Making Standard ML a Practical Database Programming Language *

Atsushi Ohori Katsuhiro Ueno
Research Institute of Electrical Communication
Tohoku University
{ohori, katsu}@riec.tohoku.ac.jp

Abstract

Integrating a database query language into a programming language is becoming increasingly important in recently emerging high-level cloud computing and other applications, where efficient and sophisticated data manipulation is required during computation. This paper reports on seamless integration of SQL into SML# – an extension of Standard ML. In the integrated language, the type system always infers a principal type for any type consistent SQL expression. This makes SQL queries first-class citizens, which can be freely combined with any other language constructs definable in Standard ML. For a program involving SQL queries, the compiler separates SQL queries and delegates their evaluation to a database server, e.g. PostgreSQL or MySQL in the currently implemented version.

The type system of our language is largely based on Machiavelli, which demonstrates that ML with record polymorphism can represent type structure of SQL. In order to develop a practical language, however, a number of technical challenges have to be overcome, including static enforcement of server connection consistency, proper treatment of overloaded SQL primitives, query compilation, and runtime connection management. This paper describes the necessary extensions to the type system and compilation, and reports on the details of its implementation.

Categories and Subject Descriptors H.2.3 [*Database Management*]: Languages—Database (persistent) programming languages

General Terms Design, Languages

Keywords SQL, Polymorphism, Type System, Interoperability, SML#

1. Introduction

Smooth integration of a database query language into a programming language is essential for any high-level applications that require efficient processing of large amount of data. The need of this integration will become particularly important in recently emerging high-level cloud computing, where sophisticated data manipulation beyond simple data aggregation will be required. Query language integration will also make currently wide-spread Web programming through various meta-level "framework" such as Ruby on Rails much more reliable.

The need of proper integration of a database query language and a programming language has been recognized and investigated in database programming community for more than two decades. See [4] for an early survey in this area. Since then a number of query language bindings and embeddings in general purpose programming languages have been proposed and implemented. Examples include SQLJ [13] and more recent LINQ [23] and Ferry [16] middle-ware. Compared with SQL command string interface to database servers, these database binding frameworks certainly increase flexibility and ease of use in accessing databases from programming languages. However, they still do not achieve seamless integration of a database query language into a host language type system. Existing database bindings either restrict full functionality and generality provided by their target database servers, or they do not treat database queries as first-class citizens in the type systems of the host programming languages.

There have also been a number of proposals for high-level database programming languages that integrate declarative queries in their type systems. Some notable examples include GemStone (OPAL) [11], Machiavelli [30], O_2 [19], and Fibonacci [2] to name a few. While these approaches provide uniform integration of database queries in a programming language, it remains to be seen whether or not some of these approaches would become a practical alternative to SQL. Based on long and intensive research and development of the relational data model [8], relational database systems have realized highly optimized data manipulation for large amount of data with various practical supports such as transaction management and network access. All of these features are made available through SQL, which is the well established and standardized query language for relational database systems. Perhaps due to this reason, most of applications that require both efficient database manipulation and programming still use general purpose programming languages such as C or Java with SQL command string interface to a relational database server.

This situation indicates that an "impedance mismatch" [22] between a database system and a programming language still exists *in practice*.

To eliminate this mismatch, we must develop a high-level programming language that seamlessly integrates SQL in such a way that SQL queries are first-class citizens in the language type system and that they are evaluated by a practical database server. This paper reports on one such integration, where SQL is seamlessly integrated in SML# [32], an extension of Standard ML [24]. In our

* The authors were partially supported by the Japan Society for the Promotion of Science Grant-in-Aid for Basic Research (C) grant no. 22500023.

development, we adopt polymorphic typing of Machiavelli [30], where the authors suggested that the programming language ML can be extended with an SQL-style declarative query language. The language proposed in [30] is, however, conceptual one focusing only on typing issues; no implementation strategy is considered or developed. As outlined in the next section, a number of technical challenges have to be overcome before a practical language can become a reality. In the present paper, we have solved those problems and have developed a compiler for SML# extended with SQL. In this extended language, SQL queries are first-class expressions directly definable in the language type system, and are evaluated by an external database server. Thanks to this feature, Standard ML programmers can readily enjoy efficient and practical database programming in their Standard ML code.

In addition to solving technical problems in language design and typing, we have also developed an implementation technique for connecting static semantics of SQL in the host language type system, and its dynamic semantics realized by an external database server. This should be useful for extending a typed higher-order language with various domain specific languages, not restricted to SQL.

All the results reported in this paper has been implemented in our SML# compiler. Most of them have already been made available in SML# version 0.60 or later releases [32].

1.1 Related works

Before presenting our technical development, we compare our approach with some more related works.

The conventional and still widely used approach to database programming is to construct SQL command strings directly and send them to a database server using low-level database server interfaces such as SQL/CLI and JDBC. Embedded SQL such as ECPG of PostgreSQL and SQLJ [13] supports a form of macro SQL statements that are expanded by a preprocessor to those low-level database library calls. Both of them are unsatisfactory in many ways; type-checking is far from adequate, and the interaction with the host language construct is rather limited. In order to improve type safety and host language interface of preprocessor-based approach, several domain specific embedded languages [18] have been proposed and implemented. HaskellDB [20] implements a set of monadic term constructors corresponding to relational operators. ARARAT [14] provides C++ template for composing SQL queries. Ur/Web [7] provides a meta-programming framework for the programmer to generate type-consistent SQL queries through type computation. LINQ [23] and Ferry [16] attempt to provide a language independent layer for querying various data structures including relational database. DHS [15] implements a combinator library for generic manipulation of collection types that are to be compiled to Ferry. While these approaches provide high-level database access, they do not fully achieve seamless integration of SQL into a general purpose programming language to the extent we wish to achieve.

We note that SQL is an algebraic language based on the relational algebra. In this language, relation-valued expressions including `SELECT` expressions can be used to construct another query expression. This properly makes SQL an elegant and powerful data algebra for relations. Seamlessly integrating SQL in a programming language must at least mean that relations are first-class citizens and their operations are freely combined with other language constructs in the host programming language. This is a common-sense knowledge in functional programming. For example, consider the case of integrating an algebra of *lists* in ML. Everyone unanimously expects that lists are first-class citizens and associated operators to construct and manipulate lists can be freely combined with any other language constructs. SQL should be given the same status, without scarifying its features specified in the standardized language definition and its efficiency realized by a matured external database server. Unfortunately, however, there does not seem to exist any implemented general purpose programming language that achieves this status for SQL. This is the property we attempt to achieve in SML#.

Since relations are well-defined mathematical objects, it should be possible to extend a type theory of programming languages with data algebra on relations. Based on this general observation, several database programming languages have been proposed, including GemStone [11], Machiavelli [30], Napier88 [26], Iris [3], Fibonacci [2], and Kleisli [34]. All of those languages, by their construction, seamlessly integrate advanced database queries in their type systems and evaluation models. See [33] for various typing issues in database queries, and [21] for a survey on various approaches to database and language integration. However, it is non-trivial to make those languages practical database programming languages that can handle hundreds of millions of tuples efficiently. Compared with the high maturity of optimization and implementation techniques of relational database systems, optimization and implementation techniques for advanced data models realized in those new database programming languages do not seem to have been well developed.

The goal of the present paper is to make the SQL itself seamlessly available in a practical general purpose programming language. For this purpose, we must first extend the type system of the host programming language so that SQL statements can be representable as typed expressions. An extension of ML type inference for generalized relational algebra presented in [29] is perhaps the first such example. This type system and its refinement [28] were the basis of the type system of Machiavelli [6, 30]. We design our type system largely based on this approach. However, in this paper, instead of uniformly integrating generalized relational algebra in its type system and evaluation model, we develop a type system that precisely represents polymorphic typing of the SQL language itself as defined in its language standard, and develop a compilation method to delegate their evaluation to an external database server. This requires careful design and development beyond the language design reported in [6, 30].

Another approach to integrate database queries and programming constructs is to compile high-level expressions into SQL [9]. Links [10] takes this approach in designing a Web programming system. The type system sketched in [10] appears to be based on the observation similar to [6, 30] – it uses sorted row variables [31] instead of kinded typing [28, 29], but these two formalisms on record polymorphism are equivalent for typing query expressions. Links compiles relation-valued expressions, possibly involving function applications and other language constructs, into SQL. The resulting system achieves integration of database queries and programming language constructs, but this is only to the extent that the language constructs can be compiled to SQL commands. While this compile-to-SQL approach would be useful for some target areas such as Web programming, it is not obvious that this approach scales up to large and complex application development that requires a general purpose programming language and the full functionality of SQL.

We note that any practical general purpose programming language involves a number of advanced features to form a complex system whose semantics can (at least under the current state-of-the-art) only be realized by a sophisticated compiler. This is true for Standard ML. SML# compiler, for example, contains more than 0.3 million lines of code. The SQL language itself also contains various features such as grouping and duplicate controls in `SELECT` lists, and transaction management in data manipulation. All these features are indispensable for serious application development. Our major technical contribution is to design a type system that can

precisely represent the SQL language in Standard ML, to develop a compilation method that seamlessly combines the dynamic semantics of Standard ML expressions realized by a Standard ML compiler and the semantics of SQL queries realized by an external database server, and to implement it in the SML# compiler that compiles the full set of Standard ML.

1.2 Paper organization

The rest of this paper is organized as follows. Section 2 discusses the problems to be overcome and outlines our strategy. Section 3 describes the language and its type system. Section 4 describes the details of its implementation. Section 5 concludes the paper with suggestions for further investigations.

2. Problems and our strategy

We have chosen Standard ML as the base language for our database extension. This is not accidental. In this section, we review the requirements and problems in achieving our goal of seamless integration, and outline our development.

2.1 Expressions and typing of SQL

We begin by reviewing the basic properties of SQL in the perspective of programming languages.

SQL can be characterized as a *impure functional language*. Its main components, i.,e. `SELECT` statements, are relation-valued expressions that can be freely composed as far as each sub-phrase is type consistent. For example, a query such as

```
SELECT name, age
FROM people                    (Q)
WHERE age >= 25
```

is an expression denoting a set of tuples of `name` and `age` and can be used as a source relation in any other queries. SQL programming consists of composing those relation-valued expressions. In addition to those functional expressions, SQL also contains imperative features such as those for updating the database state and transaction management, which assumes sequential evaluation. To represent those expressions, eager functional programming is the most appropriate framework. Moreover, since SQL programming involves manipulation of record structures across multiple tables, it is essential to enforce type discipline that statically checks complex queries before sending them to a remote database server. As observed in [30], SQL expressions are inherently polymorphic in the structures they manipulates. For example, the above very simple query is polymorphic in several ways: this query can be issued against any database that contains at least a `people` table having at least `name` field of any type and `age` field of numerical type. This indicate that a language containing SQL should have a polymorphic type system preferably with type inference.

Base on these observations, a basic strategy of extending ML to represent SQL-style database queries was proposed in [29, 30]. In their approach, a query expression is considered as a *polymorphic function* acting on an appropriate structure composed of labeled record and set data types. For example, the query Q above would be given a polymorphic type of the following form (in the notations we use in this paper):

```
Q : ['a,'b#{name:'a, age:int},'c#{people:'b}.
     'c db -> 'b relation]
```

This is the type `'c db -> 'b relation` in prenex form where the set of bound type variables are explicitly listed in its prefix `'a,'b#{name:'a, age:int},'c#{people:'b}`. Notation `'b#{name:'a, age:int}` represents a type variable `'b` with a *kind* constraint `#{name:'a, age:int}` indicating the fact that any instance of `'b` must be a labeled record (tuple) type that contains at least `name:'a` and `age:int` fields. By this polymorphic typing with record kind constraints, their approach represents the precise polymorphic nature of Q, i.e. it is a query that can be evaluated against any database that contains at least a `people` table each of whose tuples contains at least integer-valued `age` column and `name` column of any type.

We roughly follow this approach. Among the ML family of languages, we consider Standard ML the most appropriate one for the following reasons.

- Its eager functional semantics with imperative features make an ideal framework for integrating relation-valued query expressions with imperative features such as transactions.
- Its type system and operational semantics has been rigorously specified and their details are well documented [24].
- The type theory and implementation method for extending Standard ML with record polymorphism has been well established [28, 31].

We base our development on SML#, which is an extension of Standard ML with record polymorphism [28] and interoperability with C language. These two features provide sufficient basis for seamless extension of Standard ML with SQL.

2.2 The remaining problems

The solutions so far proposed in the literature are, however, only partial for extending Standard ML with SQL. There are a number of problems that have to be worked out before realizing a practical language that achieves truly seamless integration of SQL. Among them, the following are major technical challenges.

1. *Extensions of the type system.* Record polymorphism is useful in achieving the integration but adding this alone to Standard ML type system does not yield a type system for SQL. We need to construct typing mechanisms for the following.
   (a) *Server typing and type checking.* We need to develop a typing mechanism to declare a database schema on a database server, and to type check its correctness against actual contents of the server.
   (b) *Server connection typing and its consistency enforcement.* Since an SQL command needs to be sent to a database server, all the query components consisting of the command must only reference to a unique server connection. With higher-order functions, enforcing this requirement turns out to be a subtle typing problem.
   (c) *A proper treatment of null values.* Since a column may contain null values, we cannot simply infer a typing of the form `'a::{age:int}` for expression `age >= 25`.
   (d) *Overloaded operator and constants.* Literals such as `25` above play double roles; one for a constant of the host language, and one that denotes the same constant in a remote database server. These two have different representations. Even worse, `25` in an SQL query represents not only an integer but also a floating-point number. In addition, operator such as `>` for comparison is overloaded with several atomic types. In Standard ML definition, overloading is resolved statically. If we simply adopt the same strategy, then the flexibility of SQL expressions would be severely limited. Again, developing a precise typing mechanism to treat these overloading requires delicate development.
2. *Compilation and evaluation.* Expressions in a functional language is compiled to a code and executed with a runtime system with a specialized memory management. It is relatively

straightforward to add compilation rules for their own query constructs and execute them in their own runtime system. For this purpose, it is sufficient to define a data structure for tuples and relations and evaluation strategy for each primitive operations in the relational algebra. Newly designed database programming languages usually adopt this strategy. Our purpose is different. Instead of developing a query execution engine inside of the language runtime system, we need to call an existing SQL database server. For this purpose, we need to develop both a compilation method and a runtime mechanism for delegating evaluation of SQL query expressions to a remote database server. Since Standard ML is a higher-order functional language, seamless integration of SQL implies that SQL queries are intermixed with function definitions and function applications. This requires us to carefully design the compilation algorithm so that it separates SQL query expressions appearing in a given expression.

2.3 Summary of our development

We have worked out these problems, have developed compilation algorithms and have implemented a new generation of Standard ML that seamlessly integrates SQL by extending our SML# compiler. The extension to Standard ML syntax is carefully designed so that it maintains backward compatibility with the Definition of Standard ML. In our development, we have used PostgreSQL and MySQL for our target database servers, considering their wide availability; other DBMS should equally be possible, as described later in section 4.6.

In SML# with our extension, the query Q is written as follows.

```
val Q =
  _sql db => select #person.name as name,
                    #person.age as age
             from #db.people as person
             where SQL.>= (#person.age, 25)
```

where `SQL.>=` is the greater-or-equal comparison for SQL values. One can see that the above expression is a direct representation of (a verbose version of) the query Q. Similar expressions may be possible in some programming languages with embedded SQL commands, which are translated to a sequence of primitive statements. Different from those syntactic shorthands, this is an expression in the language. The extended SML# compiler type checks this and infers the following polymorphic type.

```
val Q = fn
: ['a#{people: 'b},
   'b#{age: int, name: 'd},
   'c,
   'd::{int, word, char, string, real, 'e option},
   'e::{int, word, char, bool, string, real}.
   ('a, 'c) SQL.db
      -> {age:int, name:'d} SQL.query]
```

This type indicates that Q is a function that takes a database of type `('a, 'c) SQL.db` whose contents is of type `'a#{people:'b}`, and returns a query of type `{age:int,name:'d}`. The extra parameter `'c` to a database type `'a#{people:'b}` is there to ensure the consistent usage of connection and will be explained later. `'d::{···}` constrains that `'d` can be instantiated only with a type shared among ML and SQL. It should be intuitively clear that the inferred type represents the precise polymorphic nature of the query Q, and indeed it is a principal type for Q.

In an ML-style type system, the fact that a principal type is inferred for this expression immediately means that this expression can be freely combined with any other language constructs as far as its usage is type consistent. Figure 1 shows example programs to define a database server, to connect it, to insert tuples, to execute query Q, and to fetch the results as a list of SML# records. It shows an actual interactive session in SML#. The lines starting with prompt "`#`" and ending with delimiter "`;`" are user input, which are followed by a system response.

3. SML# extended with SQL

This section presents the language we develop. We first introduce SML#, and develop necessary extensions in syntax, typing mechanisms, and operational semantics. We then show a programming example demonstrating the benefit of seamless integration of SQL into SML#.

3.1 SML#: Standard ML with record polymorphism and interoperability with C

SML# embodies record polymorphism and infers a polymorphic type for record operations as seen in the following example:

```
# fun getName x = #Name x;
val getName = fn : ['a#{Name:'b}, 'b. 'a -> 'b]
```

where `#`l selects the l field from a record. `getName` takes a record and returns its `Name` field. This record polymorphism provides a basis to extend Standard ML with SQL expressions.

In addition, SML# supports interoperability with the C language through its *natural data representation* [27]. Under this scheme, internal data representations in SML# and C have the following correspondence: atomic types including `int` and `real` are the same as `int` and `double` in C; τ `array` has the same representation as a pointer to an array of τ in C; and a labeled record type $\{l_1:\tau_1, \ldots, l_n:\tau_n\}$ has the same representation as a pointer to a structure in C that contains the elements of types $\{\tau_1, \ldots, \tau_n\}$ in the lexicographical order of their labels $\{l_1, \ldots, l_n\}$. Based on these properties, SML# allows the programmer to declare an external library function defined in C and use it as an ordinary ML function. The following example code imports C library function `sin`:

```
# val sin =
      dlsym (dlopen "/usr/lib/libm.so", "sin")
      : _import real -> real;
val sin = fn : real -> real;
# sin 1.0;
val it = 0.841470984807 : real
```

where `dlopen` and `dlsym` are built-in library functions that open and find a pointer to the named dynamically linked library function. After this declaration, `sin` is used as an ordinary ML expression of type `real -> real`, but application of `sin` directly calls the C library function without any data conversion. In our development, we use this feature to bind database server library functions provided for C.

The syntax of SML# is that of Standard ML. For the explanation purpose in this paper we consider the following subset of its core language.

$$e \quad ::= \quad x \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e\ e \mid \{l_1\texttt{=}e_1, \ldots, l_n\texttt{=}e_n\} \mid \texttt{\#}l\ \ e$$

x ranges over the set of variables. `fn` x `=>` e defines a (first-class) function. $\{l_1\texttt{=}e_1, \ldots, l_n\texttt{=}e_n\}$ constructs a labeled record. Other features are mostly orthogonal to our SQL extension and can be used without any problem. In examples, we also use the following syntax.

- `val` x `=` e binds a variable x to an expression e.
- `SQL.`x is a variable x defined in the module named `SQL`. In our implementation, which we shall describe in Section 4, most

```
# val server =
      _sqlserver "host=127.0.0.1 dbname=test"
      : {people: {name: string, age: int}};
val server = "host=127.0.0.1 dbname=test"
  : {people: {age: int, name: string}} SQL.server

# val db = SQL.connect server;
val db = <conn>
  : {people: {age: int, name: string}} SQL.conn

# val q =
      _sql db =>
           insert into #db.people (name, age)
           values ("Alice", 24);
val q = fn
  : ['a#{people: {age: int, name: string}},
     'b.
     ('a, 'b) SQL.db -> SQL.command]

# val r = _sqlexec q db;
val r = () : unit

# val q =
      _sql db =>
           insert into #db.people (name, age)
           values ("Bob", 25);
val q = fn
  : ['a#{people: {age: int, name: string}},
     'b.
    ('a, 'b) SQL.db -> SQL.command]

# val r = _sqlexec q db;
val r = () : unit

# val Q =
      _sql db =>
           select #person.name as name,
                  #person.age as age
           from #db.people as person
           where SQL.>= (#person.age, 25);
val Q = fn
  : ['a#{people:'b},
     'b#{age:int, name:'d},
     'c,
     'd::{int, word, char, string, real, 'e option},
     'e::{int, word, char, bool, string, real}.
     ('a, 'c) SQL.db
     -> {age: int, name: 'd} SQL.query]

# val r = _sqleval Q db;
val r = <rel> : {age: int, name: string} SQL.rel

# val x = SQL.fetchAll r;
val x = [{age = 25, name = "Bob"}]
  : {age: int, name: string} list
```

Figure 1. Example Program in SML#

resources (functions and types) for the SQL extension are organized into the `SQL` module, and they are referenced through this notation.

Its type system is that of Standard ML extended with record kinds on type variables. We let C range over type constructors such as `list` or `array`. Sometimes we use `option` type constructor to represent values possibly containing null. We use the following syntax for types.

- The set of monomorphic types (ranged over by τ) is given by the following syntax

  $$\tau := b \mid \tau \texttt{ -> } \tau' \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \mid \tau\ C$$

  where b ranges over atomic base types such as `int`; τ `->` τ' is a function type; $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ is a labeled record type; and $\tau\ C$ is a constructor type such as `int list`.
- `'a`, `'b`, ... are type variables (ranged over by t).
- $[t_1 \# k_1, \ldots, t_n \# k_n . \tau]$ is a polymorphic type of τ with bound type variables $t_1, \ldots, t_n$. Each t_i is constrained with a kind k_i. A kind k is either empty indicating no constraint or a record kind $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ denoting all possible records that contain at least the specified fields.

3.2 Syntax extension for SQL

To represent SQL, we need to introduce the following.

- A server definition on which SQL is run.
- SQL "commands". Among them a `SELECT` command is an algebraic (functional) expression that returns a relation. The others are those that change the server state.

Our aim is to represent SQL queries as directly as possible, and allow them as first-class citizens in the language. The important step to achieve this goal is to identify all the syntactic elements that are implicit in a `SELECT` command and introducing them as expressions in SML#. Introduction of a server definition and other SQL commands are relatively straightforward.

To identify those implicit syntactic shorthands, let us examine the following simple `SELECT` command.

```
SELECT name FROM people WHERE age >= 25
```

This implicitly assumes the following.

1. Field names `name` and `age` in `SELECT` and `WHERE` clauses denote the results of the corresponding field selections from a tuple in the table given in `FROM` clause.
2. `SELECT` phrase creates a tuple whose labels are inherited from the field names.
3. This query is executed against a given database connection. This implies that the table name `people` in `FROM` clause represents the selection of the `people` table from a database consisting of a named collection of tables.

The first two points are made explicit by the following more verbose version of the same SQL command

```
SELECT person.name AS name, person.age AS age
FROM people AS person
WHERE (person.age >= 25)
```

where a variable `person` is bound to a representative tuple, from which the mentioned fields are extracted. To represent the third point, we represent a query expression as a function that explicitly takes a database connection `db` as a parameter and selects the referenced table `people` from `db`. This analysis yield the example expression we gave in Section 2.

```
e   ::=  ...                                  (SML# expressions)
     |   _sqlserver [string] : τ              (server definition)
     |   _sql x => sql                        (SQL expressions)
     |   _sqleval e                           (query evaluation)
     |   _sqlexec e                           (command execution)
     |   #x.l                                 (selection)

sql    ::=  select | insert | update | begin | commit | rollback

select ::=  select e [ as l ] ... e [ as l ]
            [ from e as x ... e as x ]
            [ where e ]
            ··· (other clauses)

insert ::=  insert into #x.l (l, ..., l)
            values ( {e|default}, ..., {e|default})

update ::=  update #x.l [ as x ]
            set (l, ..., l) = (e, ..., e)
            [ from e as x ... e as x ]
            [ where e ]

delete ::=  delete from #x.l [ as x ]
            [ where e ]
```

Figure 2. The syntax of SQL extension in SML#

We also need to introduce syntax for executing an SQL query command against a database connection. There should be two forms, one for `SELECT`, which returns a relation, and the other for all the other commands which do not return any value.

Based on these analyses, we define the syntax extension to SML# as shown in Figure 2. In this definition, note that e ranges over any expressions including those of SML#. Those who are familiar with SQL should immediately recognize that these extensions are direct representation of (a subset of) SQL. SQL contains more features, which can be added without any problem, as we discuss later.

3.3 Typing extension for SQL

We should emphasize that these SQL expressions are not macro but they are all legal expressions that can be freely combined with other SML# expressions. Moreover, each sub-expression e appearing in those SQL syntax can be any expression definable in SML#. Of course not all syntactically well-formed expressions are legal; the set of expressions that have well-defined meaning are those that have a *type* inferred by the type system.

Our goal is to extend the type system of SML# so that it always infers a most general type for any legal SQL expression. Since the type system guarantees that any typable expression can be freely combined as far as its type is consistent, this achieves our goal of seamless integration of SQL into Standard ML.

The type system is defined through a set of typing rules to derive typing judgments in the style of [24]. We have worked out the details of the typing rules for the above SQL extension. In this section, rather than giving a formal typing derivation system, we present the type system by explaining the meanings of new type constructors and the behavior of the associated typing rules.

3.3.1 The set of new type constructors

The set of types of SML# is extended with the following new type constructors for the SQL extension.

```
τ  :=  ··· | τ server | τ conn | τ query | command | τ rel
    |  (τ, τ') db | (τ, τ') table | (τ, τ') row
    |  (τ, τ') value
```

Intuitive meanings of these types are the following; their precise meaning will become clear when we explain typing properties of associated operations. τ `server` denotes a server of a database of type τ by which SQL expressions are evaluated. In what follows, we call such a server as an SQL server. τ `conn` represents an established connection to an SQL server of type τ `server`. τ `query` represents an SQL `SELECT` command that return a relation of type τ. `command` is the type of the other SQL commands such as `insert` and `rollback`, which cause some side effects to the SQL server state. τ `rel` represents a relation containing tuples of type τ returned from an SQL server as a result of query evaluation. The other four types (τ_1, τ_2) `db`, (τ_1, τ_2) `table`, (τ_1, τ_2) `row`, and (τ_1, τ_2) `value` are used to type expressions inside of SQL queries and commands. They respectively denote an active database connection of type τ_1, a table of type τ_1 in an active database connection, a tuple of type τ_1 in an active database connection, and an atomic value of type τ_1 in an active database connection. τ_1 of type `value` is either an atomic base type b for values without null or b `option` for values possibly containing null. The extra second type parameter τ_2 in these types is introduced to ensure the consistency of database connections in SQL queries we shall explain below.

Using these types, we represent legal SQL expressions through SQL constructs. In the following, we introduce SQL constructs and explain their typing relations.

3.3.2 SQL servers

A server expression `_sqlserver [`*string*`]` : τ declares that there is an SQL server of type τ at the location specified by *string*. This *string* may also contain some extra information to establish a connection to the server. In our current version, the location specification *string* is designed so that it is expressive enough for PostgreSQL or MySQL servers. τ specifies the set of tables in the database as a form of record type. This expression is always well typed with type τ `server`. Note that giving a type to servers makes them first-class values. This feature gives flexibility to database programming. For example, a function can take list of servers with additional information such as their capacities and can dynamically choose appropriate one at runtime.

A runtime value of a server expression is a server object consisting of the server information including its location and a runtime representation of its type τ. The only operation on a server object is the primitive function

```
SQL.connect : ['a.'a server -> 'a conn].
```

This primitive attempts to connect the SML# runtime system to the SQL server using the location information stored in the server object of type τ `server`. If the attempt succeeds, then it dynamically checks that the connected server has indeed the type τ stored in the server object. This typechecking is done by retrieving database schema information from the remote SQL server. If the typecheck succeeds, then the primitive succeeds and returns a connection object of type τ `conn`, which can be used to issue SQL commands. If either the connection attempt or the typecheck fails then this primitive raises a runtime exception.

3.3.3 Consistency of database connection

A distinguishing features of our language is that all the components of SQL are given types so that they are treated as first class values. This achieves powerful SQL programming using higher-order functions. As we noted above, in our system, database servers and the corresponding database connections are all first-class values. As a consequence, one can write a program that manipulates multiple database connections to different database servers. This flexibility raises one subtle issue in ensuring well-definedness of SQL query

execution. A straightforward combination of higher-order functions and first-class database connections would inadvertently admit an SQL query that involves multiple database connections. However, an SQL query needs to be evaluated by a single SQL server through a single database connection. To see the problem, consider the following example.

```
fun f db1 =
    _sql db2 => select #r1.c1, #r2.c2
                from #db1.t1 as r1, #db2.t2 as r2
```

The function `f` generates an SQL `SELECT` command for a database connection `db2`, but it uses another database connection `db1` received as a parameter. This function definition is valid only when `db1` and `db2` denote the identical connection. The following example is an invalid use of `f`.

```
val conn1 = SQL.connect (_selserver "DB1");
val conn2 = SQL.connect (_selserver "DB2");
_sqleval
  (_sql db =>
        select ··· from ···
        where ··· (_sqleval f db conn1) ···)
  conn2
```

This results in sending to an SQL server `"DB1"` a query that involves a reference to another database server `"DB2"`. A sound integration of SQL into SML# type system requires that the extended type system should statically detect this form of inconsistency due to multiple database connections in one SQL query.

We solve this problem using existential types proposed in [25]. The idea is to consider a database connection of type τ_1 `conn` as an abstract package of type

$$\tau_1 \texttt{ conn} = \exists \tau_2.(\tau_1\texttt{ , }\tau_2\texttt{) db}$$

where τ_2 is an abstract witness representing a unique connection established for a particular connection to a database. Since existential types are second-order types, general introduction of this mechanism is beyond the power of ML typing on which SML# is based. So we introduce this mechanism only for the expressions `_sqleval` and `_sqlexec` that use a database connection. These expressions temporally opens the package by generating a unique τ_2 and converting a database connection type τ_1 `conn` into a database type $(\tau_1\texttt{ , }\tau_2\texttt{)}$ `db`. All the database components (tables and values) that forms an argument to the same `_sqleval` or `_sqlexec` are typed with the same unique witness type τ_2. This ensures the consistent use of connection without introducing ad-hoc restriction on the usage of first-class SQL constructs. The second type parameter τ_2 in $(\tau_1\texttt{ , }\tau_2\texttt{)}$ `db`, $(\tau_1\texttt{ , }\tau_2\texttt{)}$ `table`, $(\tau_1\texttt{ , }\tau_2\texttt{)}$ `row`, and $(\tau_1\texttt{ , }\tau_2\texttt{)}$ `value` are introduced for this purpose. The precise typing rule is rather involved, but it is a standard application of existential types. As we shall describe below, our implementation achieves an efficient and simple type inference method specialized for `_sqleval` and `_sqlexec`.

3.3.4 Component selection expression

The `select` expressions and other SQL commands contains component selection expressions of the form $\#x.l$, which denotes the l component of the value denoted by x. The value may be a database or a tuple, both of which have a labeled record structure. Correctly inferring the most general type of this construct is a key to achieving seamless integration of SQL into SML#.

For this expression, the type inference system performs the following.

- Infer a type of x. The result should be either a database type $(\tau_1\texttt{ , }\tau_2\texttt{)}$ `db` or a tuple type $(\tau_1\texttt{ , }\tau_2\texttt{)}$ `row`. In these types, τ_1 is a record type representing their structure, and τ_2 represents a unique database connection.
- It unifies τ_1 with a fresh type variable with a record kind $\texttt{'a\#}\{l\texttt{:'b}\}$.
- If unification succeed, then the type inference system computes an instance of the component type `'b`. Let the result be τ_3. The type inference system returns $\texttt{('b, }\tau_3\texttt{) table}$ if $\#x.l$ appears in a context that requires a table, otherwise it returns $\texttt{('b, }\tau_3\texttt{) value}$.

3.3.5 SQL `SELECT` command

With these extensions, SQL `SELECT` command can be typed. Its syntax in our extension has the following form.

```
_sql db => select e1 as l1 ... en as ln
           from e^1 as x^1 ... e^k as x^k
           where ew
```

For this expression, the typechecker performs the following.

- Infer a type τ for e_w and unify τ with $\texttt{(bool, }\tau_0\texttt{) value}$.
- For each e^i, infer types τ^i and unify τ^i with $(\tau^{i'}\texttt{, }\tau_0\texttt{) table}$, and add the bindings $\{x^i \texttt{ : } (\tau^{i'}\texttt{, }\tau_0\texttt{) row}\}$ to the typing environment.
- Under the extended typing environment, infer types $\tau_1, \ldots, \tau_n$ for $e_1, \ldots, e_n$.
- If all the inference steps succeed, then it returns the type $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ `query` as a result.

3.3.6 Sub queries

SQL allows arbitrary nesting of `SELECT` commands by treating the result of a `SELECT` command as a table. This is simply done by the introduction of the following typed primitive.

```
SQL.subquery : ['a, 'b.
                 (('a,'b) db -> 'a query)
                  -> ('a,'b) db -> ('a,'b) table
```

The first parameter is a `select` expression, and the second parameter is the current database connection. This converts the `select` expression to a virtual table on that connection.

3.3.7 Overloaded operators

In addition to the mechanism above, one more extension is required to infer a desired polymorphic type for SQL expressions. To explain this, we note that an SQL expression is not only polymorphic in the database structures it manipulates but it also implicitly uses overloaded operators such as comparison `>`. Since conventional SQL commands are executed under a fixed monomorphic database, all the overloading are resolved by the server without any problems. However, if we treat a query as a first-class value, then we need to represent the overloaded operators as first-class values as well. To see the need, consider the following query.

```
fun f x =
    _sql db => select #person.name as name
               from #db.people as person
               where SQL.> (#person.age, x)
```

There are multiple possible types for `age` and accordingly there are possible multiple instances of `SQL.>`. Since SML# is a typed language, we must infer a polymorphic type of this function. Instead of introducing a general mechanism for overloading such as type classes [17], which would significantly complicate the underlying type system of Standard ML, our solution is to introduce a new kinded type variable of the form $\texttt{'a::}\{\tau_1\texttt{,}\ldots\texttt{,}\tau_n\}$ representing

the fact that type variable `'a` is an overloaded type whose instance is restricted to one of its list $\tau_1,\ldots,\tau_n$. The type of the parameter `x` in the above function is type variable `'a` with the following kinding:

```
'a::{int, word, char, string, real, 'b option},
'b::{int, word, char, bool, string, real}
```

The listed types are atomic types shared among ML and SQL. Since the `age` column in the actual database may contain null, the set of instances must contain `int option` as well as `int` etc. Thus, the set of all possible instances of `'a` is {`int`, `word`, `char`, `string`, `real`, `int option`, `word option`, `char option`, `bool option`, `string option`, `real option`} which is the standard set of types allowed for the `age` field in an SQL database. The same restriction applies to the result value of a column selection expression such as `#person.name`.

Combining all the typing mechanisms, our type system always infers a principal type for any type-correct SQL expression. For the above `f`, for example, the following type is inferred.

```
val it = fn
 : ['a::{int, word, char, string, real, 'b option},
    'b::{int, word, char, bool, string, real},
    'c.
    ('a, 'c) SQL.value
    -> ['d#{people: 'e},
        'e#{age: 'a, name: 'f},
        'f::{int, word, char, string, real,
             'g option},
        'g::{int, word, char, bool, string, real}.
        ('d, 'c) SQL.db -> {name:'f} SQL.query]]
```

Although the inferred type is notationally involved, it is not hard to see that this is exactly the constraint imposed by the above SQL query. It is this precise typing that achieves seamless integration of SQL into a typed polymorphic high-order functional language.

3.4 The operational semantics

Once we have worked out the typing structures of all the components of SQL, the next step is to develop an evaluation model for the SQL extension and to combine it in the compilation steps in SML#. An expression in a functional language is compiled to a code that generates a value denoted by the expression. In order to seamlessly integrate SQL in this general model, we adopt the following strategy. For various SQL components that appear inside of SQL expressions, we define their values to be the syntax tree representing the component whose leaves may contain atomic SML# values, and represent the runtime value of each primitive functions operating on SQL components as a function that combines abstract syntax trees. SQL construct `_sql` x `=>` sql is compiled to a code that takes a database connection and returns an abstract syntax tree representing the corresponding SQL command to be sent to the server through the connection. `_sqleval` and `_sqlexec` are implemented as functions that take an abstract syntax tree, generate an SQL string from the tree, and sends it to the server. In actual implementation, we do not explicitly generate an abstract syntax tree but generate its string representation on the fly.

In the case of `_sqleval`, it must perform additional task of converting the returned result to the SML# runtime values. We solve this problem by associating `_sqleval` with a function that generates SML# values from the server's result. The runtime representation of τ `rel` is a value containing such a function. `SQL.fetchAll` : τ `rel -> `τ `list` converts a query result to an SML# list of records by invoking the conversion function.

3.5 Other SQL features and examples

The syntax of SQL contains additional components other than those defined in the syntax extension in section 3.2. For example, `SELECT` command has many clauses other than `FROM` and `WHERE` such as `ORDER BY`. Some of those additional components require extension to the syntax, and others are realized by adding primitive functions. In this section, we take `ORDER BY` clause of `SELECT` command as an example for the former, and logical operators including `EXISTS` predicate for the latter. Other components can be introduced similarly to one of these two.

The `ORDER BY` clause of SQL sorts tuples in the query result according to columns in the select list as well as those of the tables appearing in `FROM` clause. Original `ORDER BY` clause refers the select list in anonymous way. To realize this behavior in our syntax, we need to explicitly introduce a variable which is bound to a result tuple. We use `into` keyword for this binding. The syntax of `select` command is refined as follows.

$$\begin{array}{lll} \mathit{select} & ::= & \texttt{select}\ e\ [\ \texttt{as}\ l\ ]\ \ldots\ e\ [\ \texttt{as}\ l\ ]\ \texttt{into}\ x \\ & & \ldots \\ & & \texttt{order by}\ e\ [\{\texttt{asc}|\texttt{desc}\}]\ \cdots\ e\ [\{\texttt{asc}|\texttt{desc}\}] \end{array}$$

The type of x following `into` is (τ_1, τ_2) `row` where τ_1 is a record type representing the type of the result tuple. This x can be seen from sub-expressions in the `order by` clause and is used to refer to the select list.

Some of primitive operations in SQL expressions can be represented as typed primitive functions. The following typed primitives represent SQL logical operators.

```
SQL.andAlso :
    ['a.(bool,'a) value * (bool,'a) value
        -> (bool, 'a) value]
SQL.orElse  :
    ['a.(bool,'a) value * (bool,'a) value
        -> (bool,'a) value]
```

`EXISTS` subquery expression can be introduced by the following typed primitive, similar to `SQL.subquery`.

```
SQL.exists :
    ['a,'b.(('a,'b) db -> 'a query)
            -> ('a,'b) db -> (bool,'b) value]
```

Using those features, the programmer can enjoy database programming with full spectrum of SQL directly in Standard ML. Figure 3 shows an example that retrieves employee information from an employment database and constructs a nested list of employees for each department. `employees` is a polymorphic query that receives a condition for `where` clause and returns a list of pairs of an employee's name and his/her salary in descending order of salary. `employees'` is another polymorphic query obtained from `employees` by applying it to a partial condition. `depts` holds the list of pairs of a department name and its ID. The `map` function in the definition of `forEachDept` performs iteration over the department list and constructs and emits a query for each department. `l1` and `l2` hold resulting employee lists under different conditions.

4. Implementation

We have implemented the presented extension in SML# compiler. Through our effort of modular and systematic extension of a complex system of SML# compiler, we have developed an implementation technique for connecting static semantics of SQL in the host language, SML#, and its dynamic semantics realized by an external SQL server. This technique should be useful for extending a compiler with a domain specific language. In the next subsection, we discuss the problem in extending an existing compiler and describe

```
val db = _sqlserver "dbname=employment"
         : {employee: {name:string, age:int,
                       salary:int, deptId:int},
            department: {id:int, name:string}}

val conn = SQL.connect db

fun employees condFn =
    _sql db => select #r.name, #r.salary
               from #db.employee as r
               where condFn r
               order by #r.salary desc

fun employees' condFn deptId =
    employees
      (fn r => SQL.andAlso
                 (SQL.== (#r.deptId, deptId),
                  condFn r))

val depts =
    SQL.fetchAll
      (_sqleval
         (_sql db =>
            select #r.name as deptName,
                   #r.id as deptId
            from #db.department as r
            order by #r.name)
         conn)

fun forEachDept queryFn =
    map (fn {deptId, deptName} =>
            {deptName = deptName,
             employees =
               SQL.fetchAll (queryFn deptId) conn})
        depts;

val q1 = employees' (fn r => SQL.> (#r.salary, 5000))
val l1 = forEachDept (fn i => _sqleval (q1 i));

val q2 = employees' (fn r => SQL.> (#r.age, 50))
val l2 = forEachDept (fn i => _sqleval (q2 i));
```

Figure 3. example of combination of ML and SQL

the implementation technique we have developed. We then reports the details of the implementation in the subsequent subsections.

4.1 Implementation strategy

A natural way of extending a compiler with SQL would be to add its syntax and the corresponding intermediate representations, and then to add to each compilation phase the cases that process the new components in such a way that the added SQL components achieve the intended typing and runtime behavior. This approach requires detailed analysis on interaction between the existing system and new components, and is difficult for a large and complex language such as a compiler of Standard ML including SML# compiler, which contains more than 300K line of code organized in more than 20 compilation phases. This natural strategy would also be problematic in extensibility and maintenance, since the added extensions will become tightly built-in the core of the compiler. In order to extend the SML# compiler efficiently and reliably, we should localize the extensions and minimize the modifications of the core of the compiler. Our basic strategy for achieving this goal is to use the compiler's functions as far as possible.

Extending a typed language with some new constructs requires to implement their *static semantics*, which infers types and properly propagates them to the rest of the language, and their *dynamic semantics*, which realizes the desired runtime effect. In our implementation, we have successfully organized the SQL extension in such a way that both static and dynamic evaluation are realized systematically using the existing functionality of the SML# compiler. This is based on our following observations.

Components of SQL expressions including "databases", "tables", and "rows" are all labeled record structures combined with a type constructor for collections (relations). These components can be represented in the type system of SML#, which supports record polymorphism. We can then directly implement any SQL expression as a source program in SML#. This would yield a *toy implementation* of SQL in SML#. Of course, such a toy implementation is not what we really want. Our goal is to seamlessly extend SML# with SQL in such a way that the SQL part is evaluated by an efficient practical database server. However, as far as typing is concerned, such a toy implementation is good enough; for each SQL expression, we can generate a toy program whose type is the same as that of the SQL expression. This means that we can obtain the desired static semantics of an SQL expression by constructing an appropriate toy source program that corresponds to the SQL expression. Moreover, types are only used for static enforcement of legal combination of SQL expressions with other language constructs, and are not needed at runtime. A dynamic semantics of an SQL expression is realized by generating an SQL command string and sending them to an SQL server without using type information. This process is also easily represented by an SML# program. The desired SQL extension can then be realized by connecting the static semantics realized by a toy implementation and the dynamic semantics realized by a database server.

The above observation leads us to the following implementation strategy. The compiler first translates an SQL expression to a source program that computes a pair of a toy implementation and a target SQL command string. Each primitive operation of SQL is translated to a source program that simultaneously composes both the toy implementation and the target SQL command string. It is not hard to define a data type for the source program so that the type of the toy implementation becomes the type of the entire result. The value of the translated program is a pair of the toy implementation and the target SQL command string. So the remaining thing for the compiler to achieve the desired dynamic semantics is to compile `_sqleval` and `_sqlexec` to a source program that takes the source program generated from an SQL expression, runs the program to obtain a pair, throws away the first component, and sends SQL command string stored in the second component to the database server.

The first major phase of the SML# compiler is *elaboration*, which translates a given abstract syntax to a basic source program, called a *core ML program*, by expanding derived forms (sugared syntax). Since most of the above process are definable in the core ML language, the major part of SQL compilation can be implemented by extending the elaboration phase of the SML# compiler so that it translates SQL expressions to appropriate core ML programs described above.

In order to carry out the implementation based on this strategy, we have to develop the following components and integrate them in the SML# compiler.

1. *Extension to the SML# type system.* Although most of SQL type structures can be representable in the core ML language of SML#, a complete representation of actual SQL expressions requires some extensions to its type system. The major one is the introduction of existential types we explained in Section 3 to ensure that all the components in one SQL expression refer

to the same database connection. Introduction of this feature requires to extend the typed intermediate language of the SML# compiler and its type inference system.

2. *SQL type definitions.* Under our strategy, the compiler generates a core ML program that achieves the static semantics of the original SQL program. Those generated programs need to reference types of SQL component such as "tables" and "rows". This means that these types should be predefined and loaded before the compilation.
3. *Core ML program generation.* After the preparation of the above two steps, the compiler generates a core ML program from a given SQL expression.
4. *Connection management.* In addition to compile SQL expressions, the compiler and runtime system must also manage server connection. An important additional role beside making server connection is to dynamically check type conformance of the sever database against the declared server type. Since types are static entities that only exists during compilation, a special compiler support is necessary for this purpose.
5. *SQL server binding.* To communicate with an SQL server process, we need primitive functions to access an SQL server. Modern SQL servers usually provide such features as their own low-level APIs. To make our implementation independent of particular SQL server as much as possible, we introduce a *server binding* module to abstract the low-level communication.

The following five subsections explain these major components.

4.2 Extension to the SML# type system

New typing mechanisms necessary for SQL expressions include overloading and existential types.

4.2.1 Overloading compilation for SQL primitives

As we have explained in 3.3.7, we need to represent constant literals and primitives in SQL expressions. For this purpose, we have introduced a simple first-class overloading mechanism based on polymorphic record compilation. Compared with type classes in Haskell [17], the modification to SML# type system is simple, and it can be implemented by small modification to the existing mechanism for record compilation. Here, we outline the necessary extension to SML# type system and the implementation. For the sake of explanation, we consider an overloaded primitive O whose instances is represented by a polymorphic type of the form

$$O : \forall t :: \{b_1, \ldots, b_n\}.\tau$$

where $\{b_1, \ldots, b_n\}$ is a set of base types. For this simplified setting, each of instance functions of O is identified by a base type b listed in the kind, which we write O_b.

To deal with overloading specified in the definition of Standard ML, the original SML# type system already contains overloaded kind of free type variables of the form $t :: \{b_1, \ldots, b_n\}$. where $b_1, \ldots, b_n$ are distinct base types. If these type variables remain free at the top-level, they are not generalized but are instantiated with the first component b_1 of the list of types $\{b_1, \ldots, b_n\}$. To extend the type inference system with first-class overloading, we have only to allow these type variables with overloaded kind to be generalized. The kinded type system of SML# correctly propagates and resolves overloaded instance types.

The above modified type inference system of course does not reflect the dynamic semantics of overloaded primitive O. The compiler needs to resolve overloading by selecting appropriate instance O_b according to an instance b of t. For this purpose, we apply the idea of polymorphic record compilation and compile a term with of overloaded primitive O to a higher-order function that takes an appropriate overloaded instance as an extra parameter. For this purpose, in the target language, an overload kind is extended to be a pair $(\{O_1, \ldots, O_m\}, \{b_1, \ldots, b_n\})$ by including the set $\{O_1, \ldots, O_m\}$ of overloaded primitives involved. When type variables with overloaded kinds are unified, the type system takes the intersection of instance sets and the union of the primitive sets. When polymorphic generalization is performed on those overloaded variables, extra bound variable for each O_i is introduced, and when polymorphic instantiation is performed, the actual instance primitive is passed as an additional parameter. With this preparation, the compilation process is essentially the same as that of polymorphic record compilation. For example, a function

```
val f = fn x => +(x,*(x,x))
        : ['a::{int, real}. 'a -> 'a]
```

is compiled to the following term:

```
val f = fn I+ => fn I* => fn x => I+(x,I*(x,x))
        : ['a::({+,*},{int, real}).
             Inst(+, 'a) -> Inst(*, 'a) -> 'a -> 'a]
```

where $Inst$(+, 'a) is the singleton type that denotes the primitive + for the type 'a.

In our actual extension, we allow each overloaded instance type b_i to take another overloaded type variable as its type parameter to represent *nested* overloaded instances. By allowing the nesting, we can define overloaded primitives not only over base types but also over constructed types. This is needed to represent the set of types shared among ML and SQL as described in section 3.3.7. For example, in our SQL extension, the type of SQL.>, which is a comparison primitive for SQL queries, is declared as follows:

```
['a,
 'b::{int, word, char, bool, string, real},
 'c::{int, word, char, string, real, 'b option}.
 ('c, 'a) SQL.value * ('c, 'a) SQL.value
   -> (bool option, 'a) SQL.value ]
```

Overloaded type variable 'c in this example refers to another overloaded type variable 'b to form a nested instance set, which represents the set of types for which comparison operation is defined in SQL. The necessary extension to the SML# compiler is small and modular.

4.2.2 Existential types for database connections

In order for an SQL expression _sql db => sql to have well-defined meaning, any component in sql must reference to the same server connection. As we have pointed out, one way to ensure this restriction is to consider the database connection db as an abstract package having an existential type $\exists t.\sigma$ providing the capability of accessing its component tables and columns therein. The resulting type system yields sound typing for SQL expressions, successfully excluding anomalous terms such as the example shown in Section 3.3.3. However, introduction of existential types in their general form makes the type inference system of ML incomplete. Our solution is to restrict existential types to the combination of _sql db => sql and _sqleval (or _sqlexec.) Here we only explain the case for _sqleval. The case for _sqlexec is the same. The basic role of the construct _sqleval (_sql db => sql) conn is to bind db to a connection denoted by conn. In addition, the type system generates a unique witness type and propagated through all the db component in sql. To represent this effect, the type system treats this application specially. When typechecking the above application, the type system generates a new type 'a dbi, unifies the type of db, infers the application, and finally checks that the newly generated type variable 'a does not occur in both the type environment and the type judgment. This final check ensures

that all the elements in *sql* reference to the same server connection denoted by `conn`.

4.3 SQL type definitions

Our strategy is to translate SQL expressions to core ML programs of SML#. This requires that all the types of SQL component such as "database" and "tables" used in the generated programs should have been defined. For this purpose, we extend the set of built-in types of SML# with the following types.

```
datatype ('a, 'b) db =
         DB of 'a * 'b dbi
datatype ('a, 'b) table =
         TABLE of 'a * string * 'b dbi
datatype ('a, 'b) row =
         ROW of 'a * string * 'b dbi
datatype ('a, 'b) value =
         VALUE of 'a * string * 'b dbi
```

The first type argument `'a` of `db`, `table`, `row`, and `value` is the type of the toy program. The second type parameter `'b` is used to propagate a unique connection type explained above. The string component in the implementations of `table`, `row`, and `value` holds the (partially constructed) target SQL command string.

4.4 Core ML program generation

Using these types and their data constructors, the compiler generates core ML programs of SML# from SQL expressions. This is done by adding a case for each of SQL expression syntax to the SML# elaboration phase, which recursively traverse a given source program. Each case is largely mechanical with a special care of ensuring that a unique connection type should be properly propagated. As a simple example, component selection, #x`.`l, in a source program is translated to the following core ML program.

`case` x `of DB (`y`, {`l`=`z`,...}) => TABLE ("`l`", `y`, `z`)`

In this program, "l" is a string literal to appear in the SQL command string being generated, the part {l=z,...} => z is the toy implementation that generates polymorphic typing of this selection, and y => y is for propagating a unique connection type. Each SQL operation is translated to a program that composes the toy parts, concatenates the SQL command strings, and propagates the unique connection type.

Since some of SQL components only concatenate the SQL command strings, connection types need to be propagated through SQL command string concatenation. For this purpose, we introduce a special string concatenation function `concatQuery` of type `(string * 'a dbi) list -> string`, and passes component strings together with connection witnesses to this function. This function concatenates a list of string with the typing effect of unifying all their connection types to the same type. This achieves the desired typing effect.

There is one more subtlety that should be take care of in achieving seamless integration of SQL into SML#. We have so far identified SQL rows (tuples) with Standard ML labeled records. This identification works fine for component selection and its typing. However, internal representations of SQL rows returned from an SQL server are different from those of ML record representation. This implies that a row fetched from the result of an SQL expression returned from a server should be converted to the internal representation of Standard ML. Under our strategy, this requires to generate a core ML program that performs such conversion. Moreover, we need to generate such a conversion function for each SQL query using its type information. This is best done at the time of translating a query expression, when all the necessary type information is available. Our source level translator generates a core ML program that performs this conversion just after an entire query is constructed. The result of our translation of an SQL query is then a core ML program having the following type.

Source program:

```
_sql db =>
  select #person.name as name
  from #db.people as person
```

Translated core ML terms:

```
fn db as DB (dbi, _) =>
   let
     val (tabname, person) =
         case (case db of
                 DB (i, {people = w, ...}) =>
                 TABLE (("people", i), w)) of
           TABLE (t as (_, i), w) =>
           (t, ROW (("person", i), w))
     val VALUE (nameExp, nameW) =
         case person of
           ROW (n, {name = w, ...}) =>
           VALUE (concatDot (n, "name"), w)
     val witness = {name = nameW}
   in
     QUERY
       (concatQuery [("SELECT ", DBI),
                     nameExp,
                     (" AS name FROM ", DBI),
                     tabname,
                     (" AS person", DBI)],
        witness,
        fn RESULT result =>
           {name = fromSQL
                     (0, result, #name witness)})
   end
```

Figure 4. example of query translation

```
datatype 'a query =
         QUERY of string * 'a * (result -> 'a)
```

The first component is an SQL command string to be sent to a server, the second component is a toy implementation, and the third component is the conversion function.

Figure 4 shows an example of translating a query expression into a core ML program.

4.5 Connection management

The remaining component of our extension is connection management. This is responsible for defining a server, establishing a connection, and sending a query over connection. Among them, the third one is a simple source level library function that calls one of C API provided by the database server through SML# C function interface described earlier. Server definition and connection establishment need to perform runtime typechecking to ensure the type safety of SQL expressions.

We have developed a typing machinery to infer a most general type for any legal SQL query expression. This inferred type should be checked against the type of a database stored in a database server to be connected. Since a database server is external to SML# programs, we model a database server as an object of type `dynamic` as proposed in [1]. In this formalism, an expression having a type `dynamic` is used with an explicit type annotation. Its runtime representation is a pair of its value and its runtime type information. This runtime type information is checked against the explicit type

annotation only once when this object is loaded (linked) in a program.

In our language, this model is implemented by checking that the internal type of the connected database server is the same as the static type specified in `_sqlserver` expression when a database server is connected by `SQL.connect` primitive. This is done as follows. `_sqlserver` expression is translated to a core ML code that generates a string representation of the annotated type τ together with the location information of the server. `SQL.connect` primitive is implemented as a core ML function. This function takes a server object of type τ `server` containing a server location and type information, opens a connection using the server location information, and issues a query to the database to obtain its scheme information, and then it checks that the connected database has the type τ using its string representation and the obtained scheme information. If the type check succeeds then `SQL.connect` succeeds and yields a value of type τ `conn`, otherwise it raises runtime exception. After this connection, a database is safely used with typed first-class SQL query expressions.

4.6 SQL server bindings

Since the SQL syntax of our language is (a subset of) the standardized SQL language, our language should be independent of any particular SQL server implementation. However, C APIs needed to implement the connection management are varied with SQL servers. For our system to work with any server that conform to the SQL standard, we introduce a server binding module and its interface which absorbs the difference of low-level C APIs. Any user-level primitive features of SQL extentions, such as `SQL.fetchAll` and `SQL.connect` functions, are implemented by using this binding module.

The server binding interface requires the binding module implementator to provide the following functions.

1. Low-level connection management functions through abstract connection handles, and string-based query execution functions.
2. A function to obtain schema information for runtime type-checking.
3. A mapping of type names from database types to ML types, and the corresponding set of conversion functions from query results to ML values.

With the functionality provided by any modern SQL database systems, it is routine to implement these functions by writing a small amount of code. On the requirement 1, database systems usually offer such a string-based low-level query interface, as discussed in section 1.1. The requirement 2 can be easily realized by a series of SQL queries issued through the low-level query interface since database systems usually implement system catalogs as tables which is accessible to an ordinary SELECT command. The requirement 3 can be implemented by composing low-level methods to retrieve tuples and fields from the query result and representation conversion functions. This is also streightforward except that a little care is needed to define the type name mapping.

We have implemented binding modules for PostgreSQL and MySQL. Any other database systems can be supported just by writing a binding module for the database.

5. Conclusions and Future Works

We have presented a new generation of Standard ML that seamlessly integrates SQL. In this language, a legal SQL expression is a polymorphically typed first-class citizen that are freely combined with any features of Standard ML, including high-order functions, data type definition, and its module system. The distinguishing feature of our language is that those SQL expressions are not evaluated in the ML language runtime, but they are sent to a real database server. This makes efficient practical database programming directly available in a high-level and reliable functional programming language.

We have solved a number of typing and implementation issues and have implemented the language. The implementation is done by extending the compiler of SML#, which is an extension of Standard ML with record polymorphism. Since the only crucial technical typing device we have used is record polymorphism presented in [28], we also expect that our method can be transferred to any other ML-style language as far as its type system contains a typing mechanism that is at least as powerful as record polymorphism such as OCaml objects based on Remy's record polymorphism [31] or Haskell type classes [17].

This is a step toward making practical database programming seamlessly available in a typed high-level programming language. A number of interesting future issues remain to be investigated. We briefly mention some of them below.

As mentioned in Section 1.1, there are several proposals for high-level query languages for advanced applications. One way of making these approaches practical and scalable would be to implement those advanced data models on top of SQL running on an efficient database server. The idea would be to make SQL relations as basic data structures to implement those high-level data model primitives. Since in our language various components in SQL databases are first-class values that are freely and directly manipulated by programs, our language should serve as an ideal basis for those applications.

Web programming framework is also a promising application area of our language. Interoperation between languages and databases are crucial in Web programming, and most of practical Web application framework such as Ruby on Rails provides various database access supports. Seamless integration of SQL with higher-order functional language would open up high-level Web application framework supporting various features such as "mashup" of variety of data sources and services.

Another interesting future work is to design a large scale distributed data manipulation system based on our language. Such a system would provide high-level and powerful alternative to map-reduce model [12]. As observed in [5], many future cloud applications would be benefited from database capability such as the ability to join various data sources.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):207–268, 1991.

[2] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: a programming language for object databases. *The VLDB Journal*, 4(3):403–444, 1995.

[3] J. Annevelink. Database programming languages: a functional approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 318–327, 1991.

[4] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 1987.

[5] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 975–986, 2010. ACM.

[6] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[7] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *Proceedings of the ACM SIGPLAN Conference*

on Programming Language Design and Implementation, pages 122–133, 2010.

[8] E. F. Codd. A relational model for large shared databank. *Communications of the ACM*, 13(6):377–387, 1970.

[9] E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *Proceedings of the International Symposium on Database Programming Languages*, pages 36–51, Springer-Verlag, 2009.

[10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *Proceedings of the International Conference on Formal Methods for Components and Objects*, pages 266–296, Springer-Verlag, 2007.

[11] G. Copeland and D. Maier. Making smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–325, 1984.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] A. Eisenberg and J. Melton. SQLJ part 0, now known as SQL/OLB (object-language bindings). *SIGMOD Rec.*, 27(4):94–100, 1998.

[14] J. Gil and K. Lenz. Simple and safe SQL queries with C++ templates. *Science of Computer Programming*, 75(7):573–595, 2010.

[15] G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the Ferry: Database-supported program execution for Haskell. In *Proceedings of the 22nd international symposium on Implementation and Application of Functional Languages*, to appear.

[16] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1063–1066, 2009.

[17] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[18] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196, 1996.

[19] C. Lécluse and P. Richard. The O_2 database programming language. In *Proceedings of the International Conference on Very Large Data Bases*, pages 423–432, Morgan Kaufmann Publishers Inc, 1989.

[20] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd USENIX Conference on Domain Specific Languages*, pages 109–122, 1999.

[21] Y. Leontiev, M. T. Özsu, and D. Szafron. On type systems for object-oriented database programming languages. *ACM Comput. Surv.*, 34(4):409–449, 2002.

[22] D. Maier. Why database languages are a bad idea. In F. Bancilhon and P. Buneman, editors, *Proceedings of the International Workshop on Database Programming Languages*, Addison-Wesley, 1989.

[23] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 706–706, 2006.

[24] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, revised edition, 1997.

[25] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[26] R. Morrison, F. Brown, R. Connor, Q. Cutts, A. Dearle, G. Kirby, and D. Munro. Napier88 reference manual. Technical report, University of St. Andrews, 1996.

[27] H-D. Nguyen and A. Ohori. Compiling ml polymporphism with explicit layout bitmap. In *Proceedings of the ACM Conference on Principles and Practice of Declarative Programming*, pages 237–248, 2006.

[28] A. Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995. A preliminary summary appeared at ACM POPL, 1992 under the title "A compilation method for ML-style polymorphic record calculi".

[29] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.

[30] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 46–57, 1989.

[31] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 242–249, 1989.

[32] SML#. http://www.riec.tohoku.ac.jp/smlsharp/.

[33] J. Bussche, D. Van Gucht, and S. Vansummeren. A crash course on database queries. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 143–154, 2007.

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