A Typeful Integration of SQL into Curry

Michael Hanus Julia Krone
Institut für Informatik, CAU Kiel, Germany
mh@informatik.uni-kiel.de

We present an extension of the declarative programming language Curry to support the access to data stored in relational databases via SQL. Since Curry is statically typed, our emphasis on this SQL integration is on type safety. Our extension respects the type system of Curry so that run-time errors due to ill-typed data are avoided. This is obtained by preprocessing SQL statements at compile time and translating them into type-safe database access operations. As a consequence, the type checker of the Curry system can spot type errors in SQL statements at compile time. To generate appropriately typed access operations, the preprocessor uses an entity-relationship (ER) model describing the structure of the relational data. In addition to standard SQL, SQL statements embedded in Curry can include program expressions and also relationships specified in the ER model. The latter feature is useful to avoid the error-prone use of foreign keys. As a result, our SQL integration supports a high-level and type-safe access to databases in Curry programs.

1 Introduction

Many real world applications require access to databases in order to retrieve and store the data they process. To support this requirement, programming languages must provide mechanisms to access databases, like relational databases which we consider in this paper. Relational database systems use SQL to access and manipulate data. Since SQL is typically not part of the syntax of a programming language, there are two principle approaches:

1. Pass SQL queries and statements as strings to some database connection.
2. Provide a library of database access and manipulation operations which are used inside the program and automatically translated to SQL.

Both approaches have their pros and cons. Although the first approach is quite popular, like JDBC in Java, it is well known that passing strings is a source of security leaks if their syntax is not carefully checked, in particular, in web applications [19]. Since these SQL strings are constructed at run time, SQL syntax errors can occur during execution time. Moreover, the types of the data obtained from the database depend on the database scheme. As a consequence, data must be checked at run time, e.g., via type casts, whether they fit to the expected types. Altogether, the "string-based approach" does not lead to a reliable programming style.

These drawbacks are the motivation for the second approach, i.e., to develop specific database libraries in programming languages. If a programmer uses only operations of these libraries to communicate with a database system, syntax problems due to ill-formed SQL queries cannot occur. If the host programming language is strongly typed, one could also avoid the aforementioned run-time type errors. For instance, Haskell/DB [21] provides a set of monadic combinators to construct queries as Haskell program expressions. In this way, queries become program entities that can be statically checked and combined with other program expressions. Due to the use of a fixed set of combinators, the expressiveness is limited compared to SQL. SML# [25] avoids this drawback by extending the syntax and type

S. Schwarz and J. Voigtländer (Eds.): 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming (WLP'15/'16/WFLP'16).
EPTCS 234, 2017, pp. 104–119, doi:10.4204/EPTCS.234.8

system of Standard ML to represent SQL statements as typed expressions. These are checked at compile time (by translating them into expressions that use definitions of an SQL-specific library) and translated at run time to SQL. Hence, type safety is combined with a quite flexible type system.

One disadvantage of these library-based query embeddings is the gap to the concrete and well known SQL syntax. Programmers familiar with SQL must learn to use the specific library operations or syntactic extensions to obtain the same results as with the direct use of SQL. Furthermore, the orientation towards the type of the data stored in the database instead of higher-level data models, like entity-relationship (ER) models, could be error prone if foreign keys occur in relations.

To avoid these drawbacks, we propose a new intermediate approach. We implemented this approach in the functional logic language Curry [11, 18], but it can also be transferred to other languages with higher-order programming features and a strong type system. The basic ideas of our proposal are:

1. We allow the embedding of actual SQL queries in Curry programs. Since SQL is not conform with the standard syntax of Curry, these queries are embedded as "integrated code," a concept which is recently supported by some Curry systems. For instance, the following piece of code is a valid program:

   ```
   studAgeBetween :: Int → Int → IO (SQLResult [(String,Int)])
   studAgeBetween min max =
     ``sql Select Name, Age
           From Student Where Age between {min} and {max}
           Order By Name Desc;''
   ```

   This operation returns the name and age of all students stored in the database with an age between the bounds `min` and `max` provided as parameters, where the result list is ordered descending by the names.

2. To check the correctness of SQL queries at compile time, they are analyzed by a preprocessor. This preprocessor replaces SQL queries by calls to operations of a library to support a typed access to databases. The preprocessor takes an ER model of the database into account in order to generate calls to appropriately typed operations.

3. The ER model is also used to avoid the explicit use of foreign keys in SQL queries. For this purpose, we slightly extend the syntax of SQL and allow conditions on relations in SQL queries.

4. The actual access to the database is done during run time by translating the library operations and combinators into SQL.

As a result, we obtain a framework which allows to embed SQL query strings into the program code, but these queries are checked at compile time to avoid possible run-time errors.

In the next section, we sketch some basic features of Curry and the already existent framework to integrate foreign code in Curry programs. Section 3 surveys the Curry libraries implementing a type-safe database interface which are used in our transformation process. Since we use the entity-relationship model to specify the logical structure of the database, we describe in Sect. 4 this model and how it is used in our framework. The design and implementation of our SQL parser and transformation is described in Sect. 5 before we discuss related approaches in Sect. 6 and conclude.

2 Curry and Integrated Code

We assume familiarity with functional logic programming [2, 15] and Curry [11, 18] so that we sketch only some basic concepts. Note that we only use the functional kernel for the database integration (in

contrast to [13]) so that a detailed understanding of the logic and concurrent features of Curry is not required.

Curry is a modern declarative multi-paradigm language which amalgamates the most important features of functional and logic languages in order to provide a variety of programming concepts to the programmer. For instance, demand-driven evaluation, higher-order functions, and strong polymorphic typing from functional programming is combined with logic programming features, like computing with partial information (logic variables), constraint solving, and non-deterministic search. It has been shown that this combination leads to better abstractions in application programs such as implementing graphical user interfaces [12], web frameworks [17], or access and manipulation of persistent data [13].

The syntax of Curry is close to Haskell [27] but allows also *free* (*logic*) *variables* in program rules and a non-deterministic choice of expressions. For instance, the following program defines the well-known list concatenation and an operation `perm` which non-deterministically returns a permutation of the input list (note that the last rule contains a functional pattern [1] to select some element of the input list):

```
(++) :: [a] → [a] → [a]          perm :: [a] → [a]
[]     ++ ys = ys                perm []            = []
(x:xs) ++ ys = x : (xs ++ ys)    perm (xs++[x]++ys) = x : perm (xs++ys)
```

Higher-order declarative programming languages support the embedding of domain specific languages (DSLs) by defining powerful combinators. For instance, this has been done in functional logic languages for parser combinators [5], business models [24], graphical user interface [12], or typed web interfaces [14]. However, there are also domains where a specific notation for DSLs exists so that the transformation into a standard functional syntax is undesirable. Examples of this kind are regular expressions, XML documents, or SQL queries. To support the direct use of such notations in Curry programs, we implemented a "code integrator" for Curry (which has some similarities to quasiquoting in Haskell [22]).

Integrated code is a string in a source program which follows its own syntax rules and is translated into a standard Curry expression by a preprocessor, also called *code integrator*. To mark a piece of integrated code in a Curry program, it must be enclosed in at least two back ticks and ticks. One can also use any larger number of back ticks and ticks (for instance, if a smaller number of them might occur in the integrated code), but the number of starting back ticks and ending ticks must always be identical. After the starting back ticks, an identifier specifies the kind of integrated code so that, in principle, any number of different kinds of codes can be integrated into a Curry program. For instance, the current code integrator supports with the code identifier `regex` the POSIX syntax for regular expressions. Thus, the following expressions are valid in source programs:

```
if s ``regex (a|(bc*))+'' || s ````regex (ab*)+'''' then ...
```

The code integrator replaces this code by a standard Curry expression which matches a string against the given regular expression. This is obtained by exploiting a match operation and data structures to specify regular expressions defined in the standard library `RegExp`.[1] For instance, the second regular expression above is replaced by the Curry expression

```
`match` [Plus [Literal 'a', Star [Literal 'b']]]
```

Note that the Curry expression replaced for the integrated code is put into a single line in order to avoid potential problems with Curry's layout rule and to keep the line numbering of the original program, which is important for meaningful error messages of the front end subsequently invoked.

The integrated code of regular expressions allows to apply the usual notation for regular expressions inside Curry programs without extending the syntax of the base language. As a further example, we

[1] `http://www.informatik.uni-kiel.de/~pakcs/lib/RegExp.html`

specify a predicate which tests whether a string has the structure of Curry identifiers as follows:

```
isID :: String → Bool
isID s = s ``regex [a-zA-Z][a-zA-Z0-9_']*''
```

Currently, the code integrator, distributed with the Curry systems PAKCS [16] and KiCS2 [4], supports syntactic embeddings for regular expressions, format printing (like C's `printf`), HTML and XML (with layout rules to avoid the explicit writing of closing tags), and, quite recently, an SQL dialect which is described in this paper.

In order to allow an easy integration of new languages into the code integrator, it provides a generic interface. Basically, a language-specific parser must be implemented as a function from an input string (containing the piece of integrated code) to a translated output string (actually, it is a monadic operation to provide for error messages, warnings, etc). If such a function is implemented as a parser for the specific language and a translator that yields a Curry expression, it can be connected to the actual code integrator by registering this parser. Thus, the infrastructure to embed SQL query strings in Curry programs is available. However, in order to get a type-safe translation of SQL queries into Curry expressions, some library support is necessary. This is sketched in the following section.

3 Curry Database Interface Libraries

To abstract from the concrete database and provide abstractions for type-safe access to database entities, we developed a family of libraries for this purpose. Since they are the basis of the type-safe embedding of SQL into Curry, we sketch the basic structure of these libraries in this section.

The Curry Database Interface (CDBI) libraries implement different abstraction layers for accessing relational databases via SQL. On the base level, the libraries implement the raw connection to a relational database. Currently, only connections to SQLite databases are supported, but other types of databases could easily be added. The library defines an abstract data type `Connection` and various operations on it. For instance, the operation

```
connectSQLite :: String → IO Connection
```

returns an open connection to an SQLite database with a given name. To ease the handling of individual database accesses, there are types

```
type SQLResult a = Either DBError a
type DBAction a  = Connection → IO (SQLResult a)
```

Hence, `DBAction` is the type of operations that communicate with the database, which could return an error or some data (modeled by the type `SQLResult`). Moreover, there are also monadic operations to combine database actions. Beyond basic operations to read, write, and close (`disconnect`) the connection, there are further operations to execute SQL commands on the connection (which return the answers of the database), and begin, commit, or rollback transactions. To execute a database action on an SQLite database with a given name, we define the following operation:[2]

```
runWithDB :: String → DBAction a → IO (SQLResult a)
runWithDB dbname dbaction = do conn <- connectSQLite dbname
                               result <- dbaction conn
                               disconnect conn
                               return result
```

[2]This simple implementation closes the connection after each action. In the actual implementation, the open connection is stored instead of closing it and re-used for the execution of subsequent database actions.

The raw database access works with strings, e.g., numbers stored in database entities are communicated in their string representation between the database and the Curry program. In order to set up a typeful communication, where values of particular types are transmitted to the database and values of some expected types are returned, data types for the various kinds of SQL values and types are introduced:

```
data SQLValue = SQLString String | SQLInt  Int  | SQLFloat Float
              | SQLChar   Char   | SQLBool Bool | SQLDate ClockTime | SQLNull

data SQLType = SQLTypeString | SQLTypeInt  | SQLTypeFloat
             | SQLTypeChar   | SQLTypeBool | SQLTypeDate
```

Exploiting these types, one can define an operation to execute a "typed" SQL query with the following type:

```
select :: String → [SQLValue] → [SQLType] → DBAction [[SQLValue]]
```

The first argument is the string of the SQL query which contains some "holes" for concrete values (marked by `'?'`), like

```
"select * Student where Age = '?' and Name = '?';"
```

The second argument contains SQL values for these holes and the third argument are the types expected as return values so that the entire call is a database action which returns a table, i.e., a list of rows of the expected values. The usage of holes in the SQL string instead of concrete values is useful to ensure safe encoding of all SQL values and, thus, avoid security leaks known as SQL injections. As an example, the following expression describes a database action to extract the age and email address of a student:

```
select "select Age,Email from Student where First = '?' and Name = '?';"
       [SQLString "Joe", SQLString "Fisher"]
       [SQLTypeInt, SQLTypeString]
```

Although this typed select statement is a good starting point, its use requires the correct application of the column types of the database. Since each database stores entities of a particular kind (according to its database scheme) and these entities can be related to types of the Curry program, the CDBI libraries define a type to specify this relationship:

```
data EntityDescription a = ED String [SQLType]
                             (a → [SQLValue]) ([SQLValue] → a)
```

Hence, an entity description consists of the name of the entity, a list of column types, and conversion functions from a Curry type to SQL values and back. For instance, if we define a type to represent a student (consisting of the name, first name, matriculation number, email, and age) by

```
data Student = Student String String Int String Int
```

then a corresponding entity description could be defined as follows:

```
studentDescription :: EntityDescription Student
studentDescription =
  ED "Student"
     [SQLTypeString, SQLTypeString, SQLTypeInt, SQLTypeString, SQLTypeInt]
     (\ (Student name first matNum email age)
        → [SQLString name, SQLString first, SQLInt matNum,
           SQLString email, SQLInt age])
     (\ [SQLString name, SQLString first, SQLInt matNum, SQLString email,
         SQLInt age] → Student name first matNum email age)
```

Note that this must not be done by the programmer since the data type and entity description can be automatically derived from the ER model of the database (see next section).

Beyond the typed access to database entities, it is also important to ensure type-safe queries con-

taining selection criteria (i.e., the "where" part of SQL select statements). Thus, there is a data type to describe selection criteria (with an optional group-by clause):

```
data Criteria = Criteria Constraint (Maybe GroupBy)
```

The Constraint data type allows to define a logical combination of comparisons of values from database columns or constants. The type to describe such values is:

```
data Value a = Val SQLValue | Col (Column a)
```

Hence, a value used in a criteria is either a constant or a database column of a particular type. To ensure a type-safe use of such values, the constructor Val is not directly used but there is a family of constructor functions, like

```
int :: Int → Value Int
int = Val . SQLInt
```

The columns are introduced with their entity descriptions, e.g., the age column of a student entity has type (where the concrete definition contains the column and entity names):

```
studentColumnAge :: Column Int
```

The data type Constraint contains a number of constructors to describe the possible conditions in where clauses. Moreover, there are constructor functions for various constraints, like the greater-than relation:

```
(.>.) :: Value a → Value a → Constraint
```

Hence, we can express the criteria that the age of a selected student should be greater than 21 as:

```
Col studentColumnAge .>. int 21
```

The complete list of all possibilities to construct criteria is defined in the module CDBI.Criteria.[3] Thanks to these abstractions, a constructed selection criteria is ensured to be type safe. For instance, it is compile-time type error when a database column of type Int is compared with a string.

Based on the select operation described above, there is a higher-level operation getEntries to access entities with some given criteria and ordering options. For instance, the following expression selects the first five students who are older than 21 ordered by their names in descending alphabetical order (if the first argument is Distinct instead of All, then only distinct results are returned):

```
getEntries All
           studentDescription
           (Criteria (Col studentColumnAge .>. int 21) Nothing)
           [descOrder studentColumnName]
           (Just 5)
```

In order to implement queries that return joins of several entities, the CDBI libraries allow to combine entity descriptions and provide operations to retrieve such combined entities from a database. For the selection of single columns of one or more tables, the libraries provide functions similar to getEntries. Furthermore, there are specific operations for database insertions, updates and deletion of entries.

The CDBI libraries are the base to implement type-safe SQL queries. Each SQL statement in a Curry program will be translated into appropriate calls to the CDBI operations sketched above. However, this requires a precise description of the types of the database scheme. In order to abstract from foreign keys (to be more precise, we want to support a type-safe handling of foreign keys), we use a description of the database scheme in the form of an entity-relationship model, as discussed in the following section.

[3] http://www.informatik.uni-kiel.de/~pakcs/lib/Database.CDBI.Criteria.html

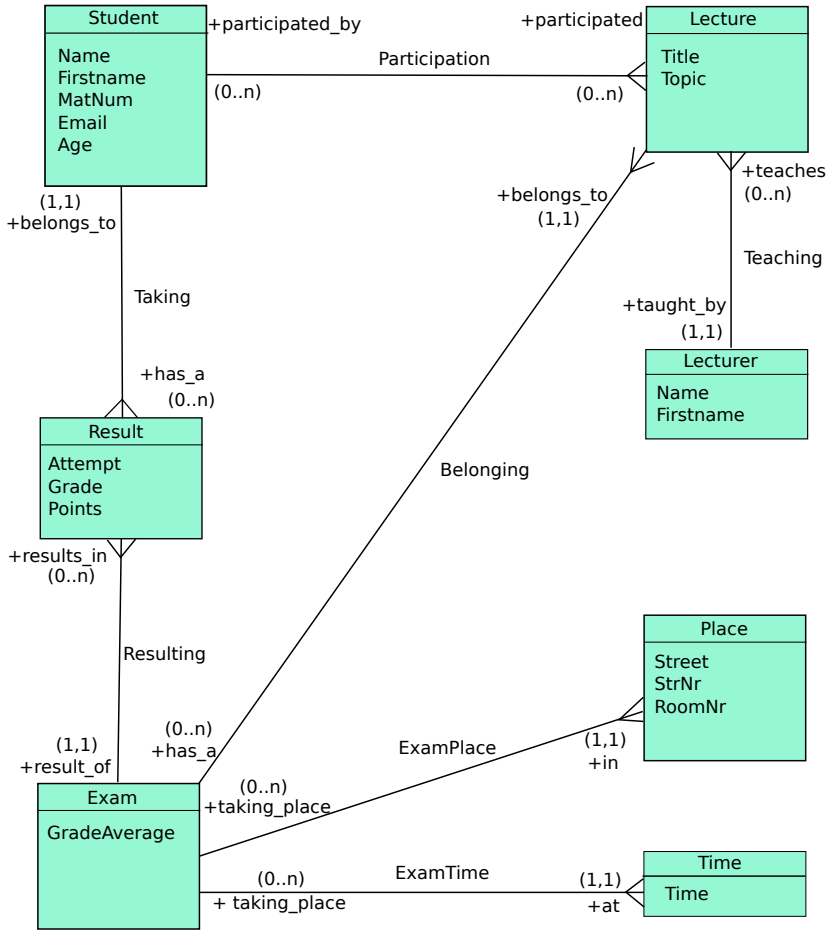

Figure 1: A simple entity-relationship diagram for university lectures

4 Entity-Relationship Models

The entity-relationship model [6] describes the structure and specific constraints of data stored in a relational database. It uses a graphical notation, called entity-relationship diagrams (ERDs), to visualize the conceptual model. The data of an application is modeled by entities that have attributes and relationships between entities. Relationships have cardinality constraints that must be satisfied in each valid state of the database, e.g., after each transaction. Figure 1 shows a concrete ERD which we use in our examples.

In order to be independent of a concrete ER modeling tool, we use a textual representation of ERDs as Curry data terms defined in [3]. In this representation, an ERD consists of a name (also used as the module name in the generated code) and lists of entities and relationships:

```
data ERD = ERD String [Entity] [Relationship]
```

Each entity consists of a name and a list of attributes, where each attribute has a name, a domain, and specifications about its key and null value property:

```
data Entity    = Entity String [Attribute]
```

```
data Attribute = Attribute String Domain Key Null
```

Similarly, each relationship consists of a name and the names and cardinalities of the related entities (more details can be found in [3]).

Since the SQL queries embedded in a Curry module are related to a particular ER model (which can be specified in a compiler directive in the head of a module), the given ER model is preprocessed to make it available to the final Curry program. This is the task of a specific translation tool (`ERD2CDBI`) which performs the following three steps:

1. The ER model is transformed into tables of a relational database, i.e., the relations of the ER model are either represented by adding foreign keys to entities (in case of (0/1:1) or (0/1:n) relations) or by new entities with the corresponding relations (in case of complex (n:m) relations). This task is performed by the existing `erd2curry` tool and described in detail in [3]. After this transformation, a relational database with the resulting structure and constraints is created.

2. A Curry module is generated containing the definitions of entities and relationships as data types. Since entities are uniquely identified via a database key, each entity definition has, in addition to its attributes, this key as the first argument. For instance, the following definitions are generated for our university ERD (among many others):

   ```
   data StudentID = StudentID Int

   data Student = Student StudentID String String Int String Int

   -- Representation of n:m relationship Participation:
   data Participation = Participation StudentID LectureID
   ```

   Note that the two typed foreign key columns (`StudentID`, `LectureID`) ensures a type-safe handling of foreign-key constraints.

   Moreover, for each entity, an entity description and definitions for their columns are generated as described in Sect. 3.

3. Finally, an *info file* containing information about all entities, attributes and their types, and relationships is created. This file provides the SQL parser and translator (see next section) with the necessary information to generate appropriate CDBI library calls. The info file contains a data term of the following type:

   ```
   data ParserInfo = PInfo String String RelationTypes NullableFlags
                           AttributeLists AttributeTypes
   ```

   The first and second components specify the absolute path to the database (later used to for the database connection) and the name of the Curry module containing the data types and operations generated from the ER model as sketched above. The latter is necessary to generate correctly qualified function calls and types in the translated SQL code.

   The third component (`RelationTypes`) contains information about all relations of the ER model, like names, related entities, and cardinality. This information is used by the SQL translator to translate constraints on relationships into database constraints relating foreign keys.

   The fourth component (`NullableFlags`) assigns to each column name a Boolean flag which is true if this column can contain null values. Note that a column of type τ with possible null values is represented in Curry by the type `Maybe` τ. In order to generate correct types for accessing column values from the Curry program, the information about null values is required by the SQL translator.

   The fifth component (`AttributeLists`) maps each table name to a list of their column names. The final component (`AttributeTypes`) maps each column name to its type, an information which is obviously required for our type-safe translation of SQL queries.

For the sake of a simple textual representation of this information, the components of this SQL parser information are stored as lists in the generated info file. When the parser reads this file, the information is converted into finite maps to support a more efficient access inside the SQL parser.

5 SQL Parsing and Translation

5.1 Design Goals

Now we have all components to implement the integration of SQL queries inside a Curry program. As mentioned in the introduction, the following piece of code should be valid in a Curry program:

```
studGoodGrades :: IO (SQLResult [(String, Float])
studGoodGrades = ‘‘sql Select Distinct s.Name, r.Grade
                       From   Student as s, Result as r
                       Where  Satisfies s has_a r And r.Grade < 2.0;’’
```

This operation connects to the database and returns, if no error occurs, a list of pairs containing the names and grades of students having a grade better than `2.0`. Note that this query is beyond pure SQL since it also includes a condition on the relation `has_a` specified in the ER model ("`Satisfies s has_a r`").

It should also be possible to include Curry expressions instead of constants in conditions of SQL queries. In order to distinguish them from SQL code, embedded Curry expressions are enclosed in curly braces. For instance, the following operation returns the email addresses of all students with a given last name (or terminates with an error):

```
studEmail :: String → IO [String]
studEmail name = do dbresult <- ‘‘sql Select s.Email From Student as s
                                      Where s.Name = {name};’’
                    return (either (error . show) id dbresult)
```

These examples demonstrate the basic objectives of our approach that must be implemented by the SQL parser and translator:

1. For easy usability, standard SQL notation is allowed (instead of specific database access operations as shown in Sect. 3).
2. SQL statements are type safe, i.e., every type inconsistency between the Curry program and the database tables is detected at compile time. For instance, if we declare the type of `studEmail` as

   ```
   studEmail :: Int → IO [String]
   ```

   a type error will be reported by the Curry compiler.
3. In addition to standard SQL conditions, we also support a condition "`Satisfies`" to check relationships between entities w.r.t. the ER model.
4. The SQL embedding is deep, i.e., we can write embedded SQL queries in any place of a Curry program (of course, only if the type is appropriate in the context of the SQL query) and we can also embed Curry expressions of appropriate types in SQL queries.
5. From the embedded SQL query, a standard SQL query is generated and transmitted at run time so that it will be processed by the database system.

5.2 Structure of the Implementation

The basic structure of the implemented SQL compiler is depicted in Fig. 2. As shown in this figure, the compiler uses the info file, described in Sect. 4, to generate appropriate Curry expressions for the

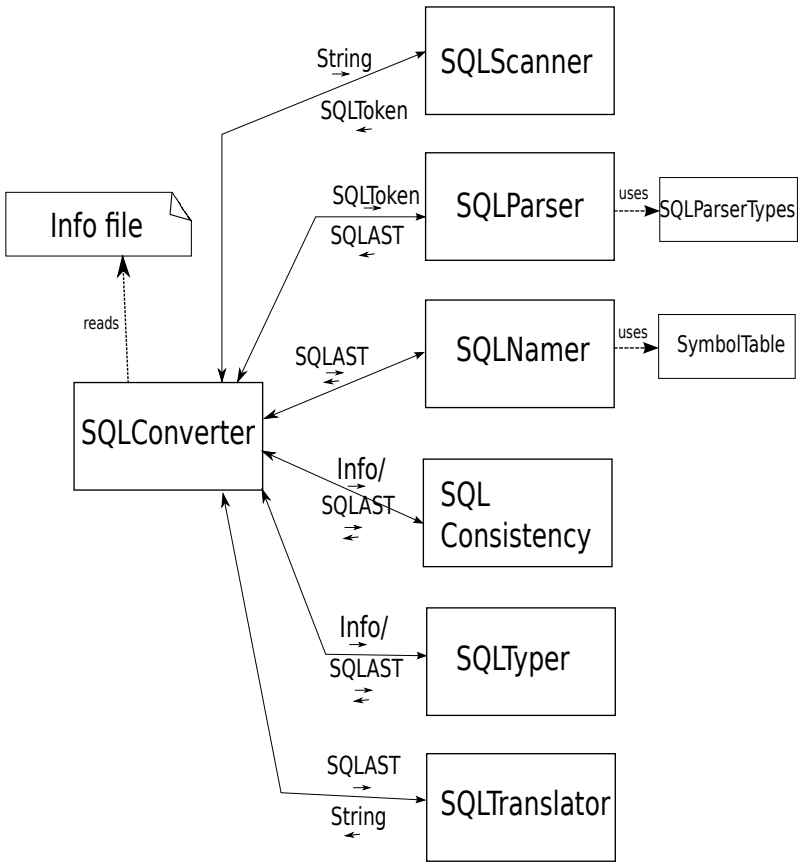

Figure 2: Structure of the SQL compiler

SQL strings. Basically, our SQL compiler follows the typical structure of compilers. For instance, in the first phase, an abstract syntax tree (AST) of the SQL statement is generated. The subsequent phases correspond to the semantic analysis and code generation and are sketched in the following.

The `SQLNamer` is the first stage of the semantic analysis. It uses a specific symbol table to resolve and replace pseudonyms for data tables (introduced by the keyword "`as`" in an SQL statement) by the full table names. Furthermore, table references are numbered in order to generate correct calls to CDBI operations (e.g., if the same table is referenced several times in an SQL statement). Errors are thrown if table pseudonyms can not be resolved. This happens in case a pseudonym was defined but not used, a pseudonym was defined for more than one table, or if the pseudonym was not defined or is not visible.

The `SQLConsistency` module implements various consistency checks. For instance, all table, column, and relationship names occurring in the SQL statement must be part of the ER model. Other consistency checks are related to the use of relations in the specific `Satisfies` condition and the correct usage of null values in insert statements. Furthermore, it is checked that null values are not used in conditions except in the predefined functions `Is Null` and `Not Null`, since this is generally forbidden by the SQL standard.

The `SQLTyper` implements an initial type checking phase. For all expressions in an SQL statement where at least one column reference is involved as well as for constant values, the types are available and, thus, can be checked. Therefore, possible type errors might be reported already by the SQL preprocessor

and not just during by the subsequent compilation of the Curry compiler. For instance, the SQL query

```
‘‘sql Select s.Name From Student as s Where s.Age = 20.5;’’
```

will not be passed to the Curry compiler, because the preprocessor reports the error message

```
... Type error: Int (Age) and Float are not compatible.
```

However, if a condition contains an embedded Curry expression instead of a constant value, the preprocessor can not ensure type safety at this stage, since the types of Curry expressions are not available during preprocessing. Instead, the `SQLTyper` deduces the type required for the Curry expression from its context in the query. This information is used in the final phase of our compiler to generate a type-correct call of the corresponding CDBI library function. Hence, the strong typing of the CDBI library, as discussed in Sect. 3, ensures the type consistency of SQL statements even in the presence of embedded Curry expressions. For instance, consider the definition

```
studNamesWithAge x = ‘‘sql Select s.Name
                           From   Student as s
                           Where  s.Age = {x};’’
```

Although the preprocessor has no information about the type of `x`, which will be eventually inferred by the subsequent invocation of the Curry compiler, it can deduce that the embedded Curry expression `{x}` must have type `Int` since it is compared with the `Int`-valued column `Age`. Thus, the preprocessor translates the `Where` condition into the following CDBI call:

```
equal (Col studentColumnAge) (int x)
```

Since the CDBI function `int` requires an argument of type `Int` (see Sect. 3), the Curry compiler would report a type error if `x` has not type `Int`. Hence, the Curry compiler infers the type

```
studNamesWithAge :: Int  →  IO (SQLResult [String])
```

and reports an error message if `studNamesWithAge` is declared with the following type:

```
studNamesWithAge :: String  →  IO (SQLResult [String])
```

Currently, it is not allowed to compare two embedded expressions, as in

```
‘‘sql Select ... Where {x*2} = {y};’’
```

Since the preprocessor is not able to infer the required types of these embedded expressions, appropriate CDBI calls cannot be generated. This is not a principle restriction of our approach but due to the current type system of Curry. If the base language supports overloading (like Haskell or an extension of Curry with type classes [23]), the preprocessor can generate a call to an overloaded CDBI operation. Another option is a complete re-implementation of the type inference of the base language in the SQL preprocessor (as done in [25]). Since we are interested in a modular SQL integration that could be used with different implementations of Curry, we do not follow the latter option. This seems a reasonable compromise since such kinds of SQL conditions are usually not necessary.

The final stage of the SQL preprocessor is the translation of the SQL AST into a valid Curry expression by the `SQLTranslator` module. Since the SQL AST is decorated with all necessary type information by the previous phases, the code generation is somehow straightforward. However, the translation of conditions on relationships (`Satisfies`) is technically involved since one has to consider the various kinds of relationships and how they are represented with foreign keys in the relational database.

To show a concrete translation example, consider the query operation `studNamesWithAge` defined above. Our SQL compiler generates the following Curry code (which is reformatted for readability):

```
studNamesWithAge x = runWithDB "/.../Uni.db"
   (getColumn []
      [SingleCS All
```

```
          (singleCol studentNameColDesc 0 none)
          (TC studentTable 0 Nothing)
          (Criteria (equal (colNum studentColumnAge 0) (int x)) Nothing)]
       [] Nothing)
```

Although there are various further parameters in order to cover the different features of SQL, the basic structure is visible. Since the CDBI libraries eventually translate the Curry calls to a single SQL query, the features of the database systems are exploited to answer the query. For instance, the call "`studNamesWithAge 30`" results in the following SQL query:

```
select ("Student"."Name") from 'Student'
                          where (("Student"."Age") == 30);
```

The complexity of the translation process becomes more visible when several relations and foreign keys are used. For instance, our initial query in this section is translated into the following Curry code:

```
studGoodGrades :: IO (SQLResult [(String, Float)])
studGoodGrades = runWithDB "Uni.db"
   (getColumnTuple []
      [TupleCS Distinct
         (tupleCol (singleCol studentNameColDesc 0 none)
                   (singleCol resultGradeColDesc 0 none))
         (TC studentTable 0 (Just (crossJoin, TC resultTable 0 Nothing)))
         (Criteria (And [equal (colNum studentColumnKey 0)
                               (colNum resultColumnStudentTakingKey 0),
                         lessThan (colNum resultColumnGrade 0) (float 2.0)])
                   Nothing)]
      [] Nothing)
```

This complex expression is translated into the compact SQL query:

```
select Distinct ("Student"."Name"), ("Result"."Grade")
from  'Student' cross join 'Result'
where (("Student"."Key") == "Result"."StudentTakingKey")
      and (("Result"."Grade") < 2.0)  ;
```

It should be obvious that a distinct advantage of our SQL embedding is the handling of foreign keys, which is type-safe (actually, hidden in the SQL query) in our framework but error prone with raw SQL.

Although we discussed only the translation of `select` queries, our SQL embedding also supports other SQL statements, like `insert`, `delete`, or `update` statements. In contrast to `select` queries, which have the result type `IO (SQLResult [`τ`])` (where τ depends on the kind of selected data), the latter SQL statements are of type `IO (SQLResult ())`. Due to the complexity of SQL, our compiler does not support the full SQL syntax. In particular, the use of join operators (e.g., natural joins) is limited. A specification of the supported SQL syntax as an EBNF grammar and more details about the implementation of the SQL compiler can be found in [20].

6 Related Work

A popular approach to integrate SQL into a programming language is the string-based approach, like JDBC in Java. The programmer writes SQL statements as string constants in the program (or implement a method to construct them during run time) which are passed to some database connection. As already discussed, this does not ensure type safety and might cause security leaks. Therefore, we consider in the following only approaches that try to avoid these problem by using specific abstractions of the base

programming language, in particular, of declarative programming languages.

Since relations stored in a relational database can be considered as facts defining a predicate of a logic program, logic programming provides a natural framework for connecting databases (e.g., see [9, 10]). Unfortunately, the well-developed theory in this area is not accompanied by practical implementations. For instance, Prolog systems rarely come with a standard interface to relational databases. An exception is Ciao Prolog which has a persistence module [8] that allows the declaration of predicates as facts that are persistently stored in a relational database. This hides the communication with the database and supports a simple method to query the relational database by logic programming methods. However, due to the untyped nature of Prolog, type safety is not ensured and queries are not explicitly distinguished from updates that are handled by predicates with side effects. A similar concept, but with a typed data access and a clear separation between queries and updates, has been proposed in [13] for Curry.

The latter concept has been extended in [3], where an entity-relationship (ER) model of the database is used to provide safe update operations that respect the constraints of the ER model and avoid the explicit use of foreign keys. Although this ensures type safety similar to our approach, the strongly typed combinators do not support all SQL constructs, in particular, no complex joins. Hence, the programmer must implement more complex queries using features of the programming language. While this is easily possible with higher-order operations like `map` and `filter`, this approach does not exploit the optimization features of database systems.

A disadvantage of such library-based approaches, like [21] for Haskell or [3] for Curry, is the fact that programmers must know, in addition to the base language and SQL, the specific methods and operations to implement SQL statements with these libraries. This can be avoided by a deeper embedding of SQL into the base language. A recent approach [25, 26] does this for Standard ML (SML). The language SML# extends to syntax of SML to write SQL queries as program expressions. In order to provide meaningful types to these expressions, the type system is extended with record polymorphism so that a principal type can be inferred for a type-correct SQL expression. This type is also used to check the type consistency of a concrete database with a given SQL query. The extension of the syntax and the type system of the base language requires the (re-)implementation of an SML compiler, in contrast to our lightweight preprocessor approach which can be combined with any Curry compiler. Due to the representation of SQL queries as SML# expressions, there are syntactic differences between SQL queries and their SML# equivalents. In our approach, these differences are avoided by considering SQL statements as integrated code instead of true Curry code. The use of record polymorphism in SML# supports the static typing of SQL expressions without a given database scheme. On the other hand, we use a high-level ER model of the database to type foreign keys (on the CDBI library level) and avoid the use of foreign keys in SQL queries. Our small syntactic extension of SQL avoids potential errors in the use of foreign keys and increases the expressiveness of queries similarly to the Ruby on Rails framework.

A strongly typed database access is also an objective of other functional languages integrating database query facilities, like Kleisli [28] or Links [7]. In contrast to our approach, these languages do not offer SQL syntax to formulate queries but require the use of other language constructs (e.g., list comprehensions) to formulate queries that are transformed into SQL.

7 Conclusions

We presented a framework to support a high-level and reliable access to relational databases from programs written in the declarative programming language Curry. The characteristic features of our framework are:

- Database access and manipulation of persistent data is formulated in a slight extension of SQL. Hence, it is easy to use since most programmers are already familiar with SQL. Moreover, one can exploit high-level features of SQL, like complex join operations or constraints.
- In order to detect ill-formed or ill-typed SQL statements at compile time, the SQL statements are preprocessed and translated into type-safe operations of a specific database access library.
- The library operations generate the corresponding SQL code at run time. Hence, the format of embedded data can be checked so that typical security leaks, like SQL injections, are avoided.
- Since the preprocessor uses a logical model of the database specified as an ER model, the use of relationships of this model inside conditions of SQL queries is supported. Hence, the potentially error-prone use of (untyped) foreign keys is avoided.

To implement this framework, we exploited some available infrastructure, in particular, to inject foreign code into Curry programs. The implementation of this preprocessor is available in the current releases of the Curry systems PAKCS [16] and KiCS2 [4]. In principle, such a framework could also be implemented for other strongly typed programming languages. Nevertheless, the declarative and high-level nature of Curry eased our implementation of this non-trivial compilation task.

For future work, it might be useful to support further database systems. In order to increase run-time reliability, it would be reasonable to check the consistency of the ER model used in the SQL compiler with the actual database scheme, e.g., when compiling a program, or before every program execution.

Acknowledgements. The authors are grateful to Jasper Paul Sikorra for the implementation of the foreign code integrator and Mike Tallarek for implementing the initial version of the CDBI libraries.

References

[1] S. Antoy & M. Hanus (2005): *Declarative Programming with Function Patterns*. In: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, Springer LNCS 3901, pp. 6–22.

[2] S. Antoy & M. Hanus (2010): *Functional Logic Programming*. *Communications of the ACM* 53(4), pp. 74–85, doi:10.1145/1721654.1721675.

[3] B. Braßel, M. Hanus & M. Müller (2008): *High-Level Database Programming in Curry*. In: *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, Springer LNCS 4902, pp. 316–332, doi:10.1007/978-3-540-77442-6_21.

[4] B. Braßel, M. Hanus, B. Peemöller & F. Reck (2011): *KiCS2: A New Compiler from Curry to Haskell*. In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, Springer LNCS 6816, pp. 1–18, doi:10.1007/978-3-642-22531-4_1.

[5] R. Caballero & F.J. López-Fraguas (1999): *A Functional-Logic Perspective of Parsing*. In: *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, Springer LNCS 1722, pp. 85–99, doi:10.1007/10705424_6.

[6] P. P.-S. Chen (1976): *The Entity-Relationship Model—Toward a Unified View of Data*. *ACM Transactions on Database Systems* 1(1), pp. 9–36, doi:10.1145/320434.320440.

[7] E. Cooper, S. Lindley, P. Wadler & J. Yallop (2006): *Links: Web Programming Without Tiers*. In: *5th International Symposium on Formal Methods for Components and Objects (FMCO 2006)*, Springer LNCS 4709, pp. 266–296, doi:10.1007/978-3-540-74792-5_12.

[8] J. Correas, J.M. Gómez, M. Carro, D. Cabeza & M. Hermenegildo (2004): *A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations)*. In: *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, Springer LNCS 3057, pp. 104–119, doi:10.1007/978-3-540-24836-1_8.

[9] S.K. Das (1992): *Deductive Databases and Logic Programming*. Addison-Wesley.

[10] H. Gallaire & J. Minker, editors (1978): *Logic and Databases*. Plenum Press, New York.

[11] M. Hanus (1997): *A Unified Computation Model for Functional and Logic Programming*. In: *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, doi:10.1145/263699.263710.

[12] M. Hanus (2000): *A Functional Logic Programming Approach to Graphical User Interfaces*. In: *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, Springer LNCS 1753, pp. 47–62.

[13] M. Hanus (2004): *Dynamic Predicates in Functional Logic Programs*. *Journal of Functional and Logic Programming* 2004(5).

[14] M. Hanus (2006): *Type-Oriented Construction of Web User Interfaces*. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, ACM Press, pp. 27–38.

[15] M. Hanus (2013): *Functional Logic Programming: From Theory to Curry*. In: *Programming Logics - Essays in Memory of Harald Ganzinger*, Springer LNCS 7797, pp. 123–168, doi:10.1007/978-3-642-37651-1_6.

[16] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre & F. Steiner (2016): *PAKCS: The Portland Aachen Kiel Curry System*. Available at `http://www.informatik.uni-kiel.de/~pakcs/`.

[17] M. Hanus & S. Koschnicke (2014): *An ER-based Framework for Declarative Web Programming*. *Theory and Practice of Logic Programming* 14(3), pp. 269–291, doi:10.1017/S1471068412000385.

[18] M. Hanus (ed.) (2016): *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-language.org`.

[19] S.H. Huseby (2003): *Innocent Code: A Security Wake-Up Call for Web Programmers*. Wiley.

[20] J. Krone (2015): *Integration of SQL into Curry*. Master's thesis, University of Kiel.

[21] D. Leijen & E. Meijer (1999): *Domain Specific Embedded Compilers*. In: *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, ACM SIGPLAN Notices 35(1), pp. 109–122, doi:10.1145/331960.331977.

[22] G. Mainland (2007): *Why It's Nice to be Quoted: Quasiquoting for Haskell*. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007)*, ACM Press, pp. 73–82, doi:10.1145/1291201.1291211.

[23] E. Martin-Martin (2011): *Type classes in functional logic programming*. In: *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*, ACM Press, pp. 121–130, doi:10.1145/1929501.1929524.

[24] S. Mazanek & M. Hanus (2011): *Constructing a Bidirectional Transformation between BPMN and BPEL with a Functional Logic Programming Language*. *Journal of Visual Languages and Computing* 22(1), pp. 66–89, doi:10.1016/j.jvlc.2010.11.005.

[25] A. Ohori & K. Ueno (2011): *Making standard ML a practical database programming language*. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, pp. 307–319, doi:10.1145/2034773.2034815.

[26] A. Ohori, K. Ueno, K. Hoshi, S. Nozaki, T. Sato, T. Makabe & Y. Ito (2014): *SML# in industry: a practical ERP system development*. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, pp. 167–173, doi:10.1145/2628136.2628164.

[27] S. Peyton Jones, editor (2003): *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press.

[28] L. Wong (2000): *Kleisli, a functional query system*. Journal of Functional Programming 10(1), pp. 19–56, doi:10.1017/S0956796899003585.