A Typeful Integration of SQL into Curry

Michael Hanus

University of Kiel
Programming Languages and Compiler Construction

WFLP 2016

Joint work with Julia Krone

Real World Applications with Databases

Access to relational databases in programming languages

1. Pass SQL statements as strings (JDBC,...)
   - \+ popular since SQL is well known
   - – source of security leaks in web applications
   - – SQL syntax errors at run time
   - – ill-typed database access or type casts
2. Language-specific database libraries (Haskell/DB,...)
   - \+ no syntax errors (and, maybe, no type errors)
   - \+ avoid security leaks with string checks/escapes
   - – expressiveness often limited (process data in programs)
   - – gap to SQL syntax (library combinators instead of SQL)

Our proposal: embed SQL in program code

- check SQL statements at compile time (preprocessor)
- compile-time detection of syntax and type errors
- exploit ER model of data, relations instead of foreign keys

Typeful SQL Embedding in Curry

Access to relational databases in programming languages

- implemented in functional logic language Curry
- ideas could be transferred to other higher-order typed languages
- concept: SQL queries are “integrated code”

```
-- Get name/age of students within a given age range:
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;''
```

- SQL code replaced by type-safe calls to DB library operations

Tools:

Curry, Integrated Code, CDBI libraries, ER models, SQL compiler

Curry [POPL'97, www.curry-language.org]

- declarative multi-paradigm language
  (higher-order concurrent functional logic language)
- extension of Haskell (non-strict functional language)
- better (high-level) APIs (GUI, web, database,...), eDSLs,...

Datatypes (values): enumerate all constructors

```
data Bool      =  True    | False
data List a    =  []      | a  : List a   -- [a]
```

Program rules: $f\ t_1 \ldots t_n \mid c = r$

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

```
last ::  [a] -> a
last xs | conc _ [x] == xs
        = x   where x free
```

Integrated Code (aka Quasi Quotation)

Concept:

- string in source program with own syntax rules
- enclosed in back ticks and ticks: ``` ``lang ...'' ```
  `lang`: specifies kind of embedded language
- *code integrator* replaces integrated code by Curry expression

Example: regular expressions in POSIX syntax

```
if s ``regex (ab*)+'' then ... else ...
```

Code integrator: exploits `RegExp` library and replaces string by

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

Another example: predicate for Curry identifiers:

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

Integrated Code

Currently embedded languages:

- regular expressions
- format printing (like C's `printf`)
- HTML and XML (with layout rules)
- SQL statements (new!)

  $\rightsquigarrow$ specific library support required!

CDBI: Curry Database Interface Libraries

Motivation

- abstract from concrete database access
- support type-safe access to database entities
- provide infrastructure for type-safe SQL embedding w.r.t. ER models

Base layer: raw database access

```
-- Return open connection to SQLite database:
connectSQLite :: String → IO Connection

-- Type of database actions:
type DBAction a = Connection → IO (SQLResult a)

-- Type of query results:
type SQLResult a = Either DBError a
```

CDBI: Curry Database Interface Libraries

Typed select operation

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

Arguments: SQL with “holes”, typed hole values, types of return values

Result: table of return values

```
data SQLValue = SQLString String | SQLInt Int | ...
data SQLType = SQLTypeString | SQLTypeInt | ...
```

Typed database access:

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

CDBI: Curry Database Interface Libraries

Next level: typed entities

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

Entity specification contains:

1. entity (table) name
2. column types
3. conversion (show/read) functions

Example: Student entity (generated from ER model)

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

studentDescription :: EntityDescription Student
studentDescription =
 ED "Student" [SQLTypeString,...,SQLTypeInt]
    (λ(Student name first num email age)  →  ...)
    (λ[SQLString name,...]  →  Student name first num email age)
```

CDBI: Curry Database Interface Libraries

Modeling SQL `where` clauses

```
-- Selection criteria
data Criteria = Criteria Constraint (Maybe GroupBy)

-- Greater-than constraint
(.>.) :: Value a → Value a → Constraint

-- Typed values: constants or DB columns
data Value a = Val SQLValue | Col (Column a)

int :: Int → Value Int
int = Val ∘ SQLInt

studentColumnAge :: Column Int   -- generated from ER model
```

Example: `...where Student.Age > 21`

```
Col studentColumnAge .>. int 21     ⇝ ok

Col studentColumnAge .>. float 3.4  ⇝ compile-time error
```

Entity-level type-safe selection: `getEntries`

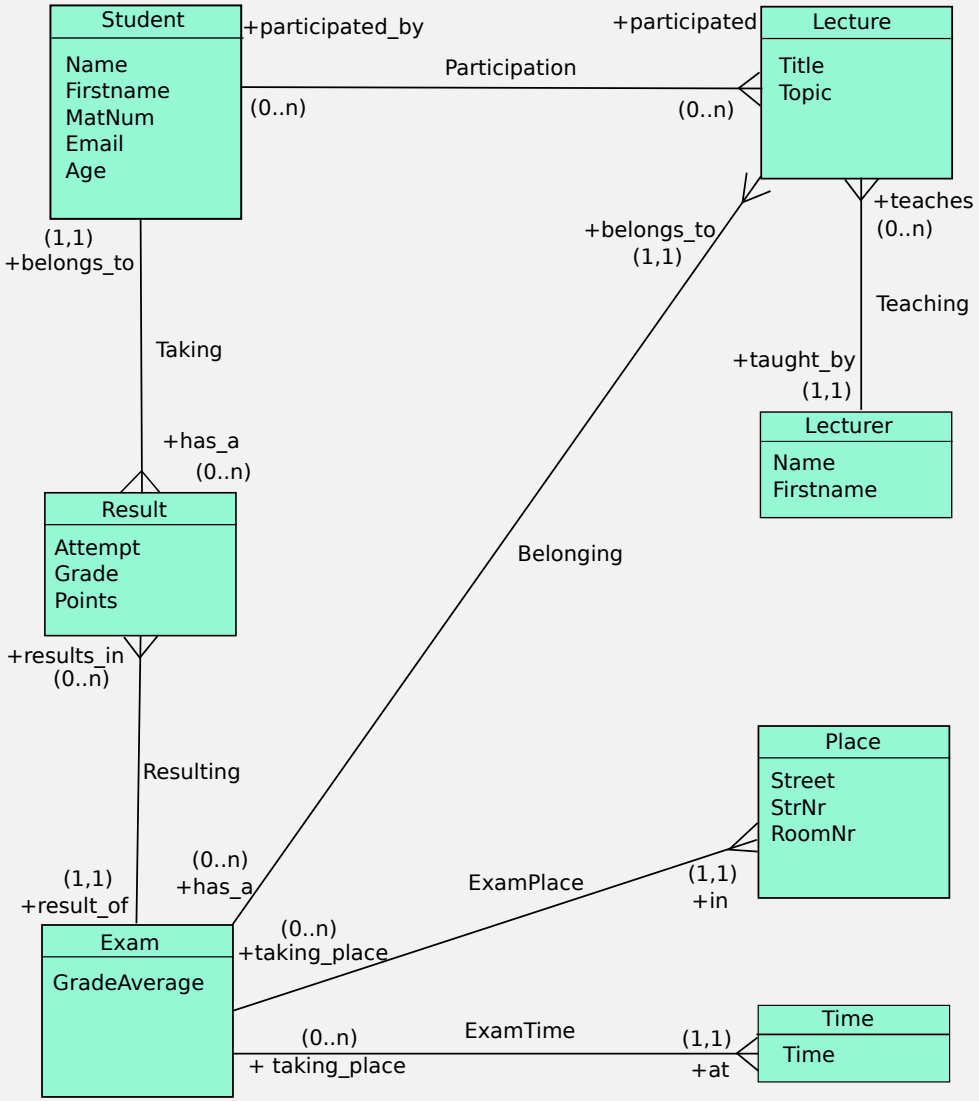
SQL query

```
Select * From Student
         Where Age > 21
         Order By Name Desc
         Limit 5;
```

corresponds to Curry expression

```
getEntries
  All                                    -- also:  Distinct
  studentDescription
  (Criteria (Col studentColumnAge .>. int 21) Nothing)
  [descOrder studentColumnName]          -- order specification
  (Just 5)                               -- limit result entries
```

Entity-Relationship Models



Entity-Relationship Models

Representation as Curry data term

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

data Entity    = Entity String [Attribute]
data Attribute = Attribute String Domain Key Null
...
```

ERD2CDBI translator

1. ER model $\mapsto$ relational data base (foreign keys,...)
2. Generates Curry module with entity descriptions
3. Generates *info file* for SQL translator

SQL Translator

Main tasks

- replace SQL string by Curry expression
- check conformity with ER model
- check types of columns and derive types for embedded Curry expressions

Get names of all students with a given age:

```
studNamesWithAge :: Int  → IO (SQLResult [String])
studNamesWithAge x =
  ``sql Select s.Name
        From Student as s
        Where s.Age = {x};''
```

SQL Translator

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

Translation:

```
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)
```

SQL query string (passed to DB at run time):

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

SQL Translator

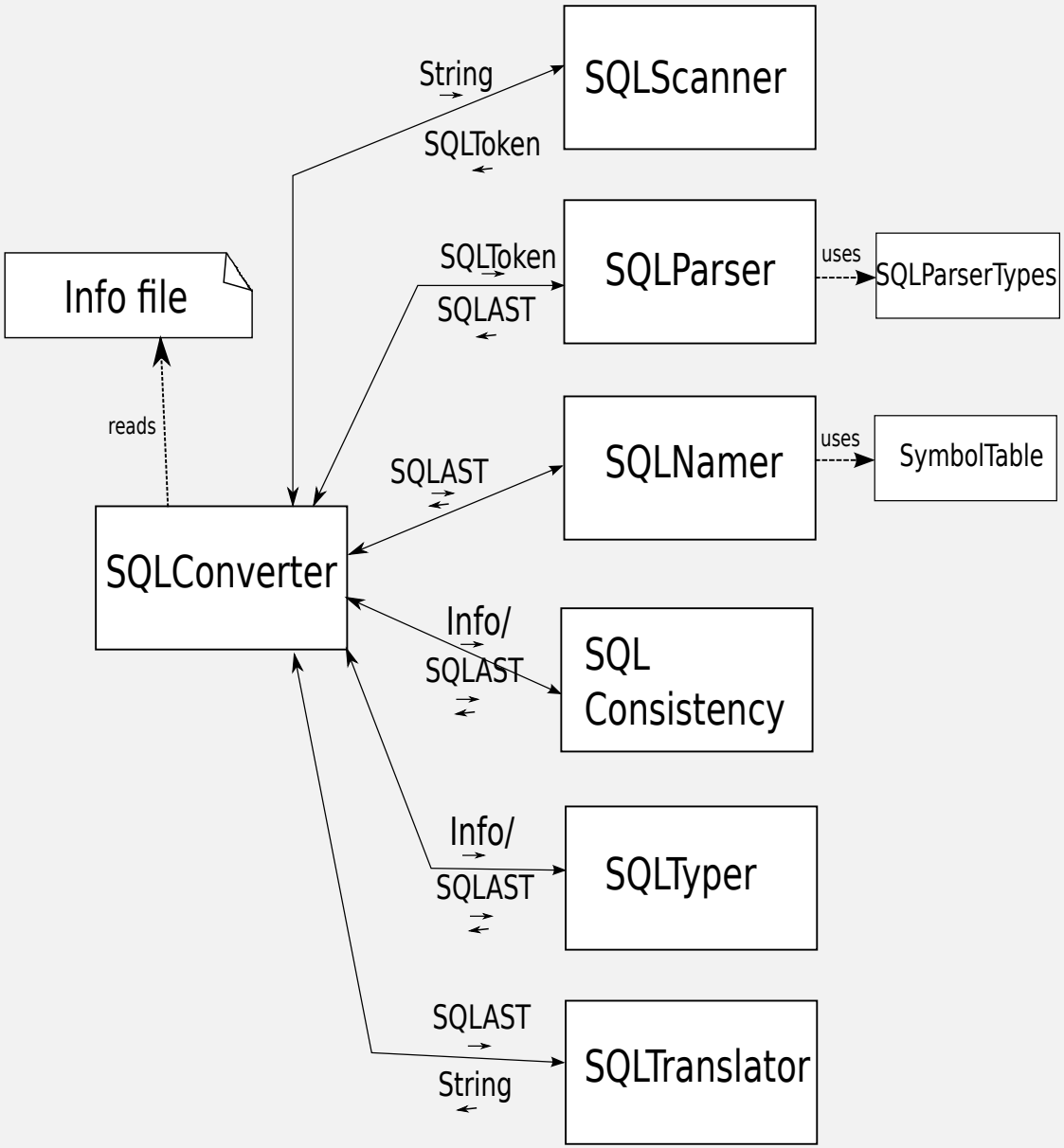
Extension to SQL: support for relations

```
-- Names/grades of students with grade better than 2.0
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;''
```

Condition `Satisfies e1 rel e2`:

- entities `e1` and `e2` are in relation `rel` of ER model
- avoid explicit uses of foreign keys

Structure of the SQL Translator



Conclusions

Typeful SQL Integration

- high-level and reliable access to databases
- easy to use due to SQL syntax
- compile-time detection of ill-formed or ill-typed SQL statements
- use of logical (ER) database model with relationships to avoid foreign keys

Future work:

- support further database systems
- check ER model against schema of actual database