DIVISION * DEPARTMENT)
    /
  (CITY == "Haifa")
)
[MANAGER]
```

(a) Operator overloading version.

```
project(
   select(
      join(DIVISION,DEPARTMENT),
      (CITY == "Haifa")
   ),MANAGER)
```

(b) Global functions version.

```
DIVISION
   .join(DEPARTMENT)
      .select(CITY == "Haifa")
         .project(MANAGER)
```

(c) Member functions version.

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. Fig. 4.2 shows how a query object for finding the managers of departments of divisions in the city of Haifa can be generated using overloaded operators, global functions and member functions.

The composition of query objects with RAT is type safe — an attempt to generate illegal queries will result 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 that do not exist in the relation, or select upon conditions that include such fields.

4.2. Storing query objects in variables

A single C++ statement was used in Fig. 4.1 to generate the desired query object. But, as can be seen in this example, as well as in Fig. 4.2, 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 types: 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. 4.3 A rewrite of function `get_employees` (Fig. 3.1) using AraRat.

```
1 const char* get_employees(short dept, char* first) {
2    DEF_V(e,EMPLOYEE);
3    if (first != null)
4       e /= (FIRST_N == first);
5    if (dept > 0)
6       e /= (DEPTNUM == dept);
7    return e[FIRST_N,LAST_N].asSQL();
8 }
```

Fig. 4.4 Using AraRat to find the names of employees who earn more than their managers.

```
1 DEF_F(M_SALARY);

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

Fig. 4.3 makes use of query variables assignment to recode our motivating example, Fig. 3.1, in Rat. Line 2 defines variable `e` which is initialized to the `EMPLOYEE` primitive query object. This line uses 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: first for setting the type of the new variable by means of Gnu C++ [19] operator **`typeof`**, and again, to initialize this variable. Hence, variable `e` can represent queries that return a relation with the same fields as `EMPLOYEE`. Initially, the evaluation procedure stored in this variable is an abstract encoding of the instruction to take all tuples of `EMPLOYEE`, but lines 3–6 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]` representing 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`. The expression on Line 7 creates a temporary value whose type reflects the structure of the query result.

Note that the example uses the abbreviated assignment operator (**`operator`** `/=`) to modify the query variable `e`. This operator, that may be thought of as an extension of relational algebra, is legal, since selection does not change the type of the result. Similarly, Rat offers abbreviated assignment operators for union and subtraction, 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()];
```

4.3. 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 [13].) The code could have been written as a single expression of relational algebra. Instead, Fig. 4.4 shows the computation in three stages.

Line 4 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 * 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 using AraRat. A three-argument function `joinOn` that does the same, but without explicitly naming all fields, is a feasible addition to AraRat.)

Then, in line 5 we rename the fields in relation `EMPLOYEE` so that they can be used as descriptives of managers. Finally, line 6 does another join, to determine manager's salary, projection of the desired fields, and then a selection of employees who earn more than their managers.

Observe that many non-useful fields are live in intermediate steps of the computation of Fig. 4.4. For example, after line 4, the query variable `e` contains all `EMPLOYEE` fields, some of which, as `LOCATION`, are not used in the next steps of the query construction. No efficiency concern is to be made here: the computation is of a query, rather that on its result. Only the final result is submitted to optimized execution by the database engine.

Fig. 5.1 The grammar of ARA. Part I: schema definition.

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

Fig. 5.2 The grammar of ARA. Part II: 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.

5. 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: the user is expected 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 how these techniques might be used for realizing other little languages.

5.1. Syntax

ARA extends C++ to allow a programmer to include any number of ARA *definitions* and *statements* within the code. The BNF grammar of definitions is given in Fig. 5.1.

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

(i) *Field definitions* are used to define the field label 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 1).

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

Field names are untyped until bound to a relation.

(ii) *Relation definitions*, which are similar to SQL's *Data Definition Language* (DDL) statements, are used for specifying a database schema, by declaring relation names and the list of field-type pairs included in each. For example, the schema of the last relation of our running example, is specified by `DEF_R(DIVISION, (DIVNUM/SMALLINT,CITY/STRING))` (Such definitions are normally generated by the DB2ARA tool but evidently can be easily produced by hand.)

(iii) *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 in Fig. 5.2, but any relation defined by `DEF_R` is also an expression. The statement `DEF_V(e,EMPLOYEE)` in Fig. 4.3 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. 5.2 gives a (partial) syntax of ARA statements.

The most important non-terminal in the grammar is Exp, which denotes a relational algebra expression obtained by applying relational algebra 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 *record*, a class that has no methods and that has all of its fields as **`public`** [21], containing 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 subtraction operators of relational algebra. Similarly, a statement may apply

Fig. 5.3 Some semantic checks applied by RAT.

(i) In union `e1+e2`, and in subtraction `e1-e2`, the sets of fields of `e1` and `e2` must be the same.
(ii) 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.
(iii) In a selection, `e/c`, expression `c` is a properly valid expression of boolean type.
(iv) 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.
(v) 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 that does not occur in the vocabulary, then this field exists in the projected relation.

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, subtraction, 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. 5.2.

Finally, note that the projection operation (`operator []`) involves a Vocabulary, which is 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.

5.2. Semantics

RAT defines numerous semantic 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. 5.3.

6. System architecture

The ARA little language is implemented solely with the 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 6.1, 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 6.2 then discusses some of the main components of the RAT architecture. Section 6.3 demonstrates the technique, showing how these components cooperate to achieve compile-time assurance that only boolean expressions are used for selection. Finally, Section 6.4 discusses the two compiler extensions that RAT uses.

6.1. Combining compile-time and runtime representations

As explained above, query objects *do not* use the standard representation of symbolic expressions as types (described in Section 2.3) for representing the expressions of relational algebra. This technique is however used for the representation of boolean and arithmetical expressions used in selection and in defining new fields.

Table 3 compares the compiler architecture (so to speak) of RAT with that of the expression templates library and SEMT. (Table rows represent compilers' main stages.)

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.

Table 3
Realizing compiler stages with template libraries.

Compiler Stage	Expression Templates [67] / SEMT [20]	RAT
Lexical Analysis	C++ Compiler	C++ Compiler
Parsing	C++ Compiler	C++ Compiler
Type Checking	Single-sorted	Template Engine
Code Generation	Template Engine	Program runtime

Fig. 6.1 A re-implementation of function `get_employees` (Fig. 4.3) 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();
6 }
```

The main objective of expression templates and SEMT is *runtime* efficiency. Accordingly, the type system in these is single-sorted, and code is generated at compile-time by the template engine. Conversely, ARA is designed to maximize compile-time safety, and includes it own type- and semantic-rules, as was described in Section 5.2. ARA has a multi-sorted type system, and non-trivial semantic 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., it may 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 type dictionary of the input schema.

This technique (together with the delayed execution 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 as `e /= (DEPTNUM == dept);` (line 6 of Fig. 4.3), which would have been impossible if the type of `e` reflected the Abstract Syntax Tree of the query rather than the result type of the query.

Curiously, this technique supports what may seem paradoxical at first sight: a selection based on fields that were projected out, making it possible to rewrite Fig. 4.3 as Fig. 6.1.

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:

(i) *Active fields.* This is a list with names and types of all fields in the tuples computed by evaluating the query; and
(ii) *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.

6.2. Concepts in RAT

A *type concept* [6] (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 $\mathcal{C}_1 \preceq \mathcal{C}_2$ (concept $\mathcal{C}_1$ *refines* concept $\mathcal{C}_2$) 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 [43,70,55] for realizing concepts in the language in such a way that they are checked at compile-time.) A language extension to support concepts [25] 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 4 summarizes the main such concepts. Comparing this table with the language grammar (Figs. 5.1 and 5.2), 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 states 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, returns a value in V. The type of this returned value represents the set of types of v_1 and v_2. For example, the expression `FIRST_N, LAST_N` belongs in V, and it records the set of these two symbols.

Table 4
The main concepts in the RAT architecture.

	Concept name	Purpose	Sample operations
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, $F \preceq S$.	$S{+}S : S$ $S{*}S : S$ $\texttt{cat}\,(S, S) : S$ $S{>=}S : S$ $S \;\texttt{||}\; S : S$
R	Relation	An expression in enriched relational algebra evaluating to a relation.	$R{*}R : R$ $R{+}R : R$ $R{-}R : R$ $R\,[V\,] : R$ $R/S : R$

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.

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. 4.1) defines a new type that models F along with its single value. Henceforth, for brevity's 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 particularly 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))
```
(1)

(line 13 in Fig. 4.1). 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 (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 relational algebra-renaming, writing, e.g., `EID(ID)` (line 14 in this figure).

Another instance of S is the expression `(DEPTNUM > 3 && SALARY <= 100000)` shown in line 10, which demonstrates 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, subtraction, 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.

6.3. 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 at both compile-time and 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.

6.3.1. 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 standard text book [56, 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 relational algebra.

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 contrast to the standard representation of an expression tree, the nodes of this tree do not have an evaluation function. Instead, the class `S_TREE` has a pure virtual

Fig. 6.2 The main component of a type modeling concept *S*.

```
1 class S {
2   public:
3     const S_TREE *t; // Expression tree of S
4     typedef ... TYPS; // Compile-time representation of t
5     typedef ... FLDS; // List of fields used in t
6     ...
7 };
```

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 relational algebra expression, which contains the scalar expression. At the time of the translation, `S_LIT` leaves are printed as SQL literals, while `S_FIELD` leaves are printed as SQL field names.

Fig. 6.2 shows what a class `S` modeling concept *S* looks like.

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`**:

(i) 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.

(ii) 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) the list `S::FLDS` is the merge of `S1::FLDS` and `S2::FLDS`, and (ii) the type tree `S::TYPS` is rooted at a node representing an addition (i.e., a union in the relational algebra sense) of 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.

6.3.2. 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 schema of the result of a query, and a list of fields that 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:

(i) A check that all fields in `S::FLDS` are present in `R`.

(ii) Binding each field of `S` to its type in `R` to analyze the types of `S::TYPS`.

(iii) Issuing a compilation error, if there is a type mismatch between an operator and its operands or if the result type is nonboolean.

(iv) 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 schema 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`.

6.4. Compiler extensions

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 Section 4 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 implementations [19] 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 the `decltype` operator, as a standard language extension.[11]

An alternative implementation of the `DEF_V` macro could rely on another extension that is being considered for the next revision of C++—the **auto** keyword for indicating that the compiler should deduce the type of a variable from its initializer expression. Using this extension, we could rewrite the definition of macro `DEF_V` as **#define** `DEF_V(var_name, exp)` **auto** `var_name = exp` and this definition would have been equivalent to the one described in Section 4.

With neither **typeof** nor **auto**, 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.

Field ordering

As shall be explained in Section 7, the `TUPLE_T` macro generates a record whose field names and types correspond to the fields in the result relation of a query object by a recursive template instantiation. In order to generate equivalent types for equivalent relations, it is essential that this instantiation chain be applied in 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 every field name. The identifier is generated using another C++ application extension, the `__COUNTER__` macro. This macro is a compile-time counter, evaluating to an integer and is incremented on each use. It is supported by Microsoft compiler [49], and is scheduled to be included in version 4.3 of g++ (pending approval on the GCC steering committee [64].) Using this macro ensures that each field has a constant identifier and thus every relation has a unique representation.

Without the `__COUNTER__` macro, AraRat must resort to using the standard `__LINE__` macro as an alternative means for imposing a total order on fields. The limitation placed on the user is that all fields must be defined on different lines in the same source file.

7. Structural typing in AraRat

7.1. Structural equivalence

Relational algebra puts no significance to fields' order: Two schemas are equivalent if they consist of the same set of fields, regardless of the order in which these fields were defined or represented internally. The `TUPLE_T` macro, which generates a record whose field names and types correspond to the fields of the result relation of a given query object, should reflect this equivalence relation. Consider again the AraRat statements

```
TUPLE_T(DIVISION * DEPARTMENT) x;
TUPLE_T(DEPARTMENT * DIVISION) y;
```

which define variables `x` and `y` to be of a record whose field names and types are the union of the field sets of `DIVISION` and `DEPARTMENT`. Macro `TUPLE_T` must make variables `x` and `y` assignment compatible, despite the nominal nature of definition of record types, that is **struct** and **class**, in C++. It is not sufficient that the type of variable `x` is a record with the correct set of fields and their types; it is required that unlike what is shown in Fig. 2.2, this record definition should obey structural typing rules.

Recall that each query object has a compile-time representation of the set of fields. The template based macro `TUPLE_T` iterates over that set of its parameter, generating an inheritance chain, with one class in this chain for each field. So, if the fields set contains three fields, an `ID` of type **int**, a `NAME` of type `string`, and a `ZIP` code of type **int**, then the resulting inheritance chain produced by Rat has four **struct** types, whose outline is as in Fig. 7.1.

We see in the figure, that type `I0` is the basis of the chain, containing no fields, while types `I1`, `I2` and `I3` add fields `ID`, `NAME` and `ZIP` in order. The type `I3` at the end of the inheritance chain is a record with all the required fields.

The actual invocation of the `TUPLE_T` macro indeed generates types similar to `I0`, `I1`, `I2`, and `I3` as outlined in Fig. 7.1, except that these types do not carry names. Instead, the macro makes use of a class template type which inherits from

11 www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/.

Fig. 7.1 An inheritance chain composing a record of fields.

```
struct I0 {
   // empty
};

struct I1: public I0 {
   int ID;
};

struct I2: public I1 {
   string NAME;
};

struct I3: public I2 {
   int ZIP;
};
```

Fig. 7.2 An inheritance lattice composing a record of every subset of fields.

```
struct TOP { };

struct ID: public virtual TOP { int ID; };
struct NAME: public virtual TOP { string name; };
struct ZIP: public virtual TOP { int ZIP; };

struct ID_ZIP: public virtual ID, public virtual ZIP {};
struct ID_NAME: public virtual ID, public virtual NAME {};
struct ZIP_NAME: public virtual ZIP, public virtual NAME {};

struct X: public virtual ID_ZIP, public virtual ID_NAME {};

struct ID_ZIP_NAME: public virtual X, public virtual ZIP_NAME {};
```

one of its arguments, to generate each of the types in the inheritance chain. Each of the types in this inheritance chain is an anonymous instantiation of this template. The total order imposed on field types is used to ensure that the fields are sorted, and that the inheritance chain is always created in the same order. This, together with the observation that template instantiation is structural, guarantees the correct working of `TUPLE_T`, and that variables `x` and `y` are of the same type.

7.2. Structural subtyping

A natural question to ask at this point is whether one can use template instantiation to emulate not only structural equivalence, but also structural subtyping. Say that the set of fields in a query object `q1` is contained in the set of fields of query object `q2`, and that two types `T1` and `T2` are defined by

```
typedef TUPLE_T(q1) T1;
typedef TUPLE_T(q2) T2;
```

Then, the question is whether type `T1` can be defined in such a manner that values of type `T2` can be assigned to it, while omitting the spurious fields? A more demanding desire is that type `T1` is a super type of `T2`. The technique used for ensuring structural equivalence does not achieve any of these ends is as follows: An inheritance chain containing fields `ID` and `NAME` in that order will generate a super type of the type `I3` as defined in Fig. 7.1, but, an inheritance chain for (say) the field `ID` and `ZIP` will not be a super type of `I3`.

As it turns out, it is possible to generalize the inheritance chain idea to what may be called an "*inheritance lattice*", in such a way that subtyping is preserved. The way this may be achieved is demonstrated in Fig. 7.2.

In the figure, we see a **`struct`** for every subset of the three fields: `ID`, `NAME` and `ZIP`. For example, the type `ID_NAME` is a record with two fields `ID` and `NAME`. Moreover, every such record inherits, directly or indirectly, from each record containing a subset of its fields. Thus, `ID_ZIP_NAME` inherits from all records with two fields, `ID_ZIP`, `ID_NAME` and `ZIP_NAME`, from all records of one field, `ID`, `NAME`, `ZIP` and from the `TOP`, the record with no fields.

The generation of record types can be done by the usual techniques of template programming, with, as before, templates that inherit from their arguments. A record with n fields should inherit from all n record types representing subsets with $n-1$ fields. Indirect inheritance may be used to restrict inheritance fan out, and for using only templates with fixed number of arguments. In the figure, instead of having three immediate parents, type `ID_ZIP_NAME` inherits from two types, `ZIP_NAME` and an intermediate type `X` which in turn inherits from `ID_ZIP` and `ID_NAME`.

Fig. 7.3 Constructor templates emulating structural subtyping.

```
1   class VOID { // An auxiliary class, common to all record types, and used by the enable_if macro
2      typedef void *POINTER;
3   };

5   class T1: public ... { // A record containing a certain set of fields using an inheritance chain
6      public: // ...
7         template <class T2> // define a constructor template
8            T1(
9               const T2 &from, // real argument
10              typename // dummy argument type
11                 enable_if< is_subtype<T2, T1>, VOID>::POINTER dummy = (void *)0
12           )
13           {
14              // constructor body ...
15           }
16  }
17 )
```

The solution demonstrated in Fig. 7.2 is exponential. If a database uses n fields, then the number of principal types so generated is 2^n. Auxiliary types, such as the type `X` in Fig. 7.2 even add to this number. The impact on compilation time should be evident. Should this be a concern, one may resort to a method of approximating structural subtyping by using conversion operators instead of the explicit inheritance lattice.

This approach, pioneered by the work of Solodkyy, Jarvi and Mlaih [57], is based on overloading of the assignment operator, or of the type cast operator. This overload is done in such a way that an assignment in the correct direction is possible between every two types that stand in a structural subtyping relation. The clever observation made by Solodkyy, Jarvi and Mlaih is that one can use a conditional template expansion, (specifically the `enable_if` templates [33]) to test first whether two types stand in a subtyping relation, and only generate the conversion code if this is the case. To do this correctly, each record T_1 type defines a *constructor template*, which takes a type argument T_2, and a constructor argument of type T_2. Now, the participation of instantiation of this constructor template in the overloading resolution tournament is guarded by an `enable_if` template, as outlined by Fig. 7.3.

Class `VOID`, defined in Lines 1–3 in the figure is an auxiliary class, used by all record types. The `POINTER` **`typedef`** definition within that class is such that `VOID::POINTER` names the type **`void`** `*`.

Examine now the type `T1`, defined in Lines 5–17. In this type, there is a constructor template which given a type `T2`, generates a constructor that takes an argument of type `T2` and creates a `T1` object out of it. The `enable_if` templates is then used to restrict the generation of this constructor. If the compile-time condition `is_subtype<T2, T1>` evaluates to the boolean constant **`false`**, `enable_if` does not generate a valid type; in this case, the second argument to this constructor has no valid type, and the substitution-failure-is-not-an-error principle [66] prevents the constructor from being generated. If however, `is_subtype<T2, T1>` evaluates to **`true`**, then `dummy`, the second argument, receives the type `VOID:POINTER`, i.e., **`void`** `*`, the constructor template is then activated. Note that this second argument receives a default value of `0`. Therefore, the constructor call `T1(t2)` where `t2` is an instance of `T2` assigns this value to `dummy`, and compiles correctly. Moreover, if a C++ function expects an argument of type `T1`, and the actual argument is of a type `T2` which happens to be a structural subtype of `T1`, then the compiler will insert the appropriate constructor call. The said function will then appear to be polymorphically applicable to all structural subtypes of `T1`.

This technique does not carry the exponential price tag. But, it should be remembered that it is only an imitation of true subtype. It would fail, for example, if subtyping is examined through runtime type information, or if a sequence of conversions is required.

8. Discussion and further research

The AraRat system demonstrates a seamless integration of the relational algebra formalism with C++, which can be used to generate safe SQL queries (using method `asSQL()`). Also, AraRat makes it possible to generate a record 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 Fig. 3.1 is shorter than that of Fig. 6.1. The compilation of the program in Fig. 3.1 took 1.04 s while the program in Fig. 6.1, which uses the Rat library that consists of about 3000 lines of C++ code, compiled in 1.26 s.[12] These time differences do not constitute 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

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

Fig. 8.1 A possible ARARAT extension supporting dynamic conditions.

```
1   DEF_DYNAMIC_COND(c, EMPLOYEE, TRUE);
2   if (dept > 0);
3      c &= (DEPTNUM == dept);
4   if (first != null)
5      c &= (FIRST_N == first)
6   ...
```

of queries. The benchmark should measure both the construction and execution time of queries. We venture to project and extrapolate from the measurements we made that the finding will be that construction 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 limited to modifications of a query object by applying selection, union and subtraction to it. It is mundane to add support to dynamically created conditions, allowing to define a selection condition with code such as shown in Fig. 8.1.

Line 1 in the figure uses the statement `DEF_DYNAMIC_COND(c, EMPLOYEE, TRUE);` to encode the symbol table of `EMPLOYEE` into `c`. Then, in lines 2–5 the condition is refined.

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 Section 6.4, RAT can easily generate 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 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 6 we described non-trivial implementation techniques, by which both compile-time and runtime values are used for realization of the embedded language. 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. 4.2(b) and Fig. 4.2(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. 5.2.

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.

In a similar manner, this work does not deal with the many different variants of the SQL language. It is well known that commercial vendors introduce their own (some may say idiosyncratic) dialect in adoption of the SQL standard. Simple statements will be understood by all such implementations. However, realization of more advanced features, such as nested queries may be very different in different database implementation. The procedure of translating the content of a query object into an SQL statement must be cognizant of these differences. This demand can be compared to the requirement that an implementation of a high-level programming language, specifically a compiler, must be aware of the target language. Writers of multi-targets compilers address this difficulty by producing object code in intermediate form, which is then translated to the specific machine code. The same idea applies to ARARAT: the internal form of a query object is an arbitrarily complex expression in relational algebra. The familiar post order traversal of such an expression yields an evaluation procedure whose elements are simple SQL queries, each of which operates on previously computed intermediate results, yielding another such result. The final such operation computes the entire query result. This process lends itself to a general purpose process of translation into an arbitrary SQL implementation. The cost is of course is in that less-sophisticated database implementation may miss optimization opportunities, when given such a sequence of operations. A specialized translator may be in place for such implementations.

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

Finally, note that work on integration of SQL with other languages can be also viewed as part of the generative programming line of research [7,22,34,36,50]. 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* [16] to prove the correctness of Rat.

Acknowledgements

The supportive work of Marina Rahmatulin, even on the eve of her wedding day, is gratefully acknowledged. We thank an anonymous reviewer for drawing our attention to the "single instantiation rule" perspective.

References

[1] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, in: C++ in Depth Series, Addison-Wesley, 2004.
[2] T. Andrews, C. Harris, Combining language and database advances in an object-oriented development environment, in: Meyrowitz [47], pp. 430–440.
[3] K. Arnold, J. Gosling, The Java Programming Language, in: The Java Series, Addison-Wesley, Reading, MA, 1996.
[4] M.P. Atkinson, O.P. Buneman, Types and persistence in database programming languages, ACM Comput. Surv. 19 (2) (1987) 105–170.
[5] M.P. Atkinson, R. Welland, Fully Integrated Data Env.: Persistent Prog. Lang., Object Stores, and Prog. Env., Springer, Secaucus, NJ, USA, 2000.
[6] M.H. Austern, Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley, 1998.
[7] D.S. Batory, C. Consel, W. Taha (Eds.), Proc. of the 1st Conf. on Generative Prog. and Component Eng., in: LNCS, vol. 2487, Springer, 2002.
[8] J. Bentley, Programming pearls: Little languages, Commun. ACM 29 (8) (1986) 711–721.
[9] T. Bloom, S.B. Zdonik, Issues in the design of object-oriented database programming languages, in: Meyrowitz [47], pp. 441–451.
[10] M. Böhme, B. Manthey, The computational power of compiling C++, Bull. EATCS 81 (2003) 264–270.
[11] A.S. Christensen, A. Møller, M.I. Schwartzbach, Precise analysis of string expressions, in: Proc. of the 10th International Static Analysis Symposium, SAS'03, in: LNCS, vol. 2694, Springer, 2003, pp. 1–18.
[12] E.F. Codd, A relational model of data for large shared data banks, Commun. ACM 13 (6) (1970) 377–387.
[13] W.R. Cook, S. Rai, Safe query objects: Statically typed objects as remotely executable queries, in: Roman et al. [53], pp. 97–106.
[14] G. Copeland, D. Maier, Making Smalltalk a database system, SIGMOD Rec. 14 (2) (1984) 316–325.
[15] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[16] E. Denney, B. Fischer, Certifiable program generation, in: Glück and Lowry [22], pp. 17–28.
[17] J.E. Donahue, Integrating programming languages with database systems, in: Data Types and Persistence (Appin), Informal Proceedings, 1985, pp. 315–324.
[18] A. Eisenberg, J. Melton, SQLJ Part 1: SQL routines using the Java programming language, SIGMOD Rec. 28 (4) (1999) 58–63.
[19] R. M.S., et al., Using GCC: The GNU Compiler Collection Reference Manual for GCC 3.3.1, Gnu Press, 2003.
[20] J. Gil, Z. Gutterman, Compile time symbolic derivation with C++ templates, in: Proc. of the USENIX C++ Conf., USENIX Association, Santa Fe, New Mexico, 1998, pp. 249–264.
[21] J. Gil, I. Maman, Micro patterns in Java code, in: R. Johnson, R.P. Gabriel (Eds.), Proc. of the 20th Ann. Conf. on OO Prog. Sys., Lang., & Appl., OOPSLA'05, San Diego, CA, ACM SIGPLAN Not. (2005) 97–116.
[22] R. Glück, M.R. Lowry (Eds.), Proc. of the 4th Conf. on Generative Prog. and Component Eng., in: LNCS, vol. 3676, Springer, 2005.
[23] A. Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, Reading, MA, 1984.
[24] C. Gould, Z. Su, P.T. Devanbu, Static checking of dynamically generated queries in database applications, in: Proc. of the 26th Int. Conf. on Soft. Eng., ICSE'04, May 23–28, 2004, IEEE Computer Society Press, Edinburgh, Scotland, United Kingdom, 2004, pp. 645–654.
[25] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G.D. Reis, A. Lumsdaine, Concepts: First-class language support for generic programming in C++, in: Tarr and Cook [62].
[26] J.R. Groff, P.N. Weinberg, SQL, the Complete Reference, Osborne, McGraw-Hill, 1999.
[27] Z. Gutterman, Symbolic pre-computation for numerical applications. Master's Thesis, Technion, 2004.
[28] W.G.J. Halfond, A. Orso, AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks, in: Proc. of the 20th IEEE/ACM international Conference on Automated Software Engineering, ACM Press, New York, NY, USA, 2005, pp. 174–183.
[29] M. Harren, B. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, R. Bordawekar, XJ: Integration of XML processing into Java, 2003.
[30] A. Hejlsberg, S. Wiltamuth, P. Golde, The C# Programming Language, 2nd ed, Addison-Wesley, Reading, MA, 2003.
[31] M. Howard, D. Leblanc, Writing Secure Code, Microsoft Press, Redmond, WA, USA, 2001.
[32] J. Järvi, G. Powell, A. Lumsdaine, The lambda library: Unnamed functions in C++, Soft. Pract. Exp. 33 (3) (2003) 259–291.
[33] J. Järvi, J. Willcock, H. Hinnant, A. Lumsdaine, Function overloading based on arbitrary properties of types, C/C++ Users J. 21 (6) (2003) 25–32.
[34] S. Jarzabek, D.C. Schmidt, T.L. Veldhuizen (Eds.), Proc. of the 5th Conf. on Generative Prog. and Component Eng., in: LNCS, vol. 3676, ACM Press, 2006.
[35] S.P. Jones, Haskell 98 Language and Libraries: The Revisited Report, Cambridge University Press, 2003.
[36] G. Karsai, E. Visser (Eds.), Proc. of the 3rd Conf. on Generative Prog. and Component Eng., in: LNCS, vol. 3286, Springer, 2004.
[37] B.W. Kernighan, D.M. Ritchie, The C Programming Language, 2nd ed, in: Software Series, Prentice-Hall, 1988.
[38] G. Koch, K. Loney, Oracle: The Complete Reference, Electronic Edition, Osborne, 1997.
[39] D. Leijen, E. Meijer, Domain specific embedded compilers, in: Proc. of the 2nd USENIX Conference on Domain-Specific Languages'99, vol. 35, ACM Press, 1999, pp. 109–122.
[40] D. Malayeri, J. Aldrich, Combining structural subtyping and external dispatch, in: OOPSLA'07: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications, ACM Press, New York, NY, USA, 2007, pp. 789–790.
[41] R.A. McClure, I.H. Krüger, SQL DOM: Compile time checking of dynamic SQL statements, in: Roman et al. [53], pp. 88–96.
[42] B. McGehee, Using Microsoft SQL Server 7.0. Que, 1999.
[43] B. McNamara, Y. Smaragdakis, Static interfaces in C++, in: 1st Workshop on C++ Template Programming, Erfurt, Germany, Oct. 2000.
[44] E. Meijer, B. Beckman, G. Bierman, LINQ: Reconciling objects, relations and XML in the NET framework, in: Proc. of the ACM SIGMOD Int. Conf. on Management of Data, ICMD'2006, Chicago, IL, 2006.

[45] E. Meijer, W. Schulte, G. Bierman, Programming with circles, triangles and rectangles, in: In XML Conference and Exposition, 2003.
[46] R.B. Melnyk, P.C. Zikopoulos, DB2: The Complete Reference, McGraw-Hill Companies, 2001.
[47] N.K. Meyrowitz (Ed.), Proc. of the 2nd Ann. Conf. on OO Prog. Sys., Lang., & Appl. OOPSLA'87, Orlando, FL, Oct. 4–8, 1987, ACM SIGPLAN Not. 22 (12) (1987).
[48] N.C. Myers, A new and useful template technique: Traits, C++ Rep. 7 (5) (1995) 32–35.
[49] C. Pappas, W.H. Murray, Visual C++ Net: The Complete Reference, McGraw-Hill Companies, 2002.
[50] F. Pfenning, Y. Smaragdakis (Eds.), Proc. of the 2nd Conf. on Generative Prog. and Component Eng., in: LNCS, vol. 2830, Springer, 2003.
[51] B.C. Pierce, Types and Programming Languages, MIT Press, 2002.
[52] P.J. Plauger, The standard template library, C/C++ Users J. 13 (12) (1995) 10–20.
[53] G.-C. Roman, W.G. Griswold, B. Nuseibeh (Eds.), Proc. of the 27th Int. Conf. on Soft. Eng., ICSE'05, May 15–21, 2005, ACM Press, New York, NY, USA, 2005, pp. 15–21.
[54] J.W. Schmidt, Some high level language constructs for data of type relation, ACM Trans. Database Syst. 2 (3) (1977) 247–261.
[55] J. Siek, A. Lumsdaine, Concept checking: Binding parametric polymorphism in C++, in: Proc. of the 1st Workshop on C++ Template Programming, Erfurt, Germany, 2000.
[56] P. Smith, Applied Data Structures with C++, Jones & Bartlett, 2004.
[57] Y. Solodkyy, J. Järvi, E. Mlaih, Extending type systems in a library — type-safe XML processing in C++, in: Workshop of Library-Centric Software Design at OOPSLA'06, Portland, OR, Oct. 2006.
[58] A. Stevens, C. Walnum, Standard C++ Bible, Wiley, 2000.
[59] B. Stroustrup, The Design and Evolution of C++, Addison-Wesley, Reading, MA, 1994.
[60] B. Stroustrup, The C++ Programming Language, 3rd ed, Addison-Wesley, Reading MA, 1997.
[61] V. Surazhsky, J.Y. Gil, Type-safe covariance in C++, in: H.M. Haddad, G.A. Papadopoulos, A. Omicini, R. Wainwright, L. Liebrock, M. Palakal, A. Andreou, C. Pattichis (Eds.), Proc. of the 19th Ann. ACM Symp. on Applied Comp. SAC'04, ACM Press, Nicosia, Cyprus, 2004, pp. 1496–1502.
[62] P.L. Tarr, W.R. Cook (Eds.), Proc. of the 21st Ann. Conf. on OO Prog. Sys., Lang., & Appl., OOPSLA'06, Portland, OR, Oct. 22–26, 2006, ACM SIGPLAN Not. (2006).
[63] M. Tatsubori, S. Chiba, K. Itano, M.-O. Killijian, OpenJava: A class-based macro system for Java, in: W. Cazzola, R.J. Stroud, F. Tisato (Eds.), Proc. of the 1st OOPSLA Workshop on Reflection and Software Engineering, OOPSLA'99, in: LNCS, vol. 1826, Springer, Denver, CO, USA, 1999, pp. 117–133.
[64] I.L. Taylor, private communication, Mar. 2007.
[65] Z.D. Umrigar, Fully static dimensional analysis with C++, SIGPLAN Not. 29 (9) (1994) 135–139.
[66] D. Vandevoorde, N.M. Josuttis, C++ Templates: The Complete Guide, Addison-Wesley, 2002.
[67] T.L. Veldhuizen, Expression templates, C++ Rep. 7 (5) (1995) 26–31.
[68] N. Welsh, F. Solsona, I. Glover, SchemeUnit and SchemeQL: Two little languages, in: Workshop on Scheme and Functional Programming, London, UK, Oct. 2002.
[69] M. Widenius, D. Axmark, MySQL Reference Manual, O'Reilly, 2004.
[70] J. Willcock, J.G. Siek, A. Lumsdaine, Caramel: A concept representation system for generic programming, in: 2nd Workshop on C++ Template Programming, Tampa, FL, Oct. 2001.
[71] N. Wirth, The programming language Pascal, Acta Inf. 1 (1971) 35–63.
[72] S.N. Woodfield, The impedance mismatch between conceptual models and implementation environments, in: Proceedings of the ER'97 Workshop on Behavioral Models and Design Transformations, 1997.
[73] E.V. Wyk, L. Krishnan, D. Bodin, E. Johnson, Adding domain-specific and general purpose language features to Java with the Java language extender, in: Tarr and Cook [62], pp. 728–729.
[74] Z. Yao, Q. long Zheng, G.-L. Chen, AOP++: A generic aspect-oriented programming framework in C++, in: Glück and Lowry [22], pp. 94–108.