Integration of SQL into Curry

Master Thesis

B.Sc. Julia Krone

Research Group:
Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

Advisors:
Prof. Dr. Michael Hanus
Dipl.-Inf. Jan Rasmus Tikovsky

Kiel, November 2015

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Weiterhin versichere ich, dass diese Arbeit noch nicht als Abschlussarbeit an anderer Stelle vorgelegen hat.

Kiel, den

Abstract

The interface of higher programming languages to databases is quite error-prone when SQL statements are passed as strings. In particular, this approach does not provide any type safety to the programmer. On the other hand, accessing the database with specialized library functions can hardly provide the conciseness of SQL while being readable and easy to use at the same time.
As the inclusion of different domain specific languages (DSLs) has to deal with similar problems, the functional logic programming language Curry provides a preprocessor to translate embedded statements of DSLs into pure Curry code before compilation. This approach meets both requirements: the statements keep their conciseness without having a negative impact on type safety.

This thesis presents a compiler, translating SQL statements into Curry code, as an enhancement for the preprocessor. To provide a large set of SQL expressions, an existing database library was extended.
Furthermore, an abstraction is proposed and implemented that allows the total avoidance of foreign keys in SQL statements and thus, provides an even more intuitive formulation of SQL queries.

Contents

I Introduction 1

1 Motivation 3

II Fundamentals 5

2 Curry 7
- 2.1 Programs 7
- 2.2 Evaluation strategy 11
- 2.3 Implementation 13

3 Structured Query Language 15
- 3.1 Relational Model 15
- 3.2 SQL Statements 16
- 3.3 SQLite 20

4 Curry-Tools 21
- 4.1 CurryPP 21
- 4.2 erd2curry 24
- 4.3 CDBI 27
- 4.4 AbstractCurry 31

5 Compiler 33
- 5.1 Scanner 34
- 5.2 Parser 34
- 5.3 Semantic Analysis 37
- 5.4 Code generation 37

III Implementation 39

6 Concept and Specification 41

7 CDBI-Enhancements **47**
7.1 Type safety regarding keys . . . 47
7.2 Auto incrementing keys . . . 49
7.3 Complex `Select` statements . . . 50
7.4 The executable `erd2cdbi` . . . 56
7.5 Modifications to `CDBI.Connection` . . . 57

8 SQL-Curry-Translation **59**
8.1 Structure . . . 59
8.2 Implementation . . . 63
8.2.1 Notation . . . 64
8.2.2 Information about the data model . . . 64
8.2.3 SQLConverter . . . 67
8.2.4 Lexical Analysis . . . 68
8.2.5 Syntactic Analysis . . . 68
8.2.6 Semantic Analysis . . . 75
8.2.7 Code Generation . . . 84

9 Integration into currypp **89**

IV Conclusion **91**

10 Summary **93**

11 Prospect **95**

Appendix **97**

Appendix A Installation and Usage **97**

Appendix B CDBI Extensions - Overview **99**

Appendix C Grammar **107**

Appendix D Examples **113**

Bibliography **117**

I

Introduction

Chapter 1

Motivation

The use of (relational) databases is indispensable in today's software systems. Especially web-based platforms have to deal with a large amount of data which has to be held permanently and reliably outside the main memory.
From a programmer's point of view, it is important that the employed programming language supports an easy-to-use database interface. In general, there are several requirements which seem to be contradictory at first sight. The queries are to be formulated in a concise, but still readable way, like an inclusion of SQL statements as strings would provide it. At the same time, it is preferable to make use of the specific features provided by the used programming language, e.g., type safety, which can better be achieved with a specialized library. Another valued advantage of library functions is the obviation of invalid and perhaps manipulative requests (e.g., SQL injections) which, once passed to the server, would end up in runtime errors or unwanted results.

A preprocessor combines all mentioned requirements and advantages. The query is formulated in SQL-syntax, embedded in the program and translated into the underlying programming language before compilation so that there is no need to resign language features.

This thesis provides the integration of SQL into the functional logic programming language Curry, applying the preprocessor approach. A corresponding preprocessing tool, `currypp`, was developed in [15] and can be easily extended for our purpose by implementing a compiler for SQL statements.
The currently used practice to access databases in Curry programs employs persistent predicates as it is presented in [1]. Persistent predicates exploit the logical paradigms of Curry, they are stored externally and their defining facts can change over time [8]. The main disadvantage of this approach is that complex queries are not performed directly on the database level. Instead, all entries of the given table are loaded into the program and thereafter filtered according to the presented criteria, resulting in a huge overhead of traffic. The CDBI library (*Curry Database Interface*) developed in [16] was a start to work

on this problem.
This thesis also evaluates and increases the functionality of the interface (CDBI) to obtain the possibility to use a larger subset of SQL from inside a Curry program. Doing so, special attention is paid to type safety and the reduction of traffic. The developed SQL compiler will finally translate SQL statements into functions provided by the CBDI library and, thus, a new database interface for the Curry language can be presented fulfilling high requirements w.r.t. safety and usability.

Additionally, an approach of abstracting foreign keys is presented which is based on entity-relationship (ER) models and constitutes an extension of pure SQL. This abstraction level is a further development of the ideas of ER-based database programming described in [1] and implemented with the `erd2curry`-tool, which will be partly described in chapter 4.

The following function gives an early example of the intended inclusion of SQL and the proposed abstraction.

```
sqlExample :: IO (SQLResult [String])
sqlExample = ``sql Select s.name
                   From Student As s
                        Inner Join Result As r
                        On Satisfies s has_a r;''
```

All parts will be explained in later chapters.

The remainder of this thesis is structured as follows: part II introduces the fundamentals, as there are the Curry programming language, the Structured Query Language (SQL) and compiler design in general. Furthermore, the Curry Database Interface and the tools `currypp` and `erd2curry` are explained briefly.
Part III represents the main part of the thesis. Chapter 6 outlines the requirements and specification of the integration. The extensions of the CDBI library are presented in chapter 7, before the design and implementation of the SQL compiler is discussed in chapter 8.
The last part (IV) summarizes the results and analyses potential prospects.

II

Fundamentals

Chapter 2

Curry

Curry is a universal multi-paradigm declarative programming language, developed for research, teaching and application of functional logic languages.
Declarative languages differ from imperative languages mainly by the level of abstraction employed. While the former describe the properties of a problem and its solution, the latter define how the problem can be solved. Declarative languages offer the principle of referential transparency, which means that the result of a computation is independent of the point in time it is evaluated, but just dependent of the terms that define it. This is possible due to the absence of side effects.
As a multi-paradigm declarative language as described in [9] and [10], Curry combines typical functional with logical paradigms and also provides the possibility of constraint programming. While functional languages are based on the lambda-calculus and feature e.g., higher-order functions and a static type system, logical languages rest upon predicate logic and provide e.g., logical variables, non-deterministic search and computation with incomplete information.
The following sections give an overview of the language itself as well as a small insight into the applied evaluation strategy. A more detailed description can be found in [13]. Since a basic familiarity with the Haskell programming language is assumed here and the Curry syntax is very similar to the syntax of Haskell, more importance is attached to the differences and the features rather particular for functional logic programming.

2.1 Programs

A Curry program is built of data type and function definitions. Since Curry is a strongly typed language and uses a polymorphic Hindley/Milner-like type system, most general types of functions can be reconstructed during compi-

lation via a type inference mechanism. Data declarations in Curry have the following general format:

data T $\alpha_1 ... \alpha_n = C_1 \tau_{11} ... \tau_{1m} \mid ... \mid C_k \tau_{k1} ... \tau_{kl}$

where T is the type constructor, C_1,.., C_k are data constructors, α_i are called type variables and τ_{ij} are types themselves. As an example we give an alternative definition of lists:

```
data List a = Empty | Cons a (List a)
```

Curry also allows type synonyms:

type T $\alpha_1 ... \alpha_n = \tau$

where the new type constructor T is a synonym for type τ.
Function definitions in Curry have the general from:

$$\mathrm{f}\, t_1 ... t_n = e$$

where f is the function name, $t_1, ..., t_n$ are data terms and e is an expression. Each function can be defined by several such rules. Definitions can also contain guards, i.e., the specified expression, e_i, is just executed in case the condition c_i is fulfilled:

$$\mathrm{f}\, t_1 ... t_n \mid c_1 = e_1$$
$$...$$
$$\mid c_n = e_n$$

The following example defines the append operation using the alternative list structure:

```
append :: List a -> List a -> List a
append Empty list2 = list2
append (Cons e list1) list2 = Cons e (append list1 list2)
```

Constructors for lists and the append operation are already predefined in Curry as `[]`, `(:)` and `++` respectively and will be used in the remainder of this chapter.

The next subsections discuss the main differences of functional logic programming to pure functional programming, which are free variables, constraints, non-deterministic operations and encapsulated search.

Constraints

Curry provides the possibility to solve equational constraints, which are denoted by the `=:=` operator (see listing 2.1 for an example). Equational con-

straints appear in conditions and have type `Success`, which has to be clearly distinguished from test equality (==) of type `Bool`. The former require strict equality and are solved by calculating a unifier for both sides of the equation so that the variables can be bound to the same ground data term. In contrast, in a term like `x == y` the values of both variables are expected to be known otherwise the execution will be suspended. In addition, test equality can have two different outcomes namely `True` and `False`, while equational constraints evaluate to `Success` if a corresponding unifier and therefore a binding can be found, and fail otherwise. This is sufficient because values in unsatisfied constraints will not be bound to the variables to generate a result later on, hence they can be discarded. The following sections clarify this proceeding.
Constraints of type Success can be combined by the &-operator:
`& :: Success -> Success -> Success`.
A trivial constraint which is always satisfied is denoted by `success`.

Free variables

Free variables are unbound variables that can be used to generate solutions. They are declared using the keyword `free` and have the same scope as local declarations. Computations with free variables provide a way to compute not only the result of an expression, but also the parameters which let the result become true. Listing 2.1 shows three functions all of them calculating all possible prefixes of a given list using free variables. Note that free variables can be declared in two different ways either using the keyword `where` (on right-hand-sides) or the keyword `let`. As the third function shows, free variables can also be anonymous if they are not needed elsewhere.

```
prefixOf :: [a] -> [a]
prefixOf xs | ys ++ zs =:= xs = ys where ys, zs free

prefixOf1 :: [a] -> [a]
prefixOf1 xs | let zs free in ys ++ zs =:= xs = ys
                  where ys free

prefixOf2 :: [a] -> [a]
prefixOf2 xs | ys ++ _ =:= xs = ys where ys free
```

Listing 2.1: Free variables

These functions will generate all possible solutions for the equation specified in the constraint, e.g.,

```
> prefixOf [1,2,3]
[]
[1]
[1,2]
```

```
[1,2,3]
```

The values are found by binding the free variables to all possible constructor-terms. This is explained in more detail in section 2.2.

Non-deterministic operations

As seen in the above example, Curry supports non-determinism, i.e., functions can gain more then one fitting result, returning different results in different calls. Another way to achieve this behaviour is by overlapping rules. E.g., the function `singletonList` in listing 2.2 returns non-deterministically a member of a given list as a single-element list and the list itself in case it is empty.

```
singletonList :: [a] -> [a]
singletonList [] = []
singletonList (x:_) = [x]
singletonList (_:xs) = singletonList xs
```

Listing 2.2: Overlapping rules

For each non-empty list the second and the third rule apply and thus, all elements are retrieved recursively:

```
> singletonList [1,2,3]
[1]
[2]
[3]
[]
```

In addition, Curry's special operator `?` provides an easy manner to introduce non-determinism. It is defined as follows:

```
(?) :: a -> a -> a
 _ ? y = y
 x ? _ = x
```

and returns non-deterministically one of its arguments. This allows a reformulation of the above function:

```
singletonList' [] = []
singletonList' (x:xs) = [x] ?  singletonList' xs
```

Attention has to be paid when using non-determinism in IO-operations. Like in Haskell these are implemented using a monadic structure which is seen as the "outside world" and each single IO-action signifies a modification to this world. To keep the principle of referential transparency it can not be allowed to maintain two versions of the outside world at any moment, so that non-determinism inside IO-operations is not supported.
Curry provides different approaches to handle this problem. One of them is the use of set functions [11], i.e., each function $f: \tau_1 \to ... \to \tau_n \to \tau$ is assigned a corresponding set function $f_s: \tau_1 \to ...\tau_n \to 2^\tau$ which represents the possible

values as a multiset and is evaluated on demand. As an example consider the following set function for the above defined function `singletonList`:

```
singletonSet :: [a] -> [[a]]
singletonSet xs = sortValues (set1 singletonList xs)
```

where `sortValues` returns all computed values as an ordered list:

```
> singletonSet [1,2]
[[],[1],[2]]
```

The advantage of this approach is that it is clearly defined which non-determinism is encapsulated, namely, the one of the associated function and not the one possibly appearing in its arguments. Thus, the function call

```
singletonSet (prefixOf [1,2])
```

is still non-deterministic and returns the values:

```
[[]]
[[],[1]]
[[],[1],[2]]
```

2.2 Evaluation strategy

The evaluation strategy is just outlined roughly here. We will explain briefly the most important theoretical terms. For a formal description and corresponding proofs see [9] and [10].

Definitions of functions and constructors in Curry programs have to be constructor based, i.e., the pattern on the left-hand side of any rule must not contain function calls, but just constructors and variables. This is an important requirement for efficient evaluation strategies. Terms are evaluated non-strict. Non-deterministic operations employ a call-time-choice semantic, which basically means that the value of a non-deterministic function call is computed exactly once even if it is referenced several times afterwards. For an example, consider the following function, which duplicates a list and concatenates it to the original:

```
doubleList :: [a] -> [a]
doubleList xs = xs++xs
```

and the call

```
> doubleList (prefixOf [1,2])
[]
[1,1]
[1,2,1,2]
```

As can be easily seen, the non-deterministic argument is just evaluated once, otherwise results as `[1,2,1]` would also be possible. As this behaviour was not explicitly exploited for this thesis, we will not give further details.

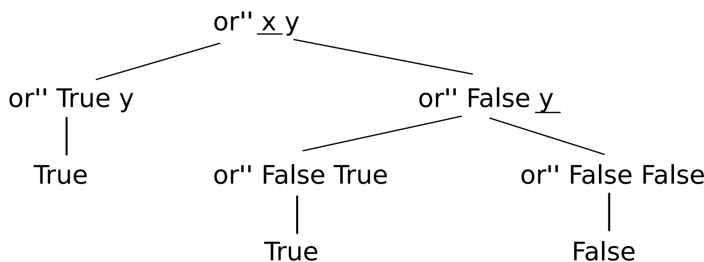

Figure 2.1: definitional tree for function `or''`

The strategy that mainly characterizes the evaluation of Curry programs is *Weakly Needed Narrowing.* Narrowing combines term rewriting known from functional programming with unification and therefore instantiation of variables as known from logical programming languages, i.e., to enable term rewriting logical variables first have to be bound to values (constructor terms).
Since this implies that all possible values have to be tried, the strategy is refined by further requirements leading to Needed Narrowing. This evaluation is lazy, just considering variables at positions that are necessary to obtain a result. Therefore, it can only be performed on operations that are *inductively sequential.* For these operations it is, informally said, always defined which argument is needed. For instance, consider the two alternative implementations of the logical operator `or` in listing 2.3. The first one provides two fitting rules for `or' True True` so that it is undefined which argument has to be evaluated first in a call like `or' x y`. The second implementation requires the first argument to be evaluated in first place to choose a subsequent rule. Thus, the second function definition is inductively sequential and deterministic.

```
or' :: Bool -> Bool -> Bool
or' True _ = True
or' _ True = True
or' False False = False

or'' :: Bool -> Bool -> Bool
or'' True _ = True
or'' False True = True
or'' False False = False
```

Listing 2.3: Different implementations of the logical operator `or`

Inductively sequential function definitions can also be represented by a definitional tree in which each rule appears exactly once. Such a tree is depicted in figure 2.1 for the above defined function or”. Needed Narrowing is shown to be sound, complete and minimal (w.r.t. the number of solutions) and successful derivations can not be found in less steps than done by this strategy [12].
So far we excluded overlapping rules introduced above in listing 2.2. Because their definitions are not inductively sequential, there is more than one possible

rule to apply and more than one possible argument to evaluate in first place. To cope with that problem, the strategy was extended to Weakly Needed Narrowing, which non-deterministically chooses one of the possible rules.

Three more terms have to be mentioned when talking about the evaluation strategy of Curry. The first one is the concept of shared variables, i.e., a variable is just evaluated once and the computed value is referenced in subsequent occurrences of the same variable. This is important for efficiency reasons and conforms with the applied concept of call-time-choice in case of non-deterministic operations.
The second concept is residuation, which means the suspension of operation as long as a referenced variable is unbound. This strategy is employed for external function calls (e.g., to the arithmetic '+'- operation).
At last Curry provides functional patterns to write more efficient and readable code. A functional pattern is a function reference on the left-hand side of a rule which was excluded earlier by the requirement of constructor based terms. Therefore those patterns are replaced on demand by the constructor terms they represent. An example is given by the following partially defined function, which returns the last argument of an non-empty list:

```
last :: [a] -> a
last (xs++[e]) = e
```

2.3 Implementation

As Curry provides some features from functional languages as well as from logic programming languages, there also exist two different implementations. The first one, *PAKCS* - The Portland Aachen Kiel Curry System [1], provides a translation into Prolog. This approach is quite simple to implement and benefits from the increasingly efficient implementations of Prolog and the availability of constraint solvers in most Prolog implementations. The disadvantage is that such a translation is determined to use Prologs backtracking strategy, which delimits the implementation of search strategies.
The second approach, presented in [2], describes the translation into Haskell programs, which is implemented in *KiCS(2)* - The Kiel Curry System [2]. This implementation can easily provide different search strategies and exploits the efficiency of Haskell in purely functional program parts. A special effort had to be made on the representation of non-determinism.
Both approaches provide an interactive environment for evaluations.
The integration of SQL developed in this thesis can also be used with both implementations.

[1]see http://www-ps.informatik.uni-kiel.de/currywiki/start
[2]see http://www-ps.informatik.uni-kiel.de/kics2/

Chapter 3

Structured Query Language

The Structured Query Language (SQL) is a well-known database language to interact with relational databases. It is based on the mathematical theory of the relational model. The first version of SQL was developed in 1974/75 at IBM. Since 1983 the language passed an enduring standardization process, `SQL:2003` being the most current result. Despite of this effort there are a lot of varying dialects since different products implement different subsets of the standard, but none of them provides the entire set of statements. In addition, most dialects are still based on the former version `SQL-92` [17].
This thesis will also provide a kind of its own dialect, which is restricted by the database software `SQLite3` presented at the end of this chapter and the database library CDBI, which is described in the following chapter. The next two sections give a short introduction to all supported statements. A complete overview can be found in appendix C.

3.1 Relational Model

The relational model as description technique for a database structure and the relational algebra to formulate requests on it was first published in [5]. The main terms applied to describe the structure of a database with help of the relational model are "relation", "tuple" and "attribute", which are often referred to as table, row and column respectively when talking about SQL. Operators on the data model were always defined as taking one or more relations as input and generating exactly one output relation. Originally the following operators were presented:

- `restrict`: restriction of a relation to tuples that fulfill a certain condition
- `project`: restriction to certain attributes contained in the resulting relation

- `product`: cartesian product of two relations
- `intersect, union, difference`: set operations (notice that a relation in sense of the relational algebra is seen as a set of tuples without duplicates and order)
- `Join`: combination of two relations depending on the value(s) of one or more attribute

As claimed in [6], not all of the precise mathematical definitions and assumptions are considered in implementations of SQL.

3.2 SQL Statements

The large amount of SQL statements can be divided into three groups:

- Data Definition Language (DDL) - manipulation of the database structure as creating or deleting tables
- Data Manipulation Language (DML) - statements to query and change contents
- Data Control Language (DCL) - modifications on data security and user privileges

The remainder of this thesis concentrates exclusively on the DML, all supported statements are presented below. The resemblance to the mathematical definitions above should be easily noticeable. The next chapter will clarify why a support of DDL statements is not necessary.

We want to demonstrate the different types of SQL statements with the help of a little exemplary database. It consists of a table named Student with four columns and three entries (rows) and a second table named Result with three columns and five entries.

Key	Name	Firstname	Age
1	Muster	Max	18
2	Maier	Anna	25
3	Schulz	Tom	20

Key	StudentKey	Grade
1	1	1.3
2	2	3.7
3	2	5.0
4	1	2.0
5	3	1.0

Table 3.1: database example: Student table (l) and Result table (r)

`Select` statement

The `Select` statement is used to read a specified subset of data from the database.

```
Select Firstname, Name From Student;
```

represents a `project` operation and results in the following relation:

```
(Max, Muster)
(Anna, Maier)
(Tom, Schultz)
```

A equivalent to the `restrict` operation is given by:

```
Select * From Student Where Age = 20;
```

The result is just one row from the exemplary database:

```
(3, Schultz, Tom, 20)
```

An exemplary join operation is given by the following statement:

```
Select s.Name, r.Grade From Student As s
                        Inner Join Result As r
                        On s.Key = r.StudentKey;
```

It returns tuples of the student's name and the retrieved grade:

```
(Muster,1.3)
(Muster, 2.0)
(Maier, 3.7)
(Maier, 5.0)
(Schultz, 1.0)
```

The keyword `As` defines alias names for tables to identify column names unambiguously. This is even more important when using the product operation on two instances of the same table:

```
Select s1.Name, s2.Name From Student As s1, Student As s2;
```

This query returns the cartesian product of all student names:

```
(Muster, Muster) (Muster, Maier) (Muster, Schultz)
(Maier, Muster) (Maier, Maier) (Maier, Schultz)
(Schultz, Muster) (Schultz, Maier) (Schultz, Schultz)
```

An exemplary set operation is used in the following statement:

```
Select Firstname From Student Where Age > 20
 Union
Select Firstname From Student Where Age < 20;
```

which returns the two corresponding names:

```
(Anna)
(Max)
```

An arbitrary combination of all operations is possible.
There are also some additional supported operators that have been defined in later extensions of the relational model, among them aggregation functions as

`Sum` or `Avg`, `Group-by`, `Having` and `Order-by`. The last one is not actually a relational operator in particular, because it returns an ordered sequence instead of an unordered set [6]. The example below demonstrates the use of these operators:

```
Select s.Firstname, Avg(r.Grade)
  From Student As s Inner Join Result As r
   On s.Key = r.StudentKey
  Group By s.Name
  Having Avg(r.Grade)
  Order By s.Firstname;
```

`Group-by` assembles rows that hold the same value for the given column while `Having` can express conditions, particularly aggregations, on these groups. Aggregation functions used as projection return just one value for a set(group) of rows, without a `Group-by` clause even just one value for all rows in the table. So the example above returns the following result:

```
(Anna, 4.35)
(Max, 1.65)
(Tom, 1.0)
```

`Insert`, `Update` and `Delete` statement

The operations `Insert`, `Update` and `Delete` modify the content of database tables. Thus, they are not directly based on relational algebra, but on a relational assignment operator. The theory will be omitted here, but can be found in [6]. We will just demonstrate the basic syntax using the above defined exemplary database:

```
Insert Into Result (Key, StudentKey, Grade)
       Values (6, 2, 1.7);

Update Student Set Firstname = 'Tim', Age = 21
               Where Name = 'Schultz';

Delete From Result Where Grade = 5.0;
```

`Insert` statements write one or more rows to the given table. In case of left out null-values the list of columns is mandatory to clarify which columns the values refer to.
`Update` statements are able to change one or more columns of the given table. Which rows are affected is specified by an optional condition.
The `Delete` statement erases all rows of the given table that fulfill the optional condition.

Conditions

This sections gives a few details about the conditions used in `Select`, `Update` and `Delete` statements. Conditions can be simple binary comparison operations between two columns, values or value and column. Each condition can be combined with another one by using the logical operators `And` and `Or`. In addition the following operators can be contained:

- `val1 Between val2 And val3` is the same as `val1 >= val2 And val1 <= val3`
- `val In set` tests whether value `val` is member of the set
- `Exists subquery` models the existential quantifier, evaluates to "True" if the subquery (shortened `Select` statement) returns at least one row
- `Is Null`, `Not Null` check whether a column contains a null-value or not respectively

Null-values play a complicated role because they are not compatible with the two-valued-logic applied by the relational model. That is why the later implementation presented in part III prohibits them in combination with any but the last two operators.
As already mentioned, the standard provides much more possibilities to express conditions.

Transactions

To provide an interface equipped for multi-user environments, the implemented SQL dialect also supports transactional statements. Transactions guarantee a set of properties, which are often referred to as ACID: atomicity, consistency, isolation and durability. Atomicity ensures that either all parts of a transaction are performed or none of them is. In case any part of the transaction fails, the complete transaction fails. Consistency guarantees that the state of the database before and after an transaction is valid. Isolation assures that the state of a transaction is not visible to other operations until the changes are committed. Durability requires that committed changes are saved persistently.

The following transactional statements are supported:

- `Begin` - indicates the start of a transaction
- `Commit` - concludes a transaction and writes changes to the database in case all actions were finished successfully

- `Rollback` - returns to the state before the start of the transaction, all changes are discarded

3.3 SQLite

SQLite[1] is not only a dialect of SQL but also a lightweight and free database software. Working on a single disk file, it is independent of a server component and can be directly embedded into the application.
SQLite3 implements most of the `SQL-92`-standard. As it is so far the only database software supported by the CDBI library, explained in the next chapter, it sometimes limited but often influenced the SQL-dialect implemented in this thesis. However, an extension of CDBI with another dialect and database software is not restricted by the implemented compiler since the library functions allow an independent intermediate representation of SQL statements.

[1]https://sqlite.org/index.html

Chapter 4

Curry-Tools

This chapter introduces all tools and specialized libraries written in Curry that are used in this thesis or were enhanced during the implementation.
The first section describes the preprocessor `currypp`, followed by the tool `erd2curry`, which translates entity-relationship diagrams to Curry programs. The last two sections introduce the CDBI library and the AbstractCurry library respectively.

4.1 CurryPP

`currypp` is a preprocessor for domain specific languages (DSLs) embedded into Curry programs, which was developed in [15]. So far it allows the integration of regular expressions, formatted printing, XML and HTML expressions. The compiler developed in this thesis provides an enhancement for SQL statements. Expressions in domain specific languages can be embedded into the Curry code using the following syntax:

```
``langtag expression''
```

where `langtag` is a language tag (e.g., "regex" for a regular expression) that specifies which DSL is used. Language tag and expression have to be surrounded by at least two accent graphs at the beginning and two single quotes at the end. Any single quote or accent grave inside the expression itself increases the number of surrounding quotes and accents by one. Between the initiating accents and the language tag no white space, newline or tab is allowed, though the language tag can be followed by any amount of those characters. Any term inside the expression surrounded by <> in case of regular expressions and formatted printing or {} in case of XML and HTML is interpreted as pure Curry code and is left unchanged.
To give a concrete example the following regular expression embedded in a Curry function checks whether a passed string contains the word "curry"/

"Curry":

```
checkCurry :: String -> Bool
checkCurry s = s ``regex [a-zA-Z]*(c|C)urry[a-zA-Z]*''
```

The expression is translated by the preprocessor into the match function and data types provided by the RegExp library available in both Curry implementations and thus, into pure Curry code:

```
checkCurry :: String -> Bool
checkCurry s =
     s `match` ([Star
                 ([Bracket [Right (('a'),('z')),
                            Right (('A'),('Z'))]]),
                Xor ([Literal ('c')])
                    ([Literal ('C')]),
                Literal ('u'),
                Literal ('r'),
                Literal ('r'),
                Literal ('y'),
                Star ([Bracket [Right (('a'),('z')),
                                Right (('A'),('Z'))]])])
```

`currypp` is provided by the current version of PAKCS as well as KiCS2 and can be invoked by the following command:
`currypp org-filename input-file output-file [options]`
The option `--foreigncode` starts the translation of embedded foreign code found in the input file and saves the translation as the output file. Option `-o` stores the result also in a file named org-filename.currypp.
It is important to mention that the input file is allowed to include snippets of any supported DSL at the same time beside parts of pure Curry code (which is not altered). The pieces of different languages are separated, preprocessed and joined again afterwards as shown in figure 4.1. This procedure makes it easy to extend the preprocessor and hence, support further DSLs.

During the development of the preprocessor a parser monad was implemented providing a profound basis for further translators. It is basically given by the data structure `PM` shown in listing 4.1 and combines a monad for errors (`PR`) with a monad for warnings (`WM`). A parser monad (`PM`) is parameterized over the result type. At any time it can contain a list of warnings and either a result or a list of errors. Each error and each warning is defined by its position (actually the position of the piece of integrated code) and a message. A position is given by the name of the file, an absolute position (counting every character), line and column number.

```
-- parser monad
type PM a = WM (PR a)
-- warning monad
data WM a = WM a [Warning]
type Warning = (Pos,String)
```

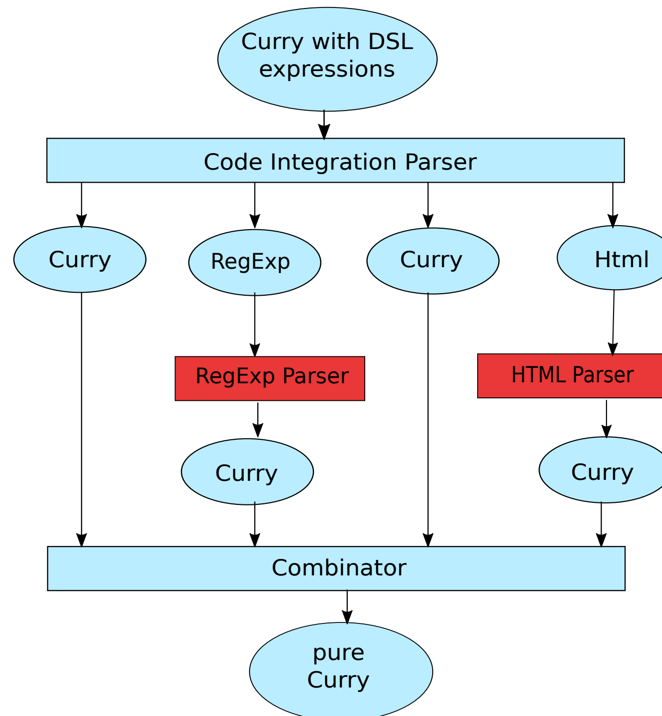

Figure 4.1: Workflow of the currypp tool. (This figure is mainly taken from [15].)

```
-- error monad
data PR a = OK a | Errors [PError]
data PError = PError Pos String
-- position
data Pos = Pos Filename Absolute Line Column
```

Listing 4.1: data type definition for Parser Monad

Several methods to access and modify an instance of the monad are provided. To ensure that surrounding code starts at the same line as before the preprocessing, each compiler has to provide a single line translation of the integrated code. In case the original embedded expression was placed over more than one line, the corresponding amount of newlines are inserted in order to keep the structure of the surrounding code.

To extend the preprocessor, a simple interface has to be implemented:

```
parse :: Pos -> String -> IO (PM String)
```

The method `parse` takes the position of the chunk of integrated code and the expression itself as a string and returns a parser monad of type string, which contains either the result i.e., pure Curry code as a string, or a list of errors.

4.2 erd2curry

The tool `erd2curry`, which is also available in all Curry implementations, was introduced in [1]. It is aimed at translating entity-relationship-models (ER-models), as they were discussed in [3], into Curry programs.
The graphical notation of an ER-model is given by an entity-relationship-diagram, which consists of entities specified by a name, attributes and relations to other entities, which are constrained by cardinalities. Figure 4.2 shows an exemplary model of a university, which will be used in any further example in this thesis.
To represent an ER-model in Curry the data type ERD is employed:

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

data Entity = Entity String [Attribute]

data Attribute = Attribute String Domain Key Null

data Key = NoKey | PKey | Unique

type Null = Bool

data Domain = IntDom (Maybe Int)
            | FloatDom (Maybe Float)
            | CharDom (Maybe Char)
            | StringDom (Maybe String)
            | BoolDom (Maybe Bool)
            | DateDom (Maybe ClockTime)
            | KeyDom String
            | UserDefined String (Maybe String)

data Relationship = Relationship String [REnd]

data REnd = REnd String String Cardinality

data Cardinality = Exactly Int
                 | Between Int MaxValue

data MaxValue = Max Int | Infinite
```

The mapping with the information given in the diagram should be straightforward. A relationship is given by its bidirectional name and its two ends, which are represented by the name of the entity, the unidirectional relation name and the cardinality which is either a range or an exact number. Attributes consist of a name, a domain, a flag whether they are a primary key, no key or unique and another flag indicating whether they can be set to NULL or not.

However, the translation finally performed by the `erd2curry` tool is a little bit more subtle. It is split into three steps. The first one consists of the direct transformation of the diagram into the `ERD` data type introduced in the listing above. The result is stored in a file named <model_name>_ERDT.term. The tool can be called directly with a model represented as an `ERD` term (in this case this first step will be omitted) or with the XML-representation as produced by the Umbrello UML Modeller (note, that this last option is no longer actively supported).
The second step is the one which is important for the later described CDBI library as well as for the tools developed in this thesis, so it is explained in more detail. The aim is to modify the model in a way, that it can be directly mapped to the finally needed database tables including their columns (i.e., the relations and attributes in a relational model). In case of entities it is a simple process: each entity represents a table, its attributes represent the columns and an additional integer value is added as primary key to provide a unique access scheme.
Relationships have to be translated in a different way depending on their cardinality. The following relationships have to be considered:

- **(1,1):(0,1)-relationship**: a foreign key column is introduced into the second entity, referencing the key of the first entity, null-values are not allowed, the new attribute has to be `Unique`
- **(0,1):(0,1)-relationship**: a foreign key column is introduced into the second entity, referencing the key of the first entity, null-values are allowed, the new attribute has to be `Unique`
- **(0,1):(1,n)-relationship, ($n > 1$)**: a foreign key column is introduced into the second entity, referencing the key of the first entity, null-values are allowed
- **(1,1):(0,n)-relationship, ($n > 1$)**: a foreign key column is introduced into the second entity, referencing the key of the first entity, null-values are not allowed
- **(0,m):(0,n)-relationship, ($m, n > 1$)**: a new entity is introduced containing two foreign key columns which are referencing the keys of both participating entities, the relationship itself is replaced by two new (1,1):(0,n)-relationships

Relationships with a minimum of more than 0 on both sides are not supported. To clarify this procedure listing 4.2 shows a partial result for the ER-diagram in figure 4.2. While the entity Result is modified by the (0,n):(1,1)-relationships "Taking" and "Resulting", the (0,m):(0,n)-relationship "Participation" is translated into a new entity and two new relationships replacing the old one.

```
Entity "Result"
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "Attempt" (IntDom Nothing) NoKey False,
        Attribute "Grade" (FloatDom Nothing) NoKey True,
        Attribute "Points" (IntDom Nothing) NoKey True,
        Attribute "StudentTakingKey"
                  (KeyDom "Student")
                  NoKey
                  False,
        Attribute "ExamResultingKey"
                  (KeyDom "Exam")
                  NoKey
                  False]

Entity "Participation"
       [Attribute "StudentParticipationKey"
                  (KeyDom "Student")
                  PKey
                  False,
        Attribute "LectureParticipationKey"
                  (KeyDom "Lecture")
                  PKey
                  False]

Relationship ""
             [REnd "Student" [] (Exactly 1),
              REnd "Participation" "participates"
                                   (Between 0 Infinite)],
Relationship ""
             [REnd "Lecture" [] (Exactly 1),
              REnd "Participation" "participated_by"
                                   (Between 0 Infinite)]
```

Listing 4.2: exemplary translation by erd2curry

It is important to notice that the added attributes are always named according to the same pattern: <referenced entity><relationship name>"Key".
Another detail important to mention is that the relationships themselves stay part of the ERD-term even after the successful translation (except the replaced m-to-n relationships).
The third step performed by erd2curry is the generation of Curry code as used by the former database library [8]. Thus, this step is unessential for this thesis and will be omitted.

4.3 CDBI

CDBI, Curry Database Interface, was developed in [16]. It provides the access to an SQLite3-database via library functions. Despite the fact that all data is stored as strings at the database level, the interface assures type safety to the programmer.
Just as the approach in [8], CDBI is based on the `erd2curry` tool and thus, on a data model provided as an ER model. But while the former approach makes use of dynamic predicates as explained in Chapter 1, the recent library generates its own data types from the `ERD`-representation.

Calls to library functions are internally translated into an SQL statement, which is then executed on an SQLite3 database instance. At the highest level the following operations on database tables are currently available:

- selection of complete rows of one or more tables (using cross joins) with conditions and `Order-by` clause
- insertion of new rows
- update of one or more columns of a special row (identified by its key) or several rows (selected by a given condition)
- deletion of rows identified by a condition
- execution of statements using a transaction including an automatic `roll-back` in case of an error

Since it is based on the `erd2curry` tool, the interface does not provide any operations to modify the data model.
All data types necessary to use the interface with any database in particular are generated by parsing the `ERD` term that was issued by `erd2curry`.
The project is organized in the following modules:

- `CDBI.ER`: main module, provides the high-level operations mentioned above
- `CDBI.Criteria`: provides data types and functions to express conditions
- `CDBI.Connection`: low-level module organizing the connection to the database
- `CDBI.Description`: contains the definition of data types that have to be generated for each entity in the used model
- `ERD2CDBI`: translation module: takes a transformed `ERD`-term and generates all needed data types according to the format given by the `Description`-module; additionally sets up the database

Translation of ERD-terms

The translation performed by the ERD2CDBI-module ignores the relationships in an ERD-term and only considers the entities. Each entity is lifted to a data type assembled by the types of its columns. For each such data type an EntityDescription is generated, whose general declaration is shown below.

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

An EntityDescription contains the name of the entity, a list of column types and two conversion functions. The type SQLValue is a disjunction of all types that can appear as column types according to ERD-terms except the user-defined type which is not supported. SQLType serves as an identifier to express which data type the user expects. Both are used for a proper type-string-conversion, that has to be performed when reading from or writing to the database. The first conversion function is used for preparing an entity for writing-operations, the second one generates an entity out of the data read from the database. Columns that can contain null-values have to be translated with a Maybe type at this stage.
An example for the entity Student is given in Listing 4.3.

```
data Student = Student StudentID
                       Int
                       String
                       String
                       String
                       (Maybe Int)

data StudentID = StudentID Int

studentDescription :: EntityDescription Student
studentDescription =
 ED "Student"
 [SQLTypeInt, SQLTypeInt, SQLTypeString,
              SQLTypeString, SQLTypeString, SQLTypeInt]
 (\(Student (StudentID key) matNum name firstName email age)
       -> [SQLInt key, SQLInt matNum, SQLString name,
           SQLString  firstName, SQLString email,
           (sqlIntOrNull age)])
 (\[SQLInt key, SQLInt matNum, SQLString name,
     SQLString firstName, SQLString email, age] ->
     Student (StudentID key) matNum name
              firstName email (intOrNothing age))
```

Listing 4.3: generated data for Student entity

Additionally every column is translated applying the pattern:

```
<entity name>"Column"<column name> :: Column a
```

where `a` is the column type. The data type `Column a` provides the name of the column and the table for a later translation to SQL.
All generated data is saved in a file named CDBI<name of ERD>.curry.

Type safety

When working with complete entities, the data types introduced in the last section already provide good type safety. Nevertheless, this requirement has to be assured in a different way when expressing conditions e.g., the comparison of a constant value with a column of a certain type. For instance, the constraint `Age > 18` is represented in CDBI as:

```
((col studentColumnAge) .>. (int 18))
```

The constructor function for the greater-than operation is defined by

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

Thus, type safety is ensured as both, column and constant value, have to be of type `Value a`.

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

The integer value is used for renaming.
Type safe constructor functions are provided for columns and all types of constant values :

```
col :: Column a -> Value a
colNum :: Column a -> Int -> Value a
int :: Int -> Value Int
string ...
```

Internally all operators are converted to type `CValue`, to avoid that constraints themselves have to be parameterized:

```
type CValue = Value ()

data Constraint
      = IsNull           CValue
      | IsNotNull        CValue
      | BinaryRel RelOp  CValue CValue
      | ...
```

More details can be found in [16] and in Chapter 7 where the extensions to the interface are described.

Formulating queries

To demonstrate the use of CDBI, an example is given corresponding to the SQL request

```
Select * From Student Where Age > 18 Order By name Desc;
```

i.e., the selection of all students who are older than 18 ordered by their names in descending alphabetical order.

```
example :: IO (Result [Student])
example = do
  conn <- connectSQLite "Uni.db"
  result <- getEntries studentDescription
            (Criteria
              ((col studentColumnAge) .>. (int 18))
            [descOrder studentColumnName]) conn
  return result
```

As can be seen the result is encapsulated by:
`type Result a = Either Error a`
which also allows the return of specific errors (described by data type `Error`) instead of the result. Return type
`type DBAction a = Connection -> IO (Result a)`
is assigned to all operations at the highest level to enable a monadic combination of several requests. `Connection` provides access to the database via a `Handle`.
To enforce a cross join of two or more tables their descriptions have to be combined:

```
  data CombinedDescription a =
               CD [(Table, Int, [SQLType])]
                  ([SQLValue] -> a)
                  (a -> [[SQLValue]])
```

The structure and functionality is similar to the one of `EntityDescription`. As an example we join the tables Student and Result using the corresponding constructor function:

```
combineDescriptions :: EntityDescription a ->
                       Int ->
                       EntityDescription b ->
                       Int ->
                       (a -> b -> c) ->
                       (c -> (a, b)) ->
                       CombinedDescription c

studentResultCD =
  combineDescriptions studentDescription 0
                      resultDescription 0
                      (\e1 e2 -> (e1,e2))
                      id
```

Other kinds of joins are not yet supported.

We end this section with a closer look at the condition. It is mainly described by the type

```
   data Criteria = Criteria Constraint [Option]
```

where the list of options represents the `Order-by` clause and `Constraint` is a data type representing all operations for conditions discussed in chapter 3. An exemplary constraint was already described in the last section.

4.4 AbstractCurry

`AbstractCurry` is a library for meta-programming in Curry. It provides functions and data types to define Curry expressions dynamically by another Curry program. The use of the library functions prevents simple errors like a forgotten parenthesis which easily occur when building up a program by the concatenation of strings. A pretty printing function for each part of an abstract Curry program is also provided. This is especially important since Curry applies the same layout rule as known from Haskell, so that the correct indentation is crucial. The pretty printing is based on a linear-time algorithm proposed in [4] so that it is also faster than pure string concatenation.
As compilers, by definition, have high requirements on the syntactical correctness of the output program, the `AbstractCurry` libraries were applied for the code generation. The `ERD2CDBI`-module presented in the last section was also rewritten now using these libraries.

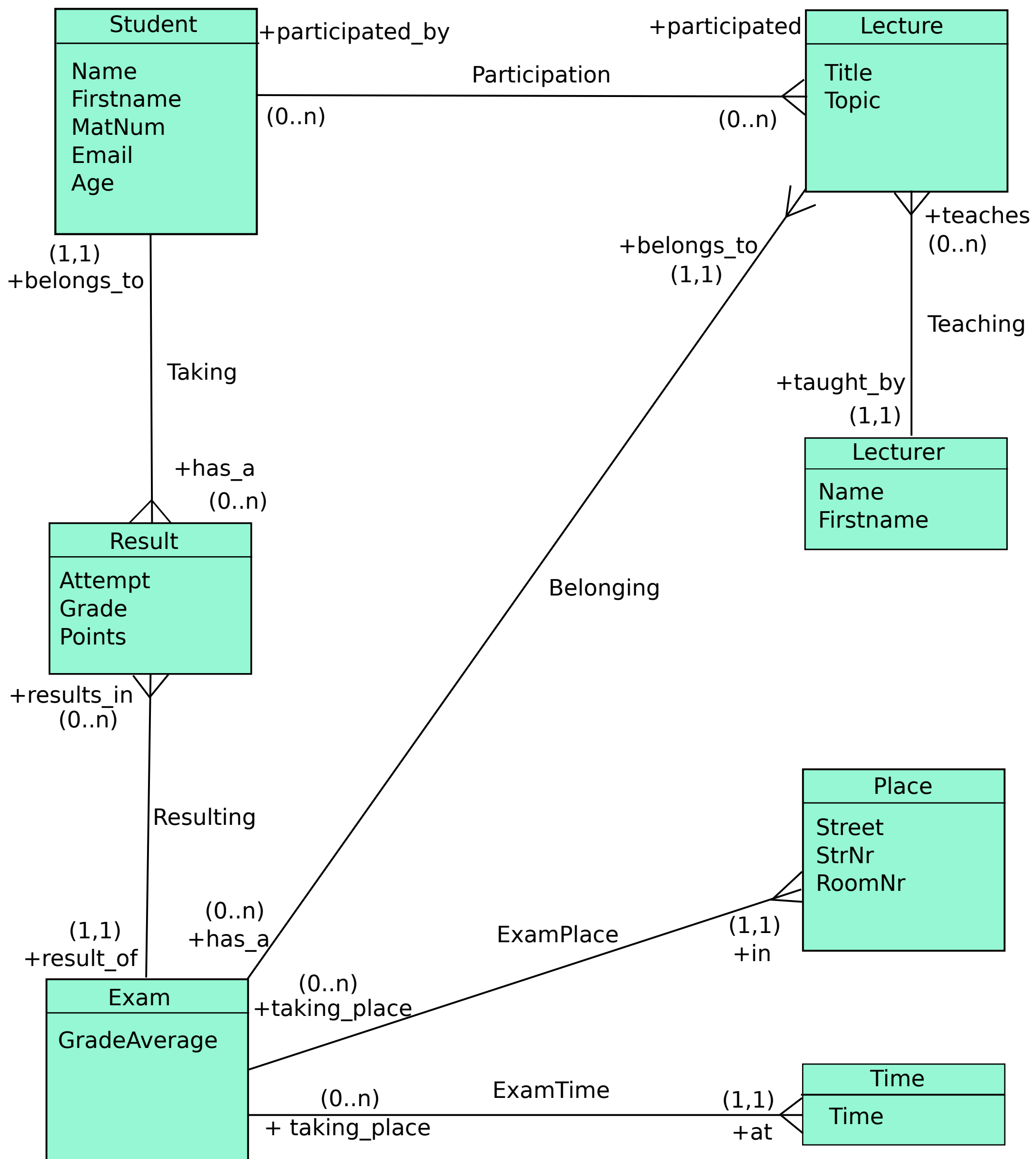

Figure 4.2: entity-relationship-diagram of a university

Chapter 5

Compiler

Generally said, compilers, also called translators, are programs that translate programs written in the one programming language into another language.
This chapter outlines the structure, the essential parts and data structures of a compiler and takes a closer look at the theory behind some of the properties.
The translation can be divided into the following phases:

1. lexical analysis
2. syntactic analysis
3. semantic analysis
4. intermediate code generation
5. optimization
6. code generation

In case of the compiler developed in this thesis stages 4 and 5 were not necessary and therefore omitted.
Stages 1-4 are known as the front end of the compiler and are independent of the target language, stage 5 and 6 form the back end.
A basic knowledge about different types of grammars and their annotation is assumed in the following sections and thus, just the most important terms are recalled.
All definitions and explanation in this chapter are based on [7] and [14].

5.1 Scanner

The lexical analysis is performed by the part of the compiler named scanner. Its input is a list of characters (or more general a string), which is processed character by character and finally transformed into a list of tokens. Tokens are data terms representing the main components of a programming language as identifiers, operators, keywords and constants.
Scanners work on regular grammars, i.e., type 3 of the Chomsky hierarchy, so that their rules can be described by a deterministic finite automaton (DFA). Regular grammars are easy to understand and implement but quite limited in their expressive power, so that a scanner is not able to recognize the syntax of a program. Thus, normally no errors are thrown during lexical analysis, but lexical errors are often tried to recover.

5.2 Parser

The syntactic analysis is performed by a parser, which is based on a context-free-grammar.

5.1 Definition (Context-free grammar)
A context-free grammar G is given by $G = (N, T, S, P)$ where

- T is a set of terminal symbols
- N is a set of non-terminals
- $S \in N$ is the start symbol
- P is a set of production rules, that follow the pattern: $A \rightarrow \beta_1 ... \beta_n$, where $A \in N, \beta_i \in (N \cup T), n \geq 0$ and $\beta_1 ... \beta_n$ is called a sentential form.

Taking the list of token as input, an abstract syntax tree (AST) is generated as output.
Two approaches in parser design are distinguished: bottom-up- and top-down-parser. The latter one is applied for the developed compiler, so it will be described in more detail.

Abstract Syntax Tree

An abstract syntax tree is a data structure generated during the parsing stage and used in all subsequent stages of the compilation process. Each stage is

not only allowed to read, but also to modify and extend the AST. Thus, the syntax tree can also present semantic information of the program.
Furthermore, as the name indicates, the syntax is presented in an abstract way, i.e., information that was important for the syntactic analysis, but is not for the semantic analysis, can be omitted.

Recursive Descent LL(1) Parser

Recursive descent parsers are the basic form of the top-down-approach, indicating the process of recursively adding child nodes (descendants) to the current node of the AST by applying grammatical rules. The term LL(1) denotes left-to-right parsing using left most derivation and 1-symbol look ahead. To enable LL(1) parsing, i.e., to ensure that 1-symbol look ahead is sufficient to choose the correct rule, the grammar has to satisfy some requirements. To formulate them we need the following definitions:

5.2 Definition (Start)
For $w \in T^*$ and a context-free grammar $G = (N, T, S, P)$

$$start(w) = \begin{cases} w & \text{if } \mid w \mid < 1 \\ u & \text{if } w = uv \text{ and } \mid u \mid = 1 \end{cases}$$

is called start of w and indicates the first terminal symbol of the word w or w itself, in case it is the empty word ϵ.

5.3 Definition (First set)
Let $G = (N, T, S, P)$ be a context-free grammar and $\alpha \in (N \cup T)^*$ a sentence. Then the first set of α, i.e., all initial terminal symbols of sentential forms derived from α, are given by:

$$First(\alpha) = \{start(w) \mid \alpha \longrightarrow^* w \in T^*\}$$

5.4 Definition (Follow set)
Let $G = (N, T, S, P)$ be a context-free grammar and $A \in N$. Then the Follow set of A, i.e., the set of terminals that can follow A in a sentential form, is defined by:

$$Follow(A) = \{w \in T \mid S \longrightarrow^* uAv \text{ and } w \in First(v)\}$$

Finally a constructive definition for the LL(1)-property can be given.

5.5 Definition (LL(1))
Let $G = (N, T, S, P)$ be a context-free grammar, $A \longrightarrow \beta$, $A \longrightarrow \gamma \in P$, $\beta \neq \gamma$.
G is LL(1)

1. in case $\varepsilon \notin First(\beta)$: if $First(\beta) \cap First(\gamma) = \emptyset$
2. in case $\varepsilon \in First(\beta)$: if $Follow(A) \cap First(\gamma) = \emptyset$

This definition directly implies that a LL(1)-grammar always has to be unambiguous and without left-recursion. Such a grammar is given for the supported subset of SQL in appendix C.

Semantic Actions

Semantic actions are performed during the construction of the syntax tree. The concept is based on the idea that each node has a semantic meaning, which can be composed by processing the other nodes in the syntax tree. Accordingly two kinds of semantic attributes can be distinguished:

- synthesized attributes: nodes consume information from their child-nodes and provide it to their parent-node
- inherited attributes: nodes consume information from their parent-node and their left-siblings, and provide it to their child-nodes and right-sibling

Error management

Error management is very important to develop user-friendly compilers. There are two kinds of strategies: error repair and error recovery. While the former tries to modify the already parsed or the still unparsed input the latter one simply sets the parser to a later position of the input to continue the parsing process. Neither of these alternatives is immune to produce even more errors by correcting or skipping one. With both methods detailed error messages have to be returned.
LL(1)-parsers are very suitable for a good error management cause the token that can follow or replace an erroneous one is necessarily given by the grammar. It is a good practice to limit the error management during syntactic analysis to syntactic errors and leave the semantic issues to later stages. This is for two reasons. The first one is to maintain the modular structure of the compiler. The second one is to save resources, i.e., the next stage of the compiler is just initiated in case the former one was free of errors. In detail: after an erroneous syntactic analysis the compilation is aborted. Thus, no semantic errors caused by an already reported syntactic error are thrown.

5.3 Semantic Analysis

This stage is working on the AST constructed by the parser. What exactly is done during the semantic analysis depends on the source language and the requirements. Typical tasks are type checking and verification of declarations.

Symbol table

A symbol table is another data structure that can be read and modified during various stages of the front end except lexical analysis. It is presented here because its most extensive use will be made during the semantic analysis. The symbol table saves information which cannot be efficiently provided by the AST.
Symbol tables bind names to any type of attributes that are important for the compilation process, e.g., data types or in case of SQL also pseudonyms. Those names and attributes are normally referenced and declared in different parts of the program, which is why an AST is not suitable for this type of information.
A symbol table is usually implemented by a data structure that provides efficient access to key-value-pairs. Furthermore, it has to be able to model a scope concept w.r.t. the visibility of names applied by most programming languages.

5.4 Code generation

After a successful semantic analysis the AST is assumed to contain any information needed for the translation. The complexity and the requirements for this stage surely heavily depend on the target language. The organisation of memory at runtime and the allocation of registers are just some examples of tasks that can be performed at this stage. The final output is the program in the target language.

III

Implementation

Chapter 6

Concept and Specification

The fundamental idea of this thesis is to provide a possibility to integrate SQL statements directly into Curry programs as shown in the following two examples of a `Select` statement and an `Update` statement with embedded Curry expression:

```
select :: SQLResult [(String, (Maybe Float))])
select =
  do
    result <- ``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;''
    return result

update :: String -> Int -> IO (SQLResult ())
update mail age =
  ``sql Update Student
        Set Email = {mail}, Age = {age}
        Where (Name like "M%" And Firstname like "A%");''
```

The itemization below lists briefly the most important requirements for this integration. Details are explained in the remainder of this chapter.

- **No loss of type safety:** An SQL statement does not provide any type information and all data is stored as string values at the database level. However, when integrating SQL into Curry, each value has to be represented by the correct type inside the Curry program. In addition, it should not be allowed to compare different types, e.g., an integer value to a string column, in conditions. Furthermore, insertion and update of entities and columns have to be restricted to the correct argument types.
- **Prevent later compilation or runtime errors:** SQL statements always refer to a certain data model. It has to be assured that tables and

columns referenced in the statement are defined in the model. Furthermore, the syntax of the statement has to be checked before execution to detect possible errors at an early stage. Special effort has to be made regarding the usage of null-values and key columns.

- **Support the functional logic programming style:** Specialized functions provided by the CDBI library can also be used as an SQL statement, e.g., an update function for complete entities, which is explained below.
- **Using a standard SQL notation for a high grade of usability:** The query language can be used in its pure format. There is no need to study the use of library functions or any additional syntax.
- **Offer a high abstraction level regarding foreign keys:** This requirement needs a more profound explanation which is provided further below.

As seen in chapter 1 the preprocessing approach is a suitable method to achieve all of these goals. The integrated SQL statement has to be translated into typesafe (CDBI) library functions before the compilation of the complete Curry program. During the translation process, types and consistency with the data model have to be checked as far as possible so that the majority of errors is already found during preprocessing the file and not just during the compilation of the generated code. This reduces the amount of time and effort spent for failed compilation attempts. Meaningful error messages are to be generated. An error recovery management has to be applied to reach a high degree of usability. Warnings are to be emitted in any situation that can easily lead to an error, for instance, differing notation of data model elements or missing type information.
The integration is to be based on ER-models. Thus, the data model has to be processed beforehand in order to provide all information needed for the mentioned analysis concerning types and consistency to the preprocessor.

The CDBI library provides some functions which are motivated by the functional logic programming style, e.g., an update function which allows to pass a complete entity (i.e., an instance of the data type) instead of assignments. The corresponding row is then fetched by its key, all changed values are updated and the entity is inserted again. It is desirable that the applied SQL dialect understood by the preprocessor supports such functionality as well. An example is shown below:

```
updEntity :: Student -> IO (SQLResult ())
updEntity student = ``sql Update Student Set {student};''
```

Furthermore, the integration of expressions of Curry code into the SQL statement ought to be possible at least for constant values and complete entities, as can be seen above. This allows the use of parameters and enables a more dynamic programming. Even more dynamic queries that permit, e.g., to pass

a complete condition clause as a string parameter, are not possible due to the preprocessor's architecture. Passing complete condition clauses as Curry expressions will not be implemented in this version of the preprocessor. It would be possible to support this feature in later versions, but we will also see in later chapters that the use of embedded expressions hampers the realization of type errors during preprocessing.

To avoid additional learning effort and confusion due to uncommon notation, a very common format of SQL is to be implemented. The applied dialect is mainly influenced by SQLite, which presents a quite relaxed handling of notation issues. However, for the implemented compiler an intermediate interpretation has to be chosen with safety and usability in mind. In detail: Keywords have to be completely case-insensitive. Names and pseudonyms have to be case-insensitive, but a warning is to be generated in case the notation differs from the original notation given in the referenced data model. In addition, the use of pseudonyms should be optional as long as column names are unambiguous.
For the sake of usability, redundant writing effort has to be prevented. Therefore the applied dialect ought to support abbreviatory notations for the `insert` statement, i.e., leaving out null-values and also key values which are auto incrementing.
Furthermore, the following transactional statements are to be provided: `Begin`, `Commit`, `Rollback` and `InTransaction`, a counterpart to the CDBI-function `runInTransaction`, which automatically invokes a `Rollback` in case of error and a `Commit` otherwise.
A concatenation of several statements should also be possible. In this case the result of the last one is to be returned.

A supplemental facilitation will be provided by an abstraction of foreign keys which allows to completely avoid the reference of the corresponding columns without influencing the expressive power. Instead of formulating complicated constraints using automatically generated foreign key columns, it is then possible to use relationship names that were defined in the ER-model by the user himself.
A review of the entity-relationship-diagram introduced in figure 4.2 clarifies why this is especially desirable when working with Curry and the tool `erd2curry`. Figure 6.1 shows the n-to-1 relationship between `Student` and `Result`. As we have already seen earlier, the foreign key columns are automatically generated and inserted by the `erd2curry`-tool, always following the same naming pattern. Thus, in this case a foreign key column called `StudentTakingKey` would be inserted into the `Result` entity. To connect both entities in a query written in SQL (e.g., to formulate a join condition), it is necessary to reference this generated column, forcing the user to gain a certain knowledge about the internal procedures of the applied tool. The abstraction proposed by this thesis aims at making use of the elements the user already has defined in his data model instead of forcing him to use additional ones. The

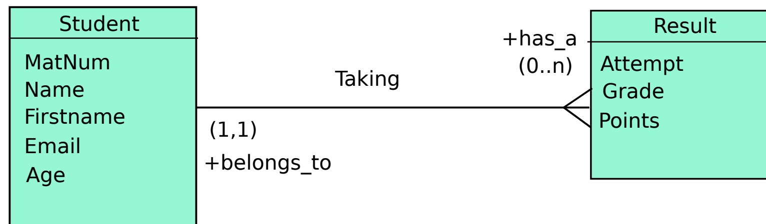

Figure 6.1: Relationship between Student and Result

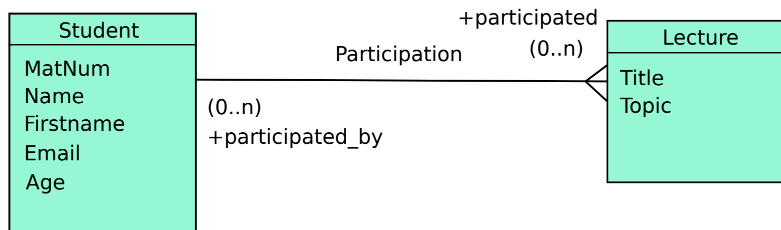

Figure 6.2: m-to-n-relationship between `Student` and `Lecture`

presented approach exploits the given relationship names, in this case `has_a` or `belongs_to`, and introduces the new keyword `Satisfies` to formulate corresponding conditions. Instead of the former query using foreign keys:

```
Select * From Student As s
         Inner Join Result As r
                   On s.Key = r.StudentTakingKey;
```

the user is now free to write one of the following:

```
Select * From Student As s
         Inner Join Result As r
                   On Satisfies s has_a r;

Select * From Student As s
         Inner Join Result As r
                   On Satisfies r belongs_to s;
```

which is also more intuitive. Note that the constraint requires the unidirectional relationship name and that the order of first entity, relationship and second entity has to be correct and cannot be altered.
The abstraction is an even greater relief regarding m-to-n relationships, whose translation results in automatically generated auxiliary entities. Figure 6.2 shows the relationship `Participation` between `Student` and `Lecture`. As seen before it is lifted to an additional entity containing two foreign keys which subsequently have to be used to connect `Student` and `Lecture` in SQL queries:

```
Select s.*
```

```
From Student As s, Lecture As l
Where Exists
       (Select * From Participation As p
         Where s.Key = p.StudentParticipationKey
          And  l.Key = p.LectureParticipationKey);
```

The proposed abstraction allows to replace the `Exists` constraint using one of the three connecting relationship names since all of them can be seen as bidirectional:

```
...Where Satisfies s participated l;
...Where Satisfies l participated_by s;
...Where Satisfies s Participation l;
...Where Satisfies l Participation s;
```

The introduced `Satisfies` constraint can be placed wherever foreign keys can be used, i.e., beside join conditions and `Where` clauses as shown above, in `Having` clauses and `Case` expressions. Of course usual foreign key expressions also have to be supported and understood by the preprocessor.

To meet the requirements of type safety and a common SQL dialect, the CDBI library has to be extended in several points, too. The next chapter explains the implementation details of these enhancements. Chapter 8 describes the implementation of the compiler. Its integration into the preprocessor `currypp` is outlined in chapter 9.

Chapter 7

CDBI-Enhancements

To provide the scope of a common SQL dialect in a type safe way as outlined in the last chapter, it is necessary to extend the CDBI libraries. The required enhancements can be divided into four main parts:

- ensure type safety regarding keys and foreign keys in conditions
- define the auto increment property for primary keys
- provide the possibility to formulate more complex `Select` statements, in particular, to select single columns
- improve usability in connection with the preprocessor

A main requirement for all enhancements is to maintain the type safety and to keep the traffic low, i.e., to load as little data as possible into the program (in fact just the data that was requested). In addition, the interface to the user is to be enhanced, but not completely changed, so that all extensions can be used intuitively in case the user is already familiar with the first version of CDBI.
This chapter describes the implementation of the above points in more detail. Keep in mind that all enhancements are made with regard to their use in combination with the later presented compiler.

7.1 Type safety regarding keys

Keys and foreign keys, although described by their own ID-types inside an entity definition, are represented as integer columns when used in conditions. That allows to compare a (foreign) key column with any integer column:

```
conn <- connectSQLite "Uni.db"
result <- getEntries resultDescription
          (Criteria
           ((col resultColumnStudentTakingKey)
                .=. (col resultColumnPoints)))
          conn
```

This is obviously an absurd request but therefore demonstrates the possible abuse of foreign keys quite well. However, it can be easily avoided by assigning an appropriate and unique type to each key.
The infix operator `.=.` is defined as constructor function for the `equal`-operation to ensure that just values and columns of the same type are compared (as well as the other infix operators). Hence, to avoid that (foreign) key columns can be compared to any integer column, it is necessary to declare (foreign) key columns with their own ID-type as it is already done in the entity declaration. Accordingly the type of column `StudentTakingKey` in table `Result` has to be changed as follows:

```
-- former declaration
resultColumnStudentTakingKey :: Column Int
-- changed declaration
resultColumnStudentTakingKey :: Column (StudentID)
data StudentID = StudentID Int
```

Now the above request results in a type error, because an integer column cannot be compared to a column of type `StudentID`. However, with this restriction we also hamper the comparison of key columns and constant values. Thus, for each ID-type a corresponding function has to be defined dynamically which makes use of the newly defined function `idVal`:

```
idVal :: Int -> Value _
idVal i = Val (SQLInt i)

studentID :: StudentID -> Value StudentID
studentID (StudentID key) = idVal key
```

Using these functions constraints like the following are possible:

```
(between (col studentColumnKey)
         (studentID (StudentID 1))
         (studentID (StudentID 4)))
```

which corresponds to the SQL constraint

```
Student.Key Between 1 And 4
```

As can be seen, the `idVal`-function can easily be abused to declare an integer value as any type that can be expressed by an `SQLValue`. To avoid such a usage the function must not be exported by the module interface finally provided to the user, but only to the generated module containing the data declarations representing the data model. This module finally exports the generated constructor functions as e.g., `studentID`.
All the changes have to be implemented in the `ERD2CDBI`-module as this is

where the type declarations of the data model are generated. Only the `idVal`-function is defined in the `Criteria`-module.

7.2 Auto incrementing keys

As a further requirement the manual modification of keys is to be minimized. The current version of the libraries allows the insertion of arbitrary key values, which could provoke errors at runtime, e.g., in case a key value is not unique. At the library level this problem can be solved by defining all primary keys as auto incrementing and modifying the insertion function in order to support this feature.
As seen in chapter 4, the primary key of each entity was added during the transformation process by the `erd2curry`-tool. Thus, its modification by the user should also be inhibited as far as possible. The use of the auto increment property avoids the insertion of arbitrary key values, instead the currently highest value is fetched and incremented by one.
Using `sqlite`, this property has to be invoked by declaring each key as *integer primary key* during the database creation process and set the key value to null when inserting a new entity.
While the first part is achieved by another minor change to the `ERD2CDBI`-module, the second part demands a modification of the conversion function in each `EntityDescription`. The following shows the corresponding function for the entity `Lecturer`:

```
--inserting given key values
(\(Lecturer (LecturerID key) name firstname) ->
      [SQLInt key, SQLString name, SQLString firstname])
--using auto incremented keys:
(\(Lecturer _ name firstname) ->
       [SQLNull , SQLString name, SQLString firstname])
```

The values given by the entity are prepared for insertion by converting them to an instance of type `SQLValue`. The parameter for the key column is now ignored and replaced by `SQLNull`.
A problem arises from a particular update function provided by CDBI. When passing an entity to this function, the corresponding database row is fetched (identified by its key), changed and inserted again - using the same key. To keep this functionality, each `EntityDescription` has to provide both conversion functions defined above. Thus, the type declaration of `EntityDescription` has to be changed as follows:

```
data EntityDescription a = ED String
                              [SQLType]
                              (a -> [SQLValue])
                              --for insertion
                              (a -> [SQLValue])
```

```
                                    ([SQLValue] -> a)
```

The same functionality is available for CombinedDescriptions. Thus, the corresponding add-on was made here as well:

```
data CombinedDescription a = CD [(Table, Int, [SQLType])]
                                ([SQLValue] -> a)
                                (a -> [[SQLValue]])
                                --for insertion
                                (a -> [[SQLValue]])
```

Accordingly, the constructor function combineDescriptions was adjusted to match the changed definition.

7.3 Complex Select statements

So far the library just allows the selection of complete table rows specified by a condition and an Order-by clause. To provide more complex Select statements, the library is extended with data types and functions to express:

- the selection of single columns, i.e., the project operation as specified in the relational model
- a Group-by clause and a Having clause
- aggregation functions Sum, Avg, Count, Max, Min
- set operations Union, Intersect, Except
- an Inner Join operation
- the specifiers Distinct and All
- a Limit clause

This section first explains the implementation of the project operation. Further below, the remaining operations are introduced, arising problems with the existing data types are demonstrated and solved. Finally, we define new functions for the selection of different numbers of columns from an arbitrary number of tables which provide the possibility to use all operations described above.

To implement the selection of single columns the data type ColumnDescription was introduced:

```
data ColumnDescription a = ColDesc String
                                   SQLType
                                   (a -> SQLValue)
                                   (SQLValue -> a)
```

It is in structure and functionality equivalent to `EntityDescription` providing the complete column name, the expected type and two conversion functions for insertion and selection respectively. The definition of the `Description`-type for each individual column is also generated by the `ERD2CDBI`-module. An example is given below for the key column of table `Lecturer`:

```
lecturerKeyColDesc :: ColumnDescription LecturerID
lecturerKeyColDesc =
  ColDesc "\"Lecturer\".\"Key\""
          SQLTypeInt
          (\(LecturerID key) -> (SQLInt key))
          (\(SQLInt key) -> (LecturerID key))
```

Besides a column name, SQL allows a case expression and the use of aggregation functions in the `Select` clause, i.e., the part of the statement following the `Select` keyword. To express all three options in Curry, we define the following data type:

```
data ColumnSingleCollection a =
   ResultColumnDescription (ColumnDescription a)
                           Int
                           String
 | Case Condition (CValue, CValue) (CaseVal a)
```

It can represent either a column given by its `ColumnDescription`, an alias as integer value and an aggregation function, which can also be the identity, or a `Case` expression. The latter is composed of a condition, a tuple of values (for the then-branch and for the else-branch respectively) and an instance of the data type `CaseVal`, which consists of the `SQLType` and the corresponding conversion function. Note that both branches in the `Case` expression have to be of the same type, which is ensured by the exported constructor function:

```
caseThen :: Condition ->
            Value a ->
            Value a ->
            (CaseVal a) ->
            ColumnSingleCollection a
```

The data type `CaseVal` was introduced to ensure that an `SQLType` is combined with the correct conversion function and is defined as follows:

```
type CaseVal a = (SQLType , (SQLValue -> a))
```

The definition itself is not exported but a constructor for each valid combination, e.g., for type string:

```
caseResultString :: CaseVal String
caseResultString = (SQLTypeString, getStringValue)

getStringValue :: SQLValue -> String
getStringValue (SQLString str) = str
```

Now we can express the SQL projection:

```
Case When Student.Age < 20 Then "Young" Else "Old" End
```

in Curry:

```
caseThen (condition (lessThan (colNum studentColumnAge 0)
                               (int 20)))
         (string "Young")
         (string "Old")
         caseResultString))
```

The constructor function for the `ColumnSingleCollection` has to ensure the correct result type when using an aggregation function. For instance, applying the function `Count` on a column of arbitrary type, the result type always has to be an integer value. Therefore the constructor function `count` returns a tuple of a string (representing name and specifier) and a `ColumnDescription` for a pseudo integer column:

```
singleCol :: ColumnDescription a ->
             Int ->
             (ColumnDescription a -> Fun b) ->
             ColumnSingleCollection b

type Fun a =  (String , ColumnDescription a)

count :: Specifier -> ColumnDescription _ -> Fun Int
```

Based on the `ColumnSingleCollection`, we define data types that represent up to five columns, which are at least tuples, triple etc. of `ColumnSingleCollection` all accessible via corresponding constructor functions and called accordingly `ColumnTupleCollection` etc. Thus, the projection on two columns with aggregation:

```
...Student.Name, Count(Result.Points)...
```

can be expressed in Curry as follows:

```
tupleCol (singleCol studentNameColDesc 0 none)
         (singleCol resultPointsColDesc 0 (count All))
```

The next step is to provide data types for the remaining operations. For specifiers, set operations and joins this is straightforward:

```
data Specifier = Distinct | All

data SetOp = Union | Intersect | Except

data  Join = Cross
           | Inner Constraint
```

However, an enhancement by set operators leads to a problem with the former definition of the `Criteria` data type:

```
data Criteria = Criteria Constraint [Option]
```

As the condition clause is explicitly connected with the `Order-by` clause, the definition contradicts the overall structure of a `Select` statement shown in listing 7.1.

```
selectStatement ::= selectHead { setOperator selectHead }
                                [ orderByClause ]
                                [ limitClause ]
selectHead ::= selectClause fromClause [ WHERE condition ]
                          [ groupByClause [ havingClause ]]
```

Listing 7.1: structure of `Select` statement in EBNF

Thus, it was changed to:

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

The list of options representing the `Order-by` clause was completely removed, because it has to be ensured that an `Order-by` clause as well as the `Limit` term can only appear once in the whole statement, while a `Group-by` clause is allowed to appear in each single `selectHead`. I.e., we want to allow the following query:

```
(Select s.Name From Student As s
 Union
 Select l.Name From Lecturer As l)
Order By s.Name Limit 5;
```

However, queries as the one below have to be excluded:

```
(Select s.Name From Student As s
 Order By s.Name Limit 5)
Union
(Select l.Name From Lecturer As l
 Order By l.name Limit 2);
```

This query is syntactically incorrect but was allowed by the former definition of the `Criteria` datatype.
To change this, the `Order-by` clause and the `Limit` clause were moved to every single selection function provided by the main module. For instance, the signature of the function `getColumn`, selecting one single column, is given by:

```
getColumn :: [SetOp] -> --set operation
             [SingleColumnSelect a] -> -- contains
                                       -- Group-by
             [Option] ->  -- order-by-clause
             Maybe Int -> -- limit-clause
             DBAction [a]
```

The type `SingleColumnSelect` corresponds to the `selectHead` in listing 7.1 and thus, contains amongst others condition, `Group-by` and `Having` clause. This is explained in more detail further below. As a list of `SingleColumnSelects` is allowed, these parts of the statement can appear several times. The list of `Option`, representing the `Order-by` clause, and the integer value, representing the `Limit` clause, can just appear once.
This structure allows to express only the first, correct statement given above:

```
getColumn [Union]
          [SingleCS All
                    (singleCol studentNameColDesc 0 none)
                    (TC studentTable 0 Nothing)
                    (Criteria None Nothing) ,
           SingleCS All
                   (singleCol lecturerNameColDesc 0 none)
                   (TC lecturerTable 0 Nothing)
                   (Criteria None Nothing)]
          [ascOrder (lecturerColumnFirstname 0)]
          (Just 3))
```

Before explaining the data types to represent a `selectHead` for different numbers of columns, we take a closer look at the definition of a `Group-by` clause:

```
data GroupBy = GroupBy CValue GroupByTail

data GroupByTail = Having Condition
                 | GBT CValue GroupByTail
                 | NoHave

data Condition = Con Constraint
               | Fun String Specifier Constraint
               | HAnd [Condition]
               | HOr [Condition]
               | Neg Condition
```

The data type `GroupBy` contains the first column the result is grouped by. The type `GroupByTail` recursively defines further columns, a `Having` condition or just no tail at all. To support aggregation functions in the `Having` clause a separate condition type is defined, which internally uses the `Constraint` type also applied in the `Where` clause.
The aggregation functions inside the `Having` clause have to be implemented using different data types than those we have already seen in the last section. While there is no need for type conversion, it still has to be ensured that both parts of the condition, which can be column, aggregation result or constant value have suitable types. The functions `Sum` and `Avg` can just be applied on numerical column types and `Avg` always has to be compared to a floating point number, while `Count` always generates an integer value. This is achieved by constructor functions like the following one for the average of integer columns:

```
avgIntCol :: Specifier ->
             Value Int ->
             Value Float ->
             (Value () -> Value () -> Constraint) ->
             Condition
avgIntCol spec c v op =
     (Fun "Avg" spec (op (toCValue c) (toCValue v)))
```

Parameter `op` can be any binary operator defined in the `CDBI.Criteria`-module. This way, already implemented functionality can be reused and the same level of type safety is guaranteed. Similar functions for the rest of supported aggregations are shown in appendix B. Using the implemented functions, it is now possible to express the following example in Curry:

```
...Group By Student.Name
   Having Avg(Result.Points) > 80.0 ...

groupBy (colNum studentColumnName 0)
        (having (condition
                   (avgIntCol All
                              (colNum resultColumnPoints 0)
                              (float 80.0)
                              greaterThan)))
```

The last step necessary to provide complex `Select` statements in a uniform and manageable way is a data structure to describe the `selectHead`. The type `SingleColumnSelect` used above is defined as follows:

```
data SingleColumnSelect a  =
              SingleCS Specifier
                       (ColumnSingleCollection a)
                       TableClause
                       Criteria

data TableClause = TC Table Int (Maybe (Join,TableClause))
```

As the `selectHead` demands, it includes specifier, columns or `Case` expressions, tables and joins and `Where` clause inclusive `Group-by` clause and `Having` clause. For the selection of two, three, four and five columns corresponding types are provided.
The data type `TableClause` was defined to ensure that at least one table is specified and that the appropriate number of joins is given in case of two or more tables. Its structure is kept close to the SQL grammar (see appendix C). Finally, selection functions similar to the `getColumn`-function shown above are defined for each supported number of columns.

All data types and functions for the description of complex `Select` statements are defined in the new `CDBI.QueryTypes`-module. Only the new selection functions are part of the main module and the `ColumnDescription` type is defined in `CDBI.Description`. An overview of all additionally exported functions and types can be found in appendix B. More complex examples, which make use of the presented extensions are given in appendix D.

Despite of all the extensions made to the interface there are still some limitations for the formulation of `Select` queries:

- the selection is limited to five columns from an arbitrary number of tables

- the combination of ∗ and set operators is not supported
- the subquery inside the `Exists` constraint is limited to the selection of all columns (∗) of exactly one table without `Order-by` clause, `Limit` clause and set operators.
- `Case` expressions do not allow key values in neither of their branches

7.4 The executable `erd2cdbi`

The first version of the module `ERD2CDBI` generated data types needed for the use of the library by parsing a given `ERD` term. Furthermore, a new, empty database was set up, every time the main function of the module was applied. The following extensions and changes were made to this module:

- a separate information file for the parser is generated containing properties of the data model
- the creation of the database is optional to enable working with existent databases

Furthermore, `erd2cdbi` became a stand-alone executable and is now part of the Curry distributions.

The information about the data model that is needed during the parsing process has to be in a different format than the one for the library. The data structure applied for this task is not part of the tool itself and is therefore described in the next chapter. Nevertheless, it has been decided to include the generation of the parser information into the `erd2cdbi`-tool to avoid an additional step for the user. This approach also prevents consistency problems because the use of the same `ERD` term in both transformations is ensured.

The extensions require some additional parameters for the invocation of the tool. Beside the path to the .term-file which contains the `ERD` term, it is necessary to pass the absolute path to the database. As the creation of the database is now optional, the option '-db' has to be used to invoke it. Of course there is a risk of using a database that doesn‘t correspond to the applied data model and thus, leads to runtime errors. However, the possibility of working on existent databases, e.g., shared ones or databases that were created with the former dynamic-predicate-approach, weights much more.
To give an example, the call

```
erd2cdbi "Uni_ERDT.term" "~/Uni.db" -db
```

produces three output files:

- the file Uni_CDBI.curry (formerly CDBIUni.curry) contains the data types for the libraries with all changes discussed above
- the file Uni_SQLCode.info contains the information about the data model readable for the compiler
- an empty database Uni.db created according to the given `ERD` term (nothing was changed here) in the home directory

More details about the installation and usage of `erd2cdbi` can be found in appendix A.
For the generation of the parser information file it is important to mention that the part of the ERD-term specifying the relationships is no longer skipped, but parsed to provide information about the type of each relationship. The concrete structure used for the info file is described in chapter 8.2.

7.5 Modifications to `CDBI.Connection`

The following two small modifications/extensions were just made to facilitate the use of the library especially in connection with the preprocessor.
First, `Result`, the result type of requests defined in the `CDBI.Connection`-module was renamed to `SQLResult`:
`type SQLResult a = Either DBError a`
to prevent a collision with a potential type called `Result`, which is frequently used in different situation as e.g., in our exemplary data model. For the same reason `Error` was renamed to `DBError`.
As an extension the function `runWithDB` was defined. Taking the path to the database and the query as `DBAction`, it returns the result as usual as `IO (SQLResult a)`, making use of the always same structure required to pass a query to the database:

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

This function will be used for the translation of all types of embedded SQL statements. Thus, it provides a unique interface and guarantees the always same return type `IO (SQLResult a)` for an embedded SQL statement.

Chapter 8

SQL-Curry-Translation

This chapter discusses the implementation of the SQL-Curry-compiler. The first section roughly outlines the structure of the project. The second section explains the implementation details of each compiler module.

8.1 Structure

For the preprocessing of SQL statements the tool `currypp`, presented in chapter 4, has to be extended by a compiler for SQL. To meet all requirements w.r.t. type safety and functional scope outlined in chapter 6, the CDBI library and the `erd2cdbi` tool were extended as described in the previous chapter. In particular, the tool was enhanced by the generation of the parser information file, which is needed during the compilation process. Figure 8.1 shows the interaction of the different parts of the project. Light blue components represent existing parts and generated files of the preprocessor `currypp`. Purple components depict the extended CDBI project. The red coloured components, the SQL compiler and the information file, were developed in this thesis and are explained in detail in this chapter.

The developed compiler for SQL statements into Curry code is designed modularly, following the structure presented in chapter 5. Below the specific function of each stage is explained briefly, outlining the complete parsing process. A graphical overview of the compiler structure is given in figure 8.2.

SQLConverter

The `SQLConverter` implements the interface for the preprocessor and thus, represents the main module of the compiler and organizes the compilation

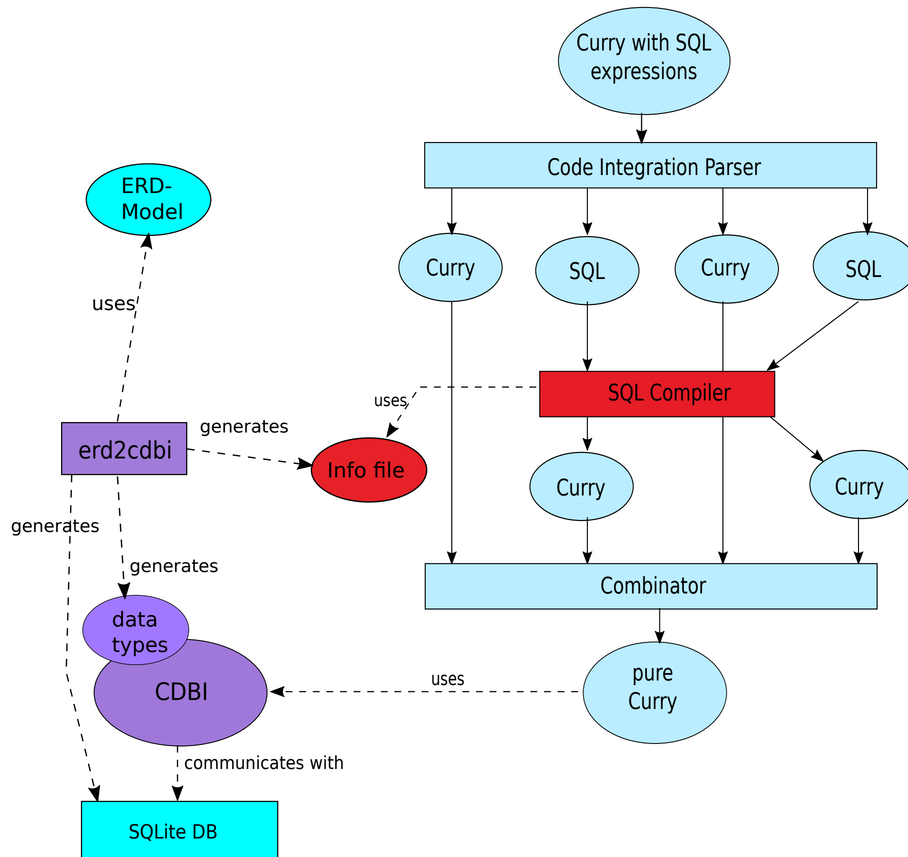

Figure 8.1: Integration of the SQL compiler into `currypp`

process. All phases are called in the correct order. In case of errors at a certain stage, the process is aborted, so that no time is wasted on a statement that is already marked as erroneous.
Furthermore, the module provides a function to read the parser information file, which contains the following information about the data model:

- the path to the corresponding database and the name of the module containing the CDBI data types for the corresponding data model
- the kind of relationship for each relation identified by its name and the names of the involved entities
- for each column name a flag whether null values are allowed or not
- for each table name its original notation and all column names
- the type for each column name

Functions to interpret this information are implemented in a separate module called `SQLParserInfoType`.

SQLScanner

This module executes the lexical analysis, i.e., it translates the string representing the SQL statement into a list of token which were especially defined for the supported subset of SQL statements and are defined in the module `SQL-Token`. No errors are thrown at this stage. Characters that are not supported, are marked as such, but still passed to the next stage.

SQLParser

The parser module converts the list of token into an abstract syntax tree - `SQLAst`, which is defined for the supported subset of SQL. This AST is a specialized structure that can contain for each node all information needed in subsequent stages particularly in the last translation stage. The following phases complete the information in each node.
The parser implements the syntactical analysis, hence, errors are only thrown in case the grammatical rules were not followed. An error recovery management is applied, so that the parsing process is not necessarily stopped in case of a broken rule. Internally the parsing procedures make use of the monadic data types defined in the `SQLParserTypes`-module.

SQLNamer

The module `SQLNamer` represents the first stage of the semantic analysis. With the help of a specialized symbol table the pseudonyms for data tables are resolved and replaced by the full table name. Exploiting this process, references for each table are counted and the numerical alias used in CDBI functions is set for each table and column reference accordingly.
Errors are thrown in case a table pseudonym cannot be resolved. This happens in case the pseudonym was defined but not used, the pseudonym was defined for more than one table, in case of typing errors or if the pseudonym was not defined or is not visible.
Warnings are generated for pseudonyms that are used in different notations within the same query.

SQLConsistency

This module performs the consistency check, i.e., all referenced table, column and relationship names have to be part of the data model. Special effort is made to prepare `Insert` statements for the later translation. As this type of statements can contain several optional parts, as e.g., a list of columns and

left-out null-values, its usage has to be checked very carefully. The mentioned optional parts are added to the statement to provide complete information and a uniform structure to subsequent stages. Details and examples can be found in the next section.
Furthermore, this module ensures that the later explained foreign key statement is just used in `Select` statements, but not in conditions for `Update` and `Delete` statements as this would result in an undefined relationship.
Moreover, it is ensured that null-values are not used in conditions instead of the predefined functions `Is Null` and `Not Null` as this is generally forbidden by the SQL standard and therefore an consistency issue.
Errors are thrown in case one of the above transformations can not be applied or one of the mentioned conditions is not complied. Warnings are generated if the notation of tables, columns or relationships differs from the notation in the data model.

SQLTyper

This module provides a first basic level of type safety. Types can be checked in all expressions with at least one column reference and for constant values. Thus, type errors are already thrown during preprocessing and not just during the compilation of the resulting file. Regarding embedded expressions of Curry code inside the SQL statement, type safety can not be guaranteed at this stage, instead the type is deduced from contextual information and a warning is generated. The comparison of two embedded expressions is not allowed because no contextual information about the type of the expressions is available.
Furthermore, it is ensured that key columns cannot be updated and that null values are not used in branches of case expressions as this would generate type errors later on.

SQLTranslator

The final stage of the preprocessing is the translation of the AST into Curry code. The only errors thrown during this phase are due to limitations of CDBI (presented in section 7.3). As maybe those limitations will be eliminated in later versions of CDBI, the design of the compiler allows to reduce the dependence on the interface to this last module.
Another limitation is indicated by the translator module: the selection of complete tables (i.e., all columns of a table) is limited for now to a maximum of three tables. A higher number of tables is supported by the CDBI library and the translator can be easily extended in case a higher number is found to be necessary.

The presented modular structure can be enhanced by further modules, which

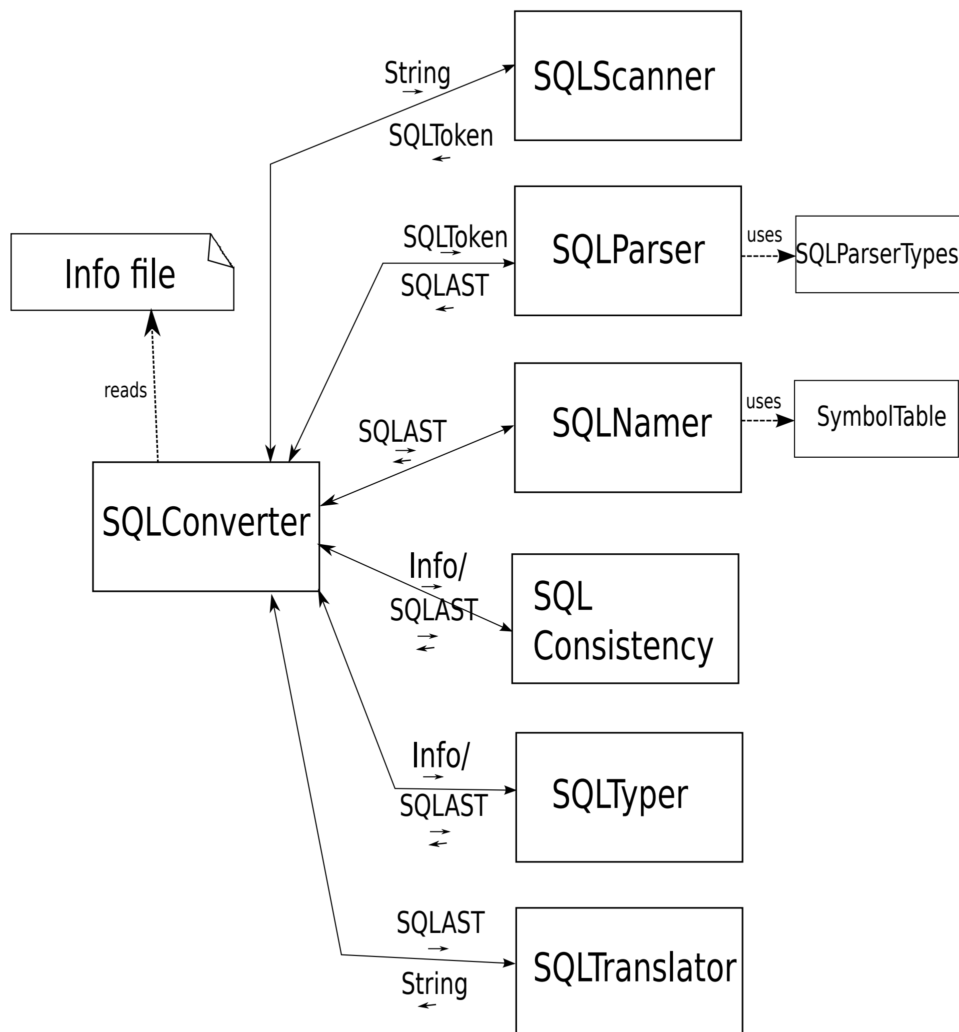

Figure 8.2: Structure of the implemented SQL compiler

can also be placed in between the existing ones. However, it is not allowed to skip one of the above modules or to rearrange their ordering, because nearly each stage is dependent on the information processed by former ones.

8.2 Implementation

This section describes the implementation of the SQL compiler devoting a subsection to each module that was introduced above. We start with an additional section to specify the terms used in the following explanations, followed by another one describing how to provide information about the data model to the compiler.

8.2.1 Notation

To avoid confusion, some terms used in the subsequent sections have to be clarified.
In the following "preprocessor" always refers to the currypp tool. "Compiler" describes the implemented project as a whole, while each single module is referred to by its name (this includes the "translator").
Parts of the Select statement are often referred to by the names used to specify the grammar that can be found in appendix C.
The term "parser information" is used for an instance of the data type ParserInfo explained in the next section, not for the file which is primarily passed to the compiler. The latter is always referenced to explicitly as a file.
Furthermore, we have to distinguish "embedded expressions" which refers to an SQL statement as a whole and "embedded Curry expressions" that describes a term of Curry inside an SQL statement.

8.2.2 Information about the data model

To implement the consistency and type check in later phases all required information about the used data model has to be fetched from the corresponding ERD term. As this implies that the information differs completely depending on which data model and therefore which database is used, a possibility to provide this information dynamically for each call of the preprocessor is required. That is achieved by generating a file named ERDName_SQLCode.info (applying the erd2cdbi-tool), which is then passed to the compiler. Thus, a specified format is needed to provide the information in a (machine-)readable way. The following data type definition is applied for this task:

```
data ParserInfo =
          PInfo (String, String)
                RelationTypes
                NullableFlags
                AttributeLists
                AttributeTypes
```

The first component is a pair of strings. The first one specifies the absolute path to the database and is later used to build up a connection. The second string is the name of the CDBI module containing the corresponding data types used by the interface. This is required by the AbstractCurry-libraries as they demand qualified names for function calls and types.
The second component is a type synonym for a list, mapping each relationship to its type. A relationship is specified by a triple containing name of the first entity, unidirectional name of the relationship and name of the second entity:

```
type RelationTypes = [((String, String, String),
                        RelationType)]
```

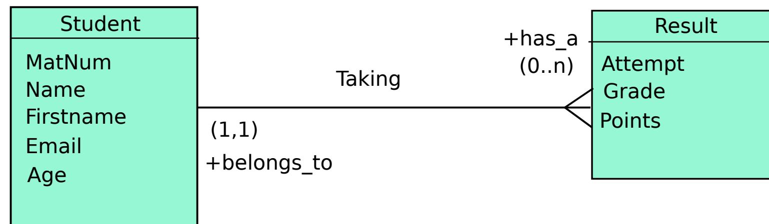

Figure 8.3: Relationship between `Student` and `Result`

The data type `RelationType` is defined as follows:

```
data RelationType =  MtoN String
                   | NtoOne String
                   | OnetoN String
```

The string parameter identifies the bidirectional name of the relation which is used for the denomination of the foreign key column by `erd2curry` and thus, also by CDBI. To clarify this, figure 8.3 shows a small part of the ER-diagram already introduced in chapter 4.2. For the shown relationship the following tuples will be included into the parser information:

```
...
(("Student", "has_a", "Result"), (OnetoN "Taking")),
(("Result", "belongs_to", "Student"), (NtoOne "Taking")),
...
```

Accordingly the column that is inserted into the table `Result` by `erd2curry` is called `studentTakingKey`. With the help of the different constructors e.g., `OnetoN` and `NtoOne` it is explicitly defined which entity was extended with the foreign key. This is an important requirement for the abstraction of foreign key constraints.
Since an m-to-n-relationship as shown in figure 8.4 has no direction by definition, four tuples are generated, which can be used interchangeably:

```
...
(("Student", "participates", "Lecture"),
                             (MtoN "Participation")),
(("Student", "participation", "Lecture"),
                             (MtoN "Participation")),
(("Lecture", "participated_by","Student"),
                             (MtoN "Participation")),
(("Lecture", "participation", "Student"),
                             (MtoN "Participation")),...
```

Examining the declaration of `RelationType` more in detail, one might notice that not exactly the same categorization of relationships is made as done by `erd2curry`. This has an easy explanation: At the preprocessing level there is no need to distinguish between foreign key columns that allow null values and those that do not, as this is already handled by the underlying functions of the

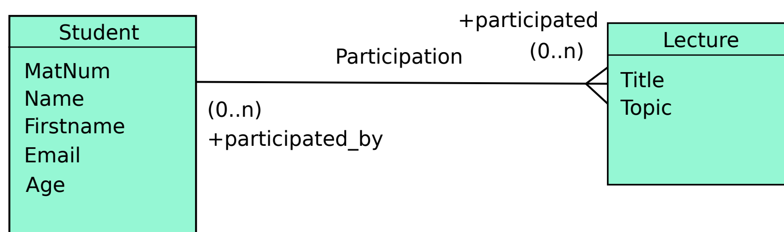

Figure 8.4: m-to-n-Relationship between `Student` and `Lecture`

CDBI library. Furthermore, as done by `erd2curry` before, cyclic relationships are also excluded by the `erd2cdbi` generation tool.

Returning to the data type `ParserInfo`, there are three more components to explain. The next one is also a list of tuples, which maps each column name to a boolean flag which declares if this column can contain null values or not.

```
type NullableFlags =  [(String, Bool)]
```

The fourth component maps each table name, completely written in lowercase, to a tuple containing the original notation of the table name and a list of the corresponding column names.

```
type AttributeLists =  [(String,(String, [String]))]
```

The final list of tuples maps each column name to its type, which is also given as a string.

```
type AttributeTypes = [(String, String)]
```

The type of (foreign) key columns is specified as the name of the referenced table, e.g.:

```
("resultStudentTakingKey", "Student")
```

For a simpler access to the information, the module `SQLParserInfoType` was defined, which beside the above data definition contains functions, that return each part of the `ParserInfo`-type (except the database name and the name of the CDBI module) as a `FiniteMap (FM)`. Since a relationship is definitely defined only by the combination of all three names (two entity names and the relationship name), a nested `FiniteMap` structure was chosen for this part of the information and a fast access is provided via an adapted lookup function. Finally the following interface to the `ParserInfo` type is implemented by the `SQLParserInfoType` module:

```
dbName       :: ParserInfo -> String
cdbiModule   :: ParserInfo -> String
getRelations :: ParserInfo -> RelationFM
getNullables :: ParserInfo -> NullableFM
getAttrList  :: ParserInfo -> AttributesFM
getTypes     :: ParserInfo -> AttrTypeFM
lookupRel    :: (String, String, String) ->
```

```
                RelationFM ->
                Maybe (RelationType, String)

type NullableFM   = FM String Bool
type AttributesFM = FM String (String, [String])
type AttrTypeFM   = FM String String
type RelationFM   = FM String
                       (FM String
                           [(String, RelationType)])
```

The `lookupRel`-function returns the original notation of the relationship and its type as a tuple.

8.2.3 SQLConverter

As the main module the `SQLConverter` implements the interface to the preprocessor, i.e., provides a function:

```
parse :: Pos -> String -> IO (PM String)
```

For the compilation of SQL statements an additional parameter for the parser information is required, so that the signature is extended to:

```
parse :: Either String ParserInfo ->
         Pos ->
         String ->
         IO (PM String)
```

Instead of the successfully read parser information an error message can also be passed e.g., in case the file could not be found.
The `SQLConverter` calls each of the subsequent modules with the corresponding part of the information, i.e., the `AttrTypeFM` is passed to the type checker, the path to the database to the translation module and the remaining components to the consistency check.
The result returned from each stage is checked for errors before the next stage is called. If the former one returned without errors, the next compilation phase is invoked, otherwise the process is aborted.
Furthermore, the module provides the read function for the parser information file:

```
readParserInfo :: String -> IO (Either String ParserInfo)
```

which for a given file name returns an instance of the `ParserInfo` type or an error message. It is called by the preprocessor before the compiler is invoked for any statement in particular to avoid that the same file is read several times.

8.2.4 Lexical Analysis

The `SQLScanner` module was implemented straight forward following the explanation in chapter 5. The exported function

```
scan :: String -> [Token]
```

reads the string parameter character by character and transforms it into a list of `Token`, defined in the corresponding module. The DFA in figure 8.5 depicts the behaviour of this function. The module also provides a `toString`-function that is used for the creation of error messages during the following syntactic analysis.
Constant values are already separated into the supported data types: Int, Float, Bool, Date, Char and String. It is important to mention that the notation of string constants is not changed. Embedded Curry expressions have to be surrounded by {} to be noticed as such, the value between the curly brackets is not altered during preprocessing.
The only token that needs closer examination is the one for the keyword `TABLE`. As DDL statements are not supported, there is actually no use for it. However, in later phases "table" is used internally as default alias for tables that no pseudonym is defined for, so it has to be prohibited as intended alias. This is achieved by the token for `TABLE`. While all other pseudonyms are read as string constant (and not even noticed to be a pseudonym) occurrences of the string "table" will be replaced by the corresponding token and thus, can be treated differently in later stages.

8.2.5 Syntactic Analysis

The syntactic analysis is based on the grammar given in appendix C, which satisfies the LL(1) property defined in chapter 5. Thus, the recursive descent technique is applied to perform this stage of the compilation process, which is implemented in the `SQLParser`-module.

SQLAst

To parse the list of token and return the statement as an abstract syntax tree a corresponding tree data structure was defined in the module `SQLAst`. The data type is kept close to the grammar but was extended with information needed in subsequent phases. To give some examples, we will have a closer look at the nodes for a column reference, for tables and for the newly introduced foreign key constraint.

```
data ColumnRef = Column Tab String Type Bool Int
data Tab = Unique String | Def [String]
data Type = I | B | F | C | S | D
```

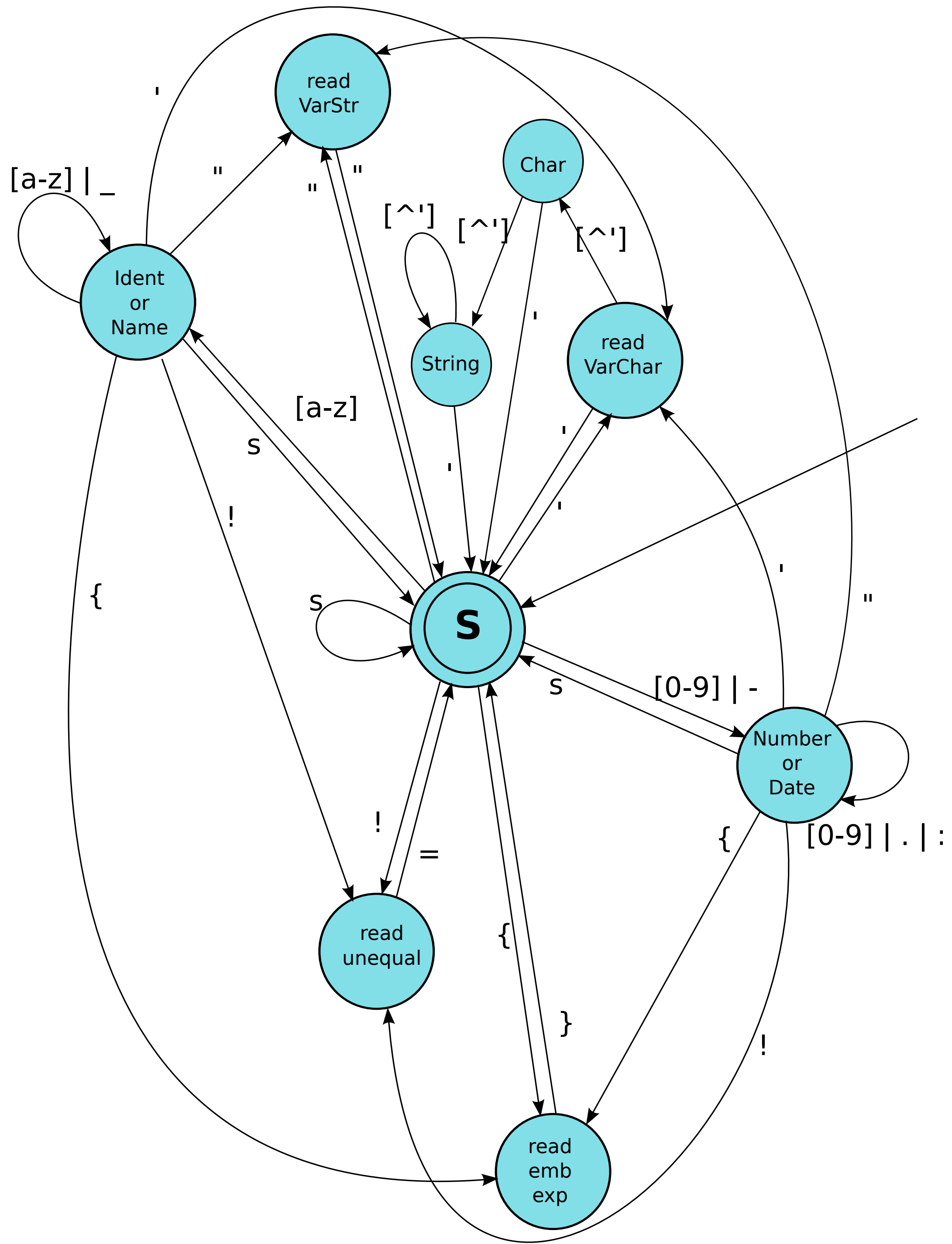

Figure 8.5: DFA for lexical analysis

```
          | Key String
          | Entity String
          | Unknown

data Table = Table String String Int

data Condition = FK (String,Int) AbsRel (String,Int)
               | ...
```

The node representing a column reference provides the following information: the table, which is given by a child node explained below, its name, its type, a flag to indicate whether it can contain null-values and an integer value, which represents the numerical alias used for calls to CDBI functions (CDBI-alias in the following). Note that not all parameters are set during the syntactic analysis. As we have no information about the type, the nullable property or the correct CDBI-alias at this stage, these arguments are for now set to their default values as there are: `Unknown`, `False` and 0 respectively. In case a pseudonym is given for the column a `Unique`-instance is created for the `Tab`-node, the string value is set to the pseudonym for now and is replaced in later phases by the table name. If no pseudonym is given, a default instance is created which at this stage just contains an empty list and is later replaced by a list of all table names no alias was defined for, before it can finally be replaced by a `Unique`-instance.
The data type for column types (and also for embedded expressions) can either be one of the basic types given in the last section, a primary or foreign key (carrying the referenced table as a string argument), an entity type (also carrying the corresponding table name) or a default value used to indicate that the type is not yet known.

The table node provides two fields of type string, the first one for the name, the second one for the alias defined or the default alias "table". The integer argument saves the CDBI-alias.

The foreign key constraint is just one alternative of a condition node. It is given by two tuples representing the tables/entities by a name or pseudonym and an integer value which is used for the CDBI-alias later on. The data type `AbsRel` is similar to the definition of `RelationType` given in section 8.2.2 with the only difference that a constructor for a default value is provided (`NotSpec String`), because at this stage we have no information about the type, but just the name of the relationship.

A sequence of statements is represented as a list of independent abstract syntax trees.

To provide a graphical example, the syntax tree of the following `Select` statement is shown in figure 8.6.

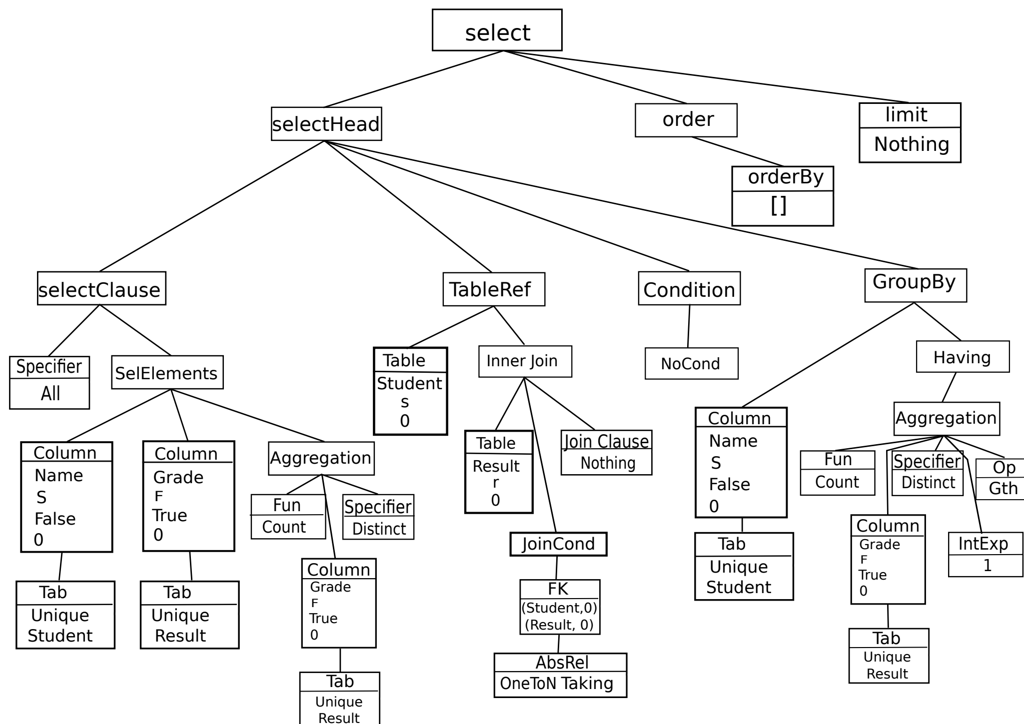

Figure 8.6: examplary AST for `Select` statement

```
Select s.Name, r.Grade, Count( Distinct r.Grade)
 From Student As s Inner Join Result As r On
      Satisfies s has_a r
 Group By s.Name
 Having Count(Distinct r.Grade) > 1;
```

SQLParserTypes

The syntactic analysis is the first stage that makes extensive use of the monadic structure `PM` defined in `currypp` and already introduced briefly in chapter 4. A small extension was made to the corresponding modules `ParseMonad` and `ParseError` with the functions:

```
combinePMs :: (a -> b -> c) -> PM a -> PM b -> PM c

combinePRs :: (a -> b -> c) -> PR a -> PR b -> PR c
```

`combinePMs` corresponds to a combination of the lift and the bind function of the `ParseMonad` if both `PM`s contain a result:

```
combinePMs' f p1 p2 =
 bindPM (liftPM f p1) (\g -> (liftPM g p2))
```

Just in case that both monads contain errors the new function keeps the errors of both `PM`s instead of ignoring the second one. This behaviour is important for a good error management.

For the parsing of SQL statements the `ParseMonad` was further encapsulated by the type `SPM` defined in the `SQLParserTypes`-module:

```
data SPM a = SPM Pos (PM a) [Token]
```

It consists of the position of the integrated code needed to generate errors, the current result or errors as `PM` and the remaining list of tokens to parse. The empty counterpart is given by:

```
data EmptySPM = ESPM Pos [Token]
```

By combining both to a function, a data type was defined, which allows to pass the list of token top-down, while the resulting abstract syntax tree is finally constructed bottom up:

```
type SPMParser a = EmptySPM -> SPM a
```

An extensive interface is provided for this type including functions like

```
-- returns flag whether the list of token is empty or not
hasToken :: EmptySPM -> Bool

-- returns first token - partially defined
headToken :: EmptySPM -> Token

-- cuts first token if there is one
continue :: EmptySPM -> EmptySPM
```

to assure modularity and keep the parser independent of internal changes of the `SPM` type.
Furthermore, we provide the following functions

```
-- return function of a SPMParser
initializeSPM :: a -> SPMParser a

liftSPM :: (a -> b) -> SPMParser a -> SPMParser b

bindSPM :: SPMParser a ->
           (a -> SPMParser b) ->
           SPMParser b
```

to enable the use as a monadic data structure.
The interface also provides terminal parsers and corresponding combinators:

```
terminal :: Token -> EmptySPM -> Either EmptySPM (SPM _)

(.~>.) :: (EmptySPM -> Either EmptySPM (SPM a)) ->
          SPMParser a ->
          SPMParser a

(.<~.) :: SPMParser a ->
          (EmptySPM -> Either EmptySPM (SPM a)) ->
          SPMParser a
```

Both combinators ignore the result of the terminal parser and return an `EmptySPM` in case of success. The (. ~> .) combinator does not even invoke the

second parser if it failed, which is a useful behaviour. Ignoring a part of a statement in case the leading keyword is missing or wrong, often prevents from introducing subsequent errors and helps to reset the parser. Another important combinator is given by:

```
combineSPMs :: (a -> b -> c) ->
               SPMParser a ->
               SPMParser b ->
               SPMParser c
```

The function is used to join two child nodes and create the parent node. At the same time it ensures that warnings as well as errors from both parsers are joined and passed upwards. To achieve this it makes use of the above described function `combinePMs`.

Error Management

An error recovery approach was implemented with the objective to parse as many parts of a statement as possible despite of encountering errors and to provide meaningful error messages for all errors that were found.
Thus, two functions for a parser to return with an error are defined. The first one takes an error message as argument, the second one is predefined for an unexpected empty list of token:

```
parseError :: String -> SPMParser _

emptyTkErr :: SPMParser _
```

Furthermore, the following functions serve to reset the parser. They drop tokens from the list until either encountering the given token or one of the tokens in the passed list respectively. The passed list of token is normally the follow set of the parsed non-terminal. Both functions stop in case of a semicolon.

```
proceedWith :: Token -> EmptySPM -> EmptySPM

proceedWithOneOf :: [Token] -> EmptySPM -> EmptySPM
```

An alternative terminal parser was defined which in case of an error continues with the next token that is member of the passed list (follow set).

```
terminalOrProc :: Token ->
                  [Token] ->
                  EmptySPM ->
                  Either EmptySPM (SPM _)
```

As described above the (. ∼> .) operator and the `combineSPMs` function already provide the correct behaviour with respect to the error management. The bind function still has to be modified to cope with the requirements. Its previous definition would abort the process in case the first parser results in an error as there is no value to pass to the second parser. Thus, it is necessary

to provide a default value which can be passed instead and to bind the errors to the result of the second parsing function.

```
bindDefSPM :: SPMParser a ->
              a -> -- default result of first parser
              (a -> SPMParser b) ->
              [Token] -> --follow set of first parser
              SPMParser b
```

In case of an error the second parsing function is invoked with the next token which also appears in the passed follow set. The default element used instead of an result is ignored when both parsing functions are combined afterwards. The original bind function is also enhanced by a list of tokens representing the follow set which is used to reset the parser in case of error:

```
bindSPM :: SPMParser a ->
           (a -> SPMParser b) ->
           [Token] ->
           SPMParser b
```

With this set of functions it is now possible to parse an incorrect statement nearly completely despite of the encountered errors and to collect all error messages.

The Parsing Process

As we use a LL(1)-grammar, it is sufficient to check the next token to invoke the correct subsequent parsing procedure in any situation. A corresponding parsing function is provided for each deducible rule and has always the same basic structure:

```
parseNT1 :: SPMParser NodeNT1
parseNT1 espm
 | hasToken espm =
    case headToken espm of
           Token1 -> parseToken1Rule (continue espm)
           Token2 -> initializeSPM NodeNT1 espm
           .
           .
           _      -> parseError "message"
                                (proceedWith tok espm)
 | otherwise = emptyTkErr espm
```

`parseTokenNRule` is a pseudonym for any combination of parsing functions. This way the syntax tree is build bottom-up. Each parsing function generates child nodes returning them to its caller. Thus, the syntax tree itself can finally be seen as a synthesized semantic attribute.

8.2.6 Semantic Analysis

The semantic analysis is divided into three phases: the resolution of pseudonyms, the consistency check and the type check. All phases work on the abstract syntax tree build by the parser and enhance the nodes representing columns, tables or values with information which is finally needed for the translation process.
The implementation details of each stage are explained below.

Namer and Symbol table

The resolution of table pseudonyms is implemented by the `SQLNamer`-module, which makes use of a customized symbol table that is described initially.
For efficiency reasons the symbol table is based on the data type `Finite Map` `(FM)` and has the following structure:

```
data Symboltable a b = ST ((FM String a),(FM String b))
                          (Maybe (Symboltable a b))
```

It combines two different Finite Maps and can contain another symbol table itself to implement a scope concept. The symbol table used in the `SQLNamer` is instantiated with parameter types as shown below

```
type AliasSymTab = Symboltable [(String,String, Int)] Int
```

The first FM serves to map a pseudonym as string to a list of triples containing the table name, the pseudonym in its original notation w.r.t. lower- and upper-case letters and the numerical alias that is later used for the translation to CDBI-functions. The second FM keeps track of how often a table name was referenced in the same statement to compute the correct numerical alias and thus, maps the table name to an integer value. Both key arguments are written completely in lower case letters to support case insensitivity.

As it is the meaning of aliases to identify table names throughout a whole statement, the latter FM must not support scopes and is just left unchanged when entering or leaving a scope. The former FM needs to implement a scope concept to properly translate `Exists` constraints. Consider the following statement:

```
Select s.Name, l.Title
  From Student As s
  Where Exists (Select *
                From Participation As p , Lecture As l
                Where p.StudentParticipationKey = s.Key
                  And p.LectureParticipationKey = l.Key);
```

This query is invalid as the pseudonym `l` for the entity `Lecture` is not visible outside of the `Exists` constraint. With a scope concept for the mapping of pseudonyms to table names this error is easily realized because pseudonym `l`

is mapped to entity `Lecture` in the inner scope of the `Exists` constraint, but remains undefined when it is used in the `Select` clause of the outer scope. At the same time, it is ensured that the pseudonym `s` for the `Student` entity defined in the outer scope is visible in the inner scope too. Nevertheless, another pseudonym `s` defined in the inner scope would overwrite the binding to the `Student` entity as the following query demonstrates:

```
Select s.Name, r.Points
 From Student As s, Result As r
  Where s.Key = r.StudentTakingKey
        And Exists (Select *
                    From Exam As s
                    Where r.ExamResultingKey = s.Key);
```

While the expression `s.Key` inside the `Exists` constraint refers to the `Exam` table, the one used in the `Where` clause refers to the `Student` table.

Another important case to consider are compound `Select` statements. The general naming process for a statement with two `selectHeads` [1] is illustrated by figure 8.7. It is important to process every branch of the `Select` statement, i.e., each `selectHead`, with a new symbol table as the bindings are totally independent of each other.
In a first pass through the `From` clause all bindings are collected and inserted into the symbol table. In detail: the current numerical alias is fetched from the second `FM` and inserted into the first one together with the referenced table name and the literal alias, using the notation given by the user. The same literal alias in lower case notation is used as key. The numerical alias is incremented and inserted again into the second `FM`.
The resulting symbol table is passed to all subsequent parts of the `Select` statement except the `Order-by` clause, so that the pseudonyms replacing the table names in front of column references can be resolved. The alias is looked up in the first `FM`, which at this stage allows to check for different notations and to throw an error in case the alias can not be matched with any table name. The pseudonyms used in the foreign key constraint are replaced as well. Finally the symbol tables of all branches are joined, overwriting identical bindings with the latest value. This joined table is used to resolve pseudonyms in the `Order-by` clause, which means that an alias that is used in this part has to be unique in the complete statement.
The use of pseudonyms in SQL is optional as long as the column names are unambiguous. Thus, the following query is totally correct:

```
Select Name, Points From Student, Result;
```

It selects the column "Name" from the table `Student` and column "Points" from table `Result`. However, using the table names themselves to specify column names is also possible:

```
Select Student.Name, Lecturer.Name From Student, Lecturer;
```

[1] see listing 7.1 for further explanation

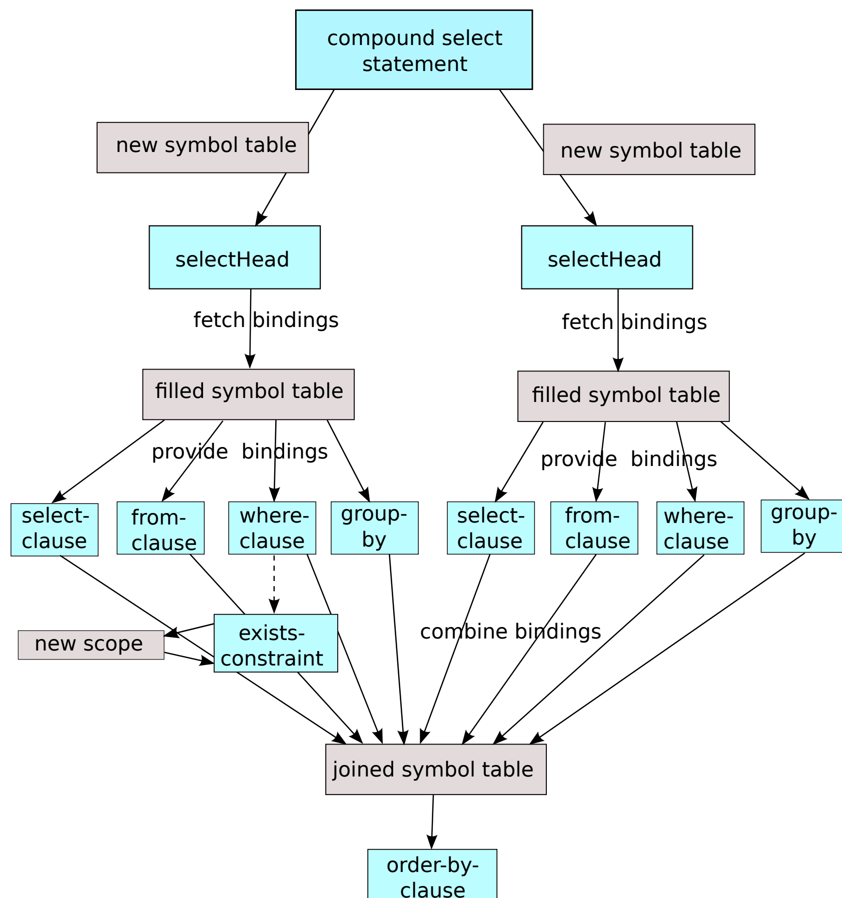

Figure 8.7: Naming process for a compound `Select` statement. Blue fields signify parts of the `Select` statements, grey fields state of the symbol table.

Hence, it is necessary to treat table names without a pseudonym definition differently. Since it is unknown at the moment of insertion into the symbol table whether the table name is used in column references or not, it has to be inserted twice with different keys: the table name itself and the default alias "table". As explained earlier "table" is not a valid alias when defined explicitly so it can be used without any limitations for this purpose. This is also the only occasion where additional bindings for an already existing alias are allowed. This results in the following key - value pairs for the last example:

```
("table", [("Student", "table", 0),
           ("Lecturer", "table", 0)])
("student", [("Student", "Student",0)])
("lecturer", [("Lecturer", "Lecturer", 0)])
```

Since columns that are not specified by an alias are also marked with the default "table", all table names that are available without an alias in the same statement can be easily fetched from the symbol table and inserted into the column node. Thus, the column reference `Points` in the above example would

be represented by the following node before and after the naming stage:

```
-- before
Column (Def ["table"]) "Points" Unknown False 0

--after
Column (Def ["Student","Result"]) "Points" Unknown False 0
```

The numerical alias was also set, but stays 0 in this example. The remaining two parameters still contain their default values and are set in subsequent phases. Which table finally contains the column (if any) is checked during the next stage.
The last example demonstrates the process in case an alias was defined and clarifies how it is ensured that once defined, a pseudonym has to be used. Consider the queries:

```
Select Name From Student As S;
Select s.Name From Student As S;
```

The following key-value pair is inserted into the symbol table for both queries:

```
("s", [("Student", "S", 0)])
```

However, in the first request the column node for ”Name” is marked with the default alias and thus, can not be matched with the `Student` table. An error is returned and the compilation process is aborted. In case of the second request, the column node contains the correct alias, which is then replaced by the table name:

```
--before
Column (Unique "s") "Name" Unknown False 0

--after
Column (Unique "Student") "Name" Unknown False 0
```

Another restriction that has to be guaranteed is the unambiguousness of table names, i.e., queries as

```
Select Name, Name From Student Cross Join Student;
```

are not valid. This is ensured by restricting the insertion of identical pseudonyms into the symbol table or rather, of identical table names into the list of tables matched to the default alias. An error during the filling of the symbol table leads to an immediate abortion of the compilation process.

Finally the correct numerical aliases have to be saved in the corresponding table node to provide it to later phases. To demonstrate this, consider the following query:

```
Select s1.name, s2.name From Student As s1, student As s2
                        Where s1.age /= s2.age;
```

The table nodes after the naming stage are listed below.

```
Table "Student" "s1" 0
Table "student" "s2" 1
```

The remaining statements `Update, Delete` and `Insert` can just contain one single table without an alias, so all work to ensure the correct column - table mapping for those statements is already done.

Consistency Check

To check the consistency with the data model, it is obviously necessary to make extensive use of the passed parser information instance. The main function of the consistency module extracts the data about nullable attributes, relationship types and the attribute lists, which is than passed to the subsequent functions where needed.
It is important to distinguish two Finite Maps of type

```
FM String (String, [String])
```

The first one is retrieved from the parser information file containing all table names and corresponding column names of the data model. The second one is build up at the beginning of the consistency check for each statement, containing just the table names and corresponding column names that were referenced so far in this particular statement. Therefore it is important that in `Select` statements the `From` clause is treated before the other parts of the statement, as well as the table reference in the remaining statement types.

In this section we will focus on the more complex issues of the consistency check, which are the treatment of the foreign key constraint, the preparation of `Insert` statements for further compilation and how a unique table affiliation is found for columns that are still marked as default at the beginning of this stage. The check of column and table names is a rather trivial task so that it is not explained in more detail.

For a proper translation in later modules it is essential to infer the correct relationship type for relations referenced in the foreign key constraints. This can be done by a simple call to the `lookupRel`-function provided by the `SQL-ParserInfoType`-module. To provide case insensitivity, first of all the correct notation of both entity names has to be fetched from the created `FM`. Subsequently the relationship type is retrieved, checked for notation differences and inserted into the condition node including the original notation of all elements. The modification of the condition node is demonstrated with the following example:

```
Select s.Name, r.Name From STUDENT As s, Result As r
                      Where Satisfies s has_a r;
```

Considering the above statement, this is the corresponding condition node before and after the consistency stage:

```
--before
FK ("STUDENT", 0) (NotSpec "has_a") ("Result", 0)
```

```
--after
FK ("Student", 0) (AOneToN "Taking") ("Result", 0)
```

Note that this transformation is a consistency issue and is not concerned with the type check of the next stage since a wrong or non-existent relationship (type) would result in an incorrect translation, not in a type error.

The `Insert` statement has to be checked and prepared very carefully to provide later compilation errors. First of all, we recall the overall structure of an `Insert` statement:

```
INSERT INTO tablename columnlist
       VALUES valuelists;
```

The list of columns is optional if each given value list contains values for all columns of the table. Furthermore, it is allowed to leave out null-values in the value lists. Additionally, the SQL dialect implemented in this thesis allows to skip the value for the key column because it is set automatically. These relaxed grammatical rules can be a huge source of errors when applied incorrectly. The process to detect those errors is explained considering the example below:

```
Insert Into Student (MatNum, Name, Firstname, Email)
       Values (6828, "Krone", "Julia", "jkr@mail.de");
```

Note that we left out the key value and the value for column age, which is allowed to be Null. Before the consistency check the statement is represented by the following syntax tree:

```
Insert
  (Table "Student" "table" 0)
  [(Column (Unique "Student") "MatNum" Unknown False 0),
   (Column (Unique "Student") "Name" Unknown False 0),
   (Column (Unique "Student") "Firstname" Unknown False 0),
   (Column (Unique "Student") "Email" Unknown False 0)]
  [[(IntExp 6828), (StringExp "Krone"),
    (StringExp "Julia"), (StringExp "jkr@mail.de")]]
```

Initially the number of given values has to be equal to the number of columns. If this requirement is fulfilled, the existence and notation of the column names is checked. To prepare the statement for translation, null-values have to be inserted where they are required. Therefore the list of columns given is compared to the list of column names fetched from the parser information for the corresponding table name. A list of integer values is generated indicating the indices where null-values have to be inserted. In case a value is missing for a column which cannot contain null-values, an error is thrown and the process is aborted. Otherwise null-values and default key values are inserted and the statement is finally checked completely to detect explicitly given null-values in illegal positions. Afterwards the above syntax tree is completed as shown below, now containing all column references with a set nullable-flag and an explicit value for each column:

```
Insert
 (Table "Student" "table" 0)
 [(Column (Unique "Student") "Key" Unknown False 0),
  (Column (Unique "Student") "MatNum" Unknown False 0),
  (Column (Unique "Student") "Name" Unknown False 0),
  (Column (Unique "Student") "Firstname" Unknown False 0),
  (Column (Unique "Student") "Email" Unknown False 0),
  (Column (Unique "Student") "Age" Unknown True 0)]
 [[(KeyExp "Student" 42), (IntExp 6828),
   (StringExp "Krone"), (StringExp "Julia"),
   (StringExp "jkr@mail.de"), AbsNull]]
```

The last transformation that is to be described in this section, is the allocation of an unique table reference to columns marked as default. Therefore we will recall the above exemplary statement:

```
Select Name, Points From Student, Result;
```

and the resulting column nodes:

```
Column (Def ["Student","Result"]) "Points" Unknown False 0
Column (Def ["Student","Result"]) "Name" Unknown False 0
```

To determine which of the tables contains the corresponding column (if any), the list of columns for each table is retrieved from the map built at the beginning of this stage. All names of tables that contain the column name are collected. In case more than one table name is obtained the column reference is ambiguous and an error is returned. The statement is also erroneous if no table name is obtained, because that means no appropriate table was referenced. If exactly one table name is obtained, it is set as unique reference in the corresponding column node:

```
-- data fetched from the parser information module
("result", ("Result",
            ["Key", "Attempt", "Grade", "Points",
             "StudentTakingKey", "ExamResultingKey"]))
("student", ("Student",
             ["Key", "MatNum", "Name", "Firstname",
              "Email", "Age"]))

-- changed column nodes
Column (Unique "Result") "Points" Unknown False 0
Column (Unique "Student") "Name" Unknown False 0
```

Note that the nullable-flag is not set in this case as the translation of columns that can contain null-values does not differ from those that cannot in any statement except the `Insert` statement.

Type Check

This stage is divided into two sub-phases. In the first step the AST is enhanced with type attributes retrieved from the parser information. The second phase verifies the correct usage to prevent later type errors. The type information is also very important for the code generation.

The first phase is a simple top-down tree-traversal. Each column node is modified by the following function:

```
typColumnRef :: Pos ->
                AttrTypeFM ->
                ColumnRef ->
                PM ColumnRef
```

The passed FM contains the type for every column name specified in the data model. So the type can be retrieved and inserted into the column node, which is then returned.

The core functionality of the second stage is given by the following three comparisons of different nodes:

- comparison of two values
- comparison of a value and a column
- comparison of two columns

When examining the definition of the value node:

```
data Value = Emb String Type
           | IntExp Int
           | KeyExp String Int --table name and value
           | FloatExp Float
           | StringExp String
           | DateExp CalendarTime
           | BoolExp Bool
           | CharExp Char
           | AbsNull
```

there stand out three instances which need special treatment: embedded Curry expressions, key expressions (also key columns) and null-values.

If an embedded Curry expression is compared to a column, it is assumed to have the same type as the column. Thus, its type attribute is set and a warning is generated. There is no possibility to identify the type of the embedded Curry expression definitely, it can just be inferred by contextual information and demonstrates the limitations of a type check during preprocessing. As an example we consider the following condition:

```
... Where Student.Age < {x}...
```

`x` is an embedded Curry variable. So far the column and the embedded Curry expression are represented by the following nodes respectively:

```
(Column (Unique "Student") "Age" I True 0)
(Emb "x" Unknown)
```

Although we have no further information about `x`, the type attribute will be changed from `Unknown` to `I`, which represents an integer value.
As a consequence an embedded Curry expression cannot be compared to another embedded Curry expression, since no contextual information is available. Embedded Curry expressions can also appear in `Update` and `Insert` statements representing complete entities. In this case the type is inferred from the table name and a warning is generated as well. For instance, the `Update` statement

```
Update Student Set {student};
```

is represented by the following simple abstract syntax tree after the type inference:

```
UpdateEntity (Table "Student" "table" 0)
             (Emb "student" (Entity "Student"))
```

In a comparison of two key columns both columns have to reference the same table in order to be accepted. Note that key values are integer expressions up to this stage, only when compared to a key column the contextual information is used here too and the value is modified to represent a key expression. Each key expression holds beside the actual value the referenced table name which is used for the translation. This way there is no need to differentiate between primary keys and foreign keys at the abstract syntax tree level. The condition below gives an example:

```
... Where Student.Key = 1...
```

After the first phase of the type check, column and value are represented by the following two nodes:

```
(Column (Unique "Student") "Key" (Key "Student") False 0)
(IntExpr 1)
```

After the second type checking phase the node representing the integer expression is replaced by:

```
(KeyExp "Student" 1)
```

Other constant values as e.g., floating point numbers, lead to an error.

Null values are not accepted at all in condition clauses and case expressions. Their appearance always results in an error.

For all remaining value types the comparisons above return successfully in case both expressions are of the same type and throw a type error otherwise.

The description given so far works fine for conditions and case expressions. However, the `Insert` and the `Update` statement require some additional treatment.
First of all null-values are cause no problems in `Insert` statements once they passed the consistency check, i.e., the comparison functions explained above do not apply here. The same occurs with key expressions which were explicitly set during the last stage.
In case of an `Update` statement a main requirement for the preprocessor was to prevent the explicit manipulation of key columns. This can be easily achieved at the type checking stage, i.e., a reference of a key column as part of an assignment provokes an error.

The last expressions that need special attention are aggregation functions in `Select` clauses as well as in `Having` clauses. In the former case it has to be ensured that the functions `Sum` and `Avg` are only applied to numerical columns. In the latter case the comparison value has to be checked too, i.e., it has to be matched against a default floating-point value in case of the `Avg`-function and against a default integer value in case of the `Count` function.

8.2.7 Code Generation

The last phase of the implemented SQL compiler is the code generation. This section describes only the most important parts in detail, which are:

- the translation of the foreign key constraint
- using type information for a translation into type safe functions
- using the nullable-flag and type information for the translation of the `Insert` statement
- the translation of transactional statements

The translation of the remaining statement parts uses the same techniques as described below and thus, can be deduced from the given examples and explanation.

Foreign Key Constraints

The main idea behind the foreign key abstraction was to use relationship names instead of generated column names. As the underlying library does not support such an abstraction, the given relationship names have to be resolved and replaced by corresponding foreign key columns. Beside the name of the relation, the type of the relationship is an essential information as it implies in

which entity the foreign key was inserted. With the help of this knowledge it is finally possible to reconstruct the name of the foreign key and connect both entities with an ordinary constraint using column names.
This process is demonstrated for the already known example:

```
Select s.Name, r.Points
  From Student As s Inner Join Result As r
       On Satisfies s has_a r;
```

At the beginning of the translation phase the constraint is represented by the node:

```
FK ("Student", 0) (AOneToN "Taking") ("Result", 0)
```

As the relationship is of type 1-to-n in reading direction, it can be inferred that the foreign key was inserted into the `Result` table. Following the naming scheme, it is called: "StudentTakingKey". Applying additionally the naming convention of CDBI the foreign key column has the name: `resultColumnStudentTakingKey`. This results in the following translation:

```
Select s.Name, r.Points
  From Student As s Inner Join Result As r
       On s.Key = r.StudentTakingKey;
```

While the translation of n-to-1 relationships does only differ in the location of the foreign key, m-to-n relationships are converted into an `Exists` constraint. The second example demonstrates the differences:

```
Select s.Name, l.Title
  From Student As s Inner Join Lecture As l
       On Satisfies s participated l;

FK ("Student", 0) (AMToN "Participation") ("Lecture", 0)
```

In case of an m-to-n relationship the information given in the abstract syntax tree already implies that an additional entity called "Participation" was created with two foreign keys namely "StudentParticipationKey" and "LectureParticipationKey". So we reformulate the above query as follows:

```
Select s.Name, l.Title
 From Student As s, Lecture As l
 Where Exists (Select *
                From Participation As p
                Where p.StudentParticipationKey = s.Key
                  And p.LectureParticipationKey = l.Key);
```

The formulation of the `Exists` constraint in CDBI-functions is shown below:

```
(Exists participationTable 0
      ( And [((col participationColumnStudentParticipationKey)
                          .=. (colNum studentColumnKey 0)),
             ((col participationColumnLectureParticipationKey)
                       .=. (colNum lectureColumnKey 0))] ))
```

Type-safe functions

As seen before, the type inference on embedded Curry expressions is a task that can only be solved insufficiently during preprocessing. The requirements of the translation stage now show the exigence of this inference.
Embedded Curry expressions are allowed in two situations: as complete entities in `Update` and `Insert` statements and as constant values in assignments and condition clauses. In the latter case the type of the expression was inferred from the type of the column included in the same assignment or the same condition. Since the functions of the CDBI interface provide definitive type safety, each value or column that is part of a condition has to be converted into the type `Value a`. A specific constructor function is provided for each supported type, e.g.,

```
string :: String -> Value String
```

Thus, to provide a correct translation, the type information has to be used to choose the adequate constructor function. For instance, the constraint

```
... Where Student.Age < {x}..
```

selects all students younger than the value of a dynamically passed variable `x`. During the type checking, the type of x was set to `I` indicating an integer, as this is the type of column `Age`. Thus, the following translation is generated:

```
... ((colNum studentColumnAge 0) .<. (int x))...
```

For constant values that were already matched with corresponding constructors during the lexical analysis the translation is obvious:

```
... Where Student.Name = "Krone"...
-- node for the constant value:
(VarStr "Krone")
--translated to:
...((colNum studentColumnName 0) .=. (string "Krone"))...
```

In case of complete entities the type was inferred from the given table name. For instance, in

```
update1 :: Student -> IO ( SQLResult ())
update1 student = ``sql Update Student Set {student};''
```

the type of variable `student` is set to type `(Entity "Student")`, which is finally needed to address the correct entity description:

```
update1 :: Student -> IO ( SQLResult ())
update1 student =
      ... updateEntry (student) studentDescription
```

`Insert` statement

The CDBI function which represents the `Insert` operation has the following signature:

```
saveEntry :: a -> EntityDescription a -> DBAction ()
```

Thus, the list of values given in an `Insert` statement has to be converted into the corresponding entity type `a`. The nullable-flag which was set for all columns referenced in `Insert` statements during the consistency check, is now required for a proper translation. Consider the following statement:

```
Insert Into Student
       Values (8,6828,"Julia","Krone","julia@mail.de",26);
```

As seen in the last section, the abstract syntax tree was already enhanced by the information that the integer value 8 at the beginning has to be a key value and that the integer value 26 at the end corresponds to the column `Age`, which can contain null-values. The CDBI library uses `Maybe` types to cope with null-values, so the information inferred during the semantic analysis is now used to provide the translation below:

```
let entry = Student (StudentID 42)
                    6828
                    "Julia"
                    "Krone"
                    "julia@mail.de"
                    (Just 26)
 in saveEntry entry studentDescription
```

Recall that the integer 42 is the default value for keys and replaced in any case since the key columns are defined as auto incrementing.

Transactional statements

As the transactional statements `Begin`, `Commit`, and `Rollback` require no complex syntactic nor semantic analysis, they were not mentioned up to this stage. Nevertheless their translation differs from the rest of the statements as the corresponding CDBI-functions do not return an `SQLResult`, but simply an `IO()`-type. Since there should be a unique interface to all translated statements to facilitate the inclusion into larger programs, the translation enhances the statement by an empty result:

```
--SQL-Transaction-Statement:
Commit

--Translation:
(\c -> (commit c) >> (return (Right ())))
```

where `c` is a `Connection`. Now the return type was changed to `SQLResult()`, which also enables a concatenation with other statements.

Putting it together

The translated SQL statement is finally passed to the `runWithDB`-function introduced in chapter 7 together with the database path, which was fetched from the parser information. A small example is given by the complete translation of the above `update` statement:

```
update1 :: Student -> IO ( SQLResult ())
update1 student =
  runWithDB
    "Uni.db"
    (updateEntry (student) studentDescription)
```

The layout of this example was changed to a more readable format.
More complex statements and their translation can be found in appendix D.

Another requirement for the translation is to hold the specification of `currypp` concerning the layout. To ensure that error messages are assigned to the correct line, the layout of the complete file must not change due to preprocessing. Thus, the translated code is not allowed to take more than a single line.
This requires a slight modification of the translation obtained by the pretty-printing-function for `AbstractCurry`-expressions, which formats the code according to the Curry layout rules. Line feed characters are replaced by a space character and indentation is removed completely.

Statements in a sequence as well as statements in a transaction are connected by the operator

```
(>+) :: DBAction a -> DBAction b -> DBAction b
```

(defined in the `CDBI.Connection`-module), which just returns the result of the last statement and ignores the former ones.

Chapter 9

Integration into currypp

Finally the implemented compiler has to be integrated into the `currypp` executable. As figure 8.1 shows, the project connects the preprocessor with the CDBI library. Hence, the integration requires two modifications:

1. the parser information file has to be passed to the SQL compiler via the preprocessor
2. a language tag for SQL has to be introduced

Thus, the main module of `currypp` was altered to support an additional argument indicating the name of the parser information file. It can be passed using the following notation:

```
--model:<file name>
```

The parameter is optional, i.e., in case no SQL statement is included into the preprocessed file it can be left out without causing any problems. However, in case the argument is given the file has to exist and must contain an instance of the data type `ParserInfo`, which was described in section 8.2.3. Otherwise it is replaced by an empty string.
A complete and correct call to the `currypp`-tool with an input-file containing SQL code is given by:

```
currypp org-filename
        input-file
        output-file
        --foreigncode
        --model:<file name>
```

The file name is passed as a string to the `Translator`-module of the `currypp` project, where the read-function of the `SQLConverter` is called. It is important to read the information file before the original program is separated into chunks of integrated and pure Curry code. Otherwise the same file would be read

several times, once for each SQL language tag, which is obviously inefficient. The read parser information instance is then passed to the `parse` function which represents the interface between the `currypp`-tool and the compiler and is discussed in 8.2.3. This last step is also shown in the listing below.

The language tag "sql" can be introduced by an easy extension of the function `parsers` in the `Translator`-module:

```
parsers :: Maybe Langtag ->
           Either String ParserInfo ->
           LangParser
parsers = maybe iden pars
  where
    iden _ _ s = return $ cleanPM s
    pars :: Langtag ->
            Either String ParserInfo ->
            LangParser
    pars l model p =
      case l of
        "sql"  -> case model of
                    Left err -> (\_ -> return $ throwPM
                                                      p
                                                      err)
                    _   -> SQLParser.parse model p
        .
        .
        .
        "regex" -> RegexParser.parse p
        _       -> (\_ -> return $ throwPM
                                   p
                                   ("Bad langtag: "++l))
```

In case the `parsers`-function is called with language tag "sql" the parameter for the parser information is checked and an error is thrown in case the file contained the wrong data type. Otherwise the SQL-compiler is invoked. If the .info file generated by the `erd2cdbi` tool (see 7.4) is used, there should be no compatibility problems.
Note that the compiler is called for each SQL language tag separately. A single tag can be followed by an arbitrary amount of SQL statements as long as they are separated by semicolons.

IV

Conclusion

Chapter 10

Summary

An SQL compiler for the functional logic programming language Curry was introduced, which serves as an extension for the preprocessor `currypp` presented in [15]. Furthermore, some enhancement for the database library CDBI were implemented that improve the type safety and the expressive power regarding `Select` statements. Included into the preprocessor, the implemented project provides the possibility to include SQL statements directly into a Curry program. Thus, database queries can be written applying a concise and well-known syntax, while at the same time type safety and a functional logic programming style can be maintained.
In addition, the presented SQL dialect offers a new abstraction for foreign key constraints making use of the relationships that were defined by the entity-relationship-diagram. This abstraction can be a large facilitation to the user as it avoids the reference of foreign key columns in SQL statements. Thus, a deeper knowledge about the used generation tools, as e.g., `erd2curry`, is no longer necessary.

Usability was an important requirement for the design of the implemented SQL dialect, the compiler and the corresponding tools. Thus, a very common dialect was implemented. Case-insensitivity is provided, but a basic safety level is still kept by warning the user in case of differing notations. Furthermore, an error recovery approach was implemented for the parsing stage and the modular design of the compiler supports the detection of errors in early stages of the preprocessing process (in particular before the compilation of the preprocessed file). A supplemental option also enables the use of existent, maybe shared, databases.

The compiler was tested with the data model of a university presented in chapter 4. All kinds of supported statements could be processed with satisfactory results. The supported part of the DML finally includes `Select, Insert, Delete, Update` and transactional statements. The behaviour in case of erroneous queries was also tested extensively.

With the presented functional range, the created database interface features a good alternative to the currently used library applying dynamic predicates. The underlying enhanced CDBI library reduces the traffic while providing larger expressive power and type safety. Due to the preprocessor extension even complex queries can now be written with concise SQL statements.

Chapter 11

Prospect

Although the presented database interface, composed by the SQL compiler as part of the preprocessor, the CDBI library and the `erd2cdbi`-tool, already provide a relatively large scope of operation, future work can highly improve the project in both: safety and function volume.

As seen in chapter 8.2, the type inference on embedded Curry expressions is currently based on contextual information. This can be a source of error, which stays undetected until the compilation of the translated file. It would be desirable to ensure complete type safety including embedded Curry expressions already at the stage of preprocessing in later versions of the compiler. Therefore it would be necessary to extend `currypp`, so that beside the chunk of integrated code, also the type information is passed to the SQL compiler. Obviously, this raises further questions, e.g., how the type information can be obtained in case of local definitions, before it is computed by the type inference mechanism of the Curry system. The following example demonstrates the problem:

```
func :: IO (SQLResult ())
func =
  let name = callFuncToGetName
  in ``sql Delete From Student Where Name = {name};''
```

An easy solution would be to demand the type information explicitly from the user, which is undesirable. This approach reduces usability since it hampers the inclusion of SQL expressions into Curry code as normal functions.
Once a satisfactory solution is found, it is possible to provide more dynamic queries and allow e.g., complete condition clauses as embedded Curry expressions, which would require extensions concerning the consistency check.

Another improvement could be the inclusion of the `erd2curry` transformation into `erd2cdbi` to reduce the actions necessary to invoke the tool. The `erd2cdbi` tool should also be enhanced by a function which ensures the con-

sistency of a given, already existent database with the applied data model. This is even more important as providing the name of the wrong database to the `erd2cdbi`-tool would not result in compilation but in run time errors.

Not least, the CDBI library is still expandable to support more SQL expressions, the current limitations listed in chapter 7 have to be mentioned here. The support of another database software could also be a useful enhancement of the interface. Furthermore, [16] noted some more possible improvements concerning the allowed types of values and the possible error kinds which were not yet implemented. In this context a support of the user-defined type proposed by `erd2curry` is also imaginable.

Appendix A

Installation and Usage

This section explains the installation process and the usage of the preprocessor for embedded SQL statements on a Linux machine. For Windows the process might differ. An installed Curry compiler, preferably KiCS2, is assumed. It is also possible to use PAKCS for the installation but this has currently a negative impact on the runtime of the compiler
First of all, the preprocessor has to be installed. The recent version containing the extension for SQL is currently available here [1], but will be included in later versions of both Curry implementations.
Since the translated version of SQL statements is dependent on the CDBI library, the tool `erd2cdbi` has to be installed too. Currently the source code can be downloaded here [2], but the tool is also already included in PAKCS (version 1.13.1 or higher) and KiCS2 (version 0.4.1 or higher), so the installation of a current version of the Curry implementation suffices.

Download the extended preprocessor version using the link given below.
To install the project from the source code, first change to the SqlCurryPP directory. The Makefile is configured for the use of KiCS2. For the use of PAKCS uncomment line 12 and comment line 13. Then run `make`.

Next change to your projects main directory and run `erd2cdbi`:

```
erd2cdbi <ERDName_ERDT.term>
         <absolute path to the database/dbName.db>
         [-db]
```

The first parameter represents the name of the file containing the ERD-term of the data model in use. Make sure to use the transformed version of the ERD-term as generated by `erd2curry` (see chapter 4.2). The second parameter has to be the absolute path to the database in case it is already existent or to the directory where to create it otherwise. In either case the name of the database

[1] https://git-ps.informatik.uni-kiel.de/theses/2015-jkr-ma/trees/master/SqlCurryPP
[2] https://git-ps.informatik.uni-kiel.de/theses/2015-jkr-ma/trees/master/erd2cdbi

has to be given. The last parameter is optional, if provided, a new empty database is created otherwise a database corresponding the given ERD-term is expected in the specified directory and with the given name.

Two files will be created in the current directory. A curry-file with the name *ERDName*_CDBI.curry containing the data types for CDBI and a file named *ERDName*_SQLCode.info which contains the information needed during the translation process.
Include the module *ERDName*_CDBI.curry into the program you want to preprocess. Note that it is also necessary to include the module CDBI.ER to use the interface.
To translate the embedded expressions now run the preprocessor:

```
currypp <org-filename>
        <input-file>
        <output-file>
        --foreigncode
        --model:ERDName_SQLCode.info
```

The option --foreigncode invokes the translation process, the --model option is followed by the .info-file that was created by `erd2cdbi` and specifies which data model to use for the translation of SQL statements. In case the preprocessing was successful the output file can now be used as any other Curry module.

With the inclusion of compiler directives like the following one at the beginning of the file containing embedded foreign code it is also possible to load the program directly with PAKCS or KiCS2.

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp
                        --optF=--foreigncode
                        --optF=--model:Uni_SQLCode.info #-}
```

Appendix B

CDBI Extensions - Overview

This section lists all exported data types and functions that were changed or added to the CDBI library ordered by modules. The original comments are given for new data types and functions whereas for changed data just the modified part is described.

Module CDBI.Description

```
--- Datatype representing columns for selection.
--- This datatype has to be distinguished from type Column
--- which is just for definition of conditions.
data ColumnDescription a = ColDesc String
                                   SQLType
                                   (a -> SQLValue)
                                   (SQLValue -> a)

--- A second conversion function was inserted converting
--- the key value always to SQLNull to ensure
--- that keys are auto incrementing.
data EntityDescription a = ED String
                              [SQLType]
                              (a -> [SQLValue])
                               --for insertion
                              (a -> [SQLValue])
                              ([SQLValue] -> a)

--- A second conversion function was inserted converting
--- the key value always to SQLNull to ensure
--- that keys are auto incrementing.
data CombinedDescription a = CD [(Table, Int, [SQLType])]
                                ([SQLValue] -> a)
                                (a -> [[SQLValue]])
```

```
                                                -- for insertion
                                                (a -> [[SQLValue]])
```

Module CDBI.Criteria

```
--- Instead of a list of Options the Criteria type
--- does contain the group-by-clause now
data Criteria = Criteria Constraint (Maybe GroupBy)

--- specifier for queries
data Specifier = Distinct | All

--- data type to represent group-by statement
data GroupBy = GroupBy CValue GroupByTail

--- subtype for additional columns or
--- having-Clause in group-by statement
data GroupByTail = Having Condition
                 | GBT CValue GroupByTail
                 | NoHave

--- data type for conditions inside a having-clause
data Condition = Con Constraint
               | Fun String Specifier Constraint
               | HAnd [Condition]
               | HOr [Condition]
               | Neg Condition

--- Constructor for Values of ID-types
idVal :: Int -> Value _

---Constructor for the group-by-clause
groupBy :: Value a -> GroupByTail -> GroupBy

--- Constructor to specifiy more than one
--- column for group-by
groupByCol :: Value a -> GroupByTail -> GroupByTail

---Constructor for simple having condition
having :: Condition -> GroupByTail

--- Constructor for empty having-Clause
noHave :: GroupByTail

---Constructor for Condition with just a simple Constraint
condition :: Constraint -> Condition
```

```
---Constructor for aggregation function sum for
---columns of type Int having-clauses.
sumIntCol :: Specifier ->
             Value Int ->
             Value Int ->
             (Value () -> Value () -> Constraint) ->
             Condition

--- Constructor for aggregation function sum for
--- columns of type float in having-clauses.
sumFloatCol :: Specifier ->
               Value Float ->
               Value Float ->
               (Value () -> Value () -> Constraint) ->
               Condition

--- Constructor for aggregation function avg for
--- columns of type Int in having-clauses.
avgIntCol :: Specifier ->
             Value Int ->
             Value Float ->
             (Value () -> Value () -> Constraint) ->
             Condition

--- Constructor for aggregation function avg for
--- columns of type float in having-clauses.
avgFloatCol :: Specifier ->
               Value Float ->
               Value Float ->
               (Value () -> Value () -> Constraint) ->
               Condition

---Constructor for aggregation function count
---in having-clauses.
countCol :: Specifier ->
            Value _ ->
            Value Int ->
            (Value () -> Value () -> Constraint) ->
            Condition

--- Constructor for aggregation function min
--- in having-clauses.
minCol :: Specifier ->
          Value a ->
          Value a ->
          (Value () -> Value () -> Constraint) ->
          Condition
```

```
--- Constructor for aggregation function max
--- in having-clauses.
maxCol :: Specifier ->
          Value a ->
          Value a ->
          (Value () -> Value () -> Constraint) ->
          Condition
```

Module CDBI.QueryTypes

```
---datatype for set operations
data SetOp = Union | Intersect | Except

--- datatype for joins
data  Join =  Cross | Inner Constraint

--- Constructor for  inner join
innerJoin :: Constraint -> Join

--- Constructor for cross join
crossJoin :: Join

--- data structure to represent a table-clause
--- (tables and joins) in a way that at least
--- one table has to be specified
data TableClause = TC Table Int (Maybe (Join,TableClause))

--- Constructor function for expression:
--- CASE WHEN condition THEN val1 ELSE val2 END.
--- It does only work for the same type in then
--- and else branch.
caseThen :: Condition ->
            Value a ->
            Value a ->
            (CaseVal a) ->
            ColumnSingleCollection a

--- Constructor function for ColumnSingleCollection.
singleCol :: ColumnDescription a ->
             Int ->
             (ColumnDescription a -> Fun b) ->
             ColumnSingleCollection b

---Constructor function for ColumnTupleCollection.

tupleCol :: ColumnSingleCollection a ->
            ColumnSingleCollection b ->
```

```
            ColumnTupleCollection a b

---Constructor function for ColumnTripleCollection.
tripleCol :: ColumnSingleCollection a
          -> ColumnSingleCollection b
          -> ColumnSingleCollection c
          -> ColumnTripleCollection a b c

---Constructor function for ColumnFourTupleCollection.
fourCol :: ColumnSingleCollection a
        -> ColumnSingleCollection b
        -> ColumnSingleCollection c
        -> ColumnSingleCollection d
        -> ColumnFourTupleCollection a b c d

---Constructor function for ColumnFiveTupleCollection.
fiveCol :: ColumnSingleCollection a
        -> ColumnSingleCollection b
        -> ColumnSingleCollection c
        -> ColumnSingleCollection d
        -> ColumnSingleCollection e
        -> ColumnFiveTupleCollection a b c d e

--- Data type to describe all parts of a select-query
--- except set operators  order-by and limit
--- (selecthead) for a single column.
data SingleColumnSelect a  =
          SingleCS Specifier
                   (ColumnSingleCollection a)
                   TableClause
                   Criteria

--- Data type to describe all parts of a select-query
--- except set operators  order-by and limit
--- (selecthead) for two columns.
data TupleColumnSelect a b =
           TupleCS  Specifier
                    (ColumnTupleCollection a b)
                    TableClause
                    Criteria

--- Data type to describe all parts of a select-query
--- except set operators  order-by and limit
--- (selecthead) for three columns.
data TripleColumnSelect a b c =
            TripleCS Specifier
                     (ColumnTripleCollection a b c)
                     TableClause
```

```
                    Criteria

--- Data type to describe all parts of a select-query
--- except set operators  order-by and limit
--- (selecthead) for a four columns.
data FourColumnSelect a b c d =
          FourCS Specifier
                 (ColumnFourTupleCollection a b c d)
                 TableClause
                 Criteria

--- Data type to describe all parts of a select-query
--- except set operators  order-by and limit
--- (selecthead) for five columns.
data FiveColumnSelect a b c d e =
             FiveCS Specifier
                    (ColumnFiveTupleCollection a b c d e)
                    TableClause
                    Criteria

--- Constructor for aggregation function sum
--- in select-clauses.
--- A pseudo-ColumnSingleCollection of type
--- float is created for correct return type.
sum :: Specifier -> ColumnDescription _ -> Fun Float

--- Constructor for aggregation function avg
--- in select-clauses.
--- A pseudo-ColumnSingleCollection of type
---  float is created for correct return type.
avg :: Specifier -> ColumnDescription _ -> Fun Float

--- Constructor for aggregation function count
--- in select-clauses.
--- A pseudo-ColumnSingleCollection of type
--- float is created for correct return type.
count :: Specifier -> ColumnDescription _ -> Fun Int

--- Constructor for aggregation function min
-- in select-clauses.
minV :: ColumnDescription a -> Fun a

--- Constructor for aggregation function max
-- in select-clauses.
maxV :: ColumnDescription a -> Fun a

--- Constructor function in case no aggregation
-- function is specified.
```

```
none :: ColumnDescription a -> Fun a

---Constructor for CaseVal of type Int
caseResultInt :: CaseVal Int

---Constructor for CaseVal of type Float
caseResultFloat :: CaseVal Float

---Constructor for CaseVal of type String
caseResultString :: CaseVal String

---Constructor for CaseVal of type Date
caseResultDate :: CaseVal Time.ClockTime

---Constructor for CaseVal of type Bool
caseResultBool :: CaseVal Bool

---Constructor for CaseVal of type Char
caseResultChar :: CaseVal Char
```

Module CDBI.ER

```
--- specifier, group-by-clause and limit-clause were added
getEntries :: Specifier ->
              EntityDescription a ->
              Criteria ->
              [Option] ->
              Maybe Int  ->
              DBAction [a]

--- Gets a single Column from the database.
getColumn :: [SetOp] ->
             [SingleColumnSelect a] ->
             [Option] ->
             Maybe Int ->
             DBAction [a]

--- Gets two Columns from the database.
getColumnTuple :: [SetOp] ->
                  [TupleColumnSelect a b] ->
                  [Option] ->
                  Maybe Int ->
                  DBAction [(a,b)]

--- Gets three Columns from the database.
getColumnTriple :: [SetOp] ->
                   [TripleColumnSelect a b c] ->
```

```
                    [Option] ->
                    Maybe Int ->
                    DBAction [(a,b,c)]

--- Gets four Columns from the database.
getColumnFourTuple :: [SetOp]
                   -> [FourColumnSelect a b c d]
                   -> [Option]
                   -> Maybe Int
                   -> DBAction [(a,b,c,d)]

--- Gets five Columns from the database.
getColumnFiveTuple :: [SetOp]
                   -> [FiveColumnSelect a b c d e]
                   -> [Option]
                   -> Maybe Int
                   -> DBAction [(a,b,c,d,e)]

--- specifier, group-by-clause and limit-clause were added
getEntriesCombined :: Specifier ->
                      CombinedDescription a ->
                      [Join] ->
                      Criteria ->
                      [Option] ->
                      Maybe Int ->
                      DBAction [a]
```

Appendix C

Grammar

This section shows the supported part of SQL in EBNF. The grammar fulfills the LL(1) property and was influenced by the grammar given in [17] and the SQLite-dialect [1].

```
--------------type of statements--------------------------------

 statement ::= queryStatement | transactionStatement
 queryStatement ::= ( deleteStatement
                    | insertStatement
                    | selectStatement
                    | updateStatement )
                    ';'

 ------------- transaction ------------------------------------

 transactionStatement ::= (BEGIN
                          |IN TRANSACTION '(' queryStatement
                                         { queryStatement }')'
                          |COMMIT
                          |ROLLBACK ) ';'

-------------- delete -----------------------------------------

 deleteStatement ::= DELETE FROM tableSpecification
                                 [ WHERE condition ]

 -------------insert -----------------------------------------

 insertStatement ::= INSERT INTO tableSpecification
                                 insertSpecification

 insertSpecification ::= ['(' columnNameList ')' ] valuesClause

 valuesClause ::= VALUES valueList
```

[1] https://sqlite.org/lang.html

```
 ------------update ------------------------------------------

updateStatement ::= UPDATE tableSpecification
                    SET (columnAssignment {',' columnAssignment}
                                             [ WHERE condition ]
                        | embeddedCurryExpression )

columnAssignment ::= columnName '=' literal


-------------select statement --------------------------------

selectStatement ::= selectHead { setOperator selectHead }
                               [ orderByClause ]
                               [ limitClause ]
selectHead ::= selectClause fromClause
                            [ WHERE condition ]
                            [ groupByClause [ havingClause ]]

setOperator ::= UNION | INTERSECT | EXCEPT

selectClause ::= SELECT [( DISTINCT | ALL )]
                        ( selectElementList | '*' )

selectElementList ::=  selectElement { ',' selectElement }

selectElement ::= [ tableIdentifier'.' ] columnName
                 | aggregation
                 | caseExpression

aggregation ::= function '(' [ DISTINCT ] columnReference ')'

caseExpression ::= CASE WHEN condition THEN operand
                                       ELSE operand END

function ::= COUNT | MIN | MAX | AVG | SUM

fromClause ::= FROM tableReference  { ',' tableReference }

groupByClause ::= GROUP BY columnList

havingClause ::= HAVING conditionWithAggregation

orderByClause ::= ORDER BY columnReference [ sortDirection ]
                           {',' columnReference
                                [ sortDirection ] }

sortDirection ::= ASC | DESC

limitClause = LIMIT integerExpression

-------------common elements ---------------------------------

columnList ::=  columnReference  { ',' columnReference }
```

```
columnReference ::= [ tableIdentifier'.' ] columnName

columnNameList ::= columnName  { ',' columnName}

tableReference ::=  tableSpecification [ AS tablePseudonym ]
                                       [ joinSpecification ]
tableSpecification ::= tableName

condition ::=   operand operatorExpression
                        [logicalOperator condition]
              | EXISTS subquery [logicalOperator condition]
              | NOT condition
              | '(' condition  ')'
              | satConstraint [logicalOperator condition]

operand ::=  columnReference
            | literal


subquery ::= '(' selectStatement ')'

operatorExpression ::=  IS NULL
                      | NOT NULL
                      | binaryOperator operand
                      | IN setSpecification
                      | BETWEEN operand operand
                      | LIKE quotes pattern quotes

setSpecification ::=  literalList

binaryOperator ::= '>'| '<' | '>=' | '<=' | '=' | '!='

logicalOperator ::= AND | OR

conditionWithAggregation ::=
        aggregation [logicalOperator disaggregation]
      | '(' conditionWithAggregation  ')'
      | operand operatorExpression
               [logicalOperator conditionWithAggregation]
      | NOT conditionWithAggregation
      | EXISTS subquery
               [logicalOperator conditionWithAggregation]
      | satConstraint
               [logicalOperator conditionWithAggregation]

aggregation ::= function '('(ALL | DISTINCT) columnReference')'
                          binaryOperator
                          operand

satConstraint ::= SATISFIES tablePseudonym
                            relation
                            tablePseudonym

joinSpecification ::=  joinType tableSpecification
                                [ AS tablePseudonym ]
```

```
                               [ joinCondition ]
                               [ joinSpecification ]

joinType ::= CROSS JOIN | INNER JOIN

joinCondition ::= ON condition

-------------identifier and datatypes-------------------------

valueList ::= ( embeddedCurryExpression | literalList )
               {',' ( embeddedCurryExpression | literalList )}

literalList ::= '(' literal  { ',' literal }  ')'

literal ::=   numericalLiteral
            | quotes alphaNumericalLiteral quotes
            | dateLiteral
            | booleanLiteral
            | embeddedCurryExpression
            | NULL

numericalLiteral ::=  integerExpression
                     |floatExpression

integerExpression ::= [ - ] digit { digit }

floatExpression := [ - ] digit { digit } '.' digit { digit }

alphaNumericalLiteral ::=  character { character }
character ::= digit | letter

dateLiteral ::=  year ':' month ':' day ':'
                 hours ':' minutes ':' seconds

month ::= digit digit
day ::= digit digit
hours ::= digit digit
minutes ::= digit digit
seconds ::= digit digit
year ::= digit digit digit digit

booleanLiteral ::= TRUE | FALSE

embeddedCurryExpression ::= '{' curryExpression '}'

pattern ::= ( character | specialCharacter )
             {( character | specialCharacter )}
specialCharacter ::= '%' | '_'

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter ::= (a...z) | (A...Z)

tableIdentifier ::= tablePseudonym | tableName
columnName ::= letter [alphanumericalLiteral]
```

```
tableName ::= letter [alphanumericalLiteral]
tablePseudonym ::= letter
relation ::= letter [[alphanumericalLiteral] | '_' ]
quotes ::= ('"'|''')
```

Appendix D

Examples

This section shows some exemplary SQL queries and their translation. the two `Select` statements demonstrate some of the new features of the CDBI library and the abstraction of foreign keys by the use of relations. Furthermore, we give less complex examples of an `Insert` and an `Update` statement which also demonstrate the usage of embedded Curry expressions. For readability reasons the layout of the translation is modified.

The following request selects the name of students ordered alphabetically, the average of all points achieved in any exam (given in table result) and the number of exams the student has participated in:

```
example1 :: IO (SQLResult [(String, Float, Int)])
example1 = ``sql Select s.name, Avg(r.Points), Count(r.Points)
                 From Student As s Inner Join Result As r
                                   On Satisfies s has_a r
                  Group By s.name
                   Having Count(r.Points) > 1 Order By s.name;''
```

This query will be translated to:

```
example1 :: IO (SQLResult [(String, Float, Int)])
example1 =
 runWithDB
   "Uni.db"
   (getColumnTriple
      []
      [TripleCS
        All
        (tripleCol
           (singleCol studentNameColDesc 0 none)
           (singleCol resultPointsColDesc 0 (avg All))
           (singleCol resultPointsColDesc 0 (count All)))
        (TC studentTable
            0
            (Just
```

```
          (innerJoin
             (equal (colNum studentColumnKey 0)
                    (colNum resultColumnStudentTakingKey 0))
          ,TC resultTable 0 Nothing)))
    (Criteria None
              (Just (groupBy
                      (colNum studentColumnName 0)
                      (having
                         (countCol
                           All
                           (colNum resultColumnPoints 0)
                           (int 1)
                           greaterThan))))))]
   [ascOrder (colNum studentColumnName 0)]
   Nothing)
```

The second request joins all names and first names of students and lecturers ordered by their first name and returns the first three rows:

```
example2 :: IO( SQLResult [(String, String)])
example2 = ``sql Select Distinct Firstname, Name From Student
                 Union
                 Select Distinct Firstname, Name From Lecturer
                 Order By Firstname Asc
                 Limit 3;''
```

It is translated to:

```
example2 :: IO( SQLResult [(String, String)])
example2 =
 runWithDB
   "Uni.db"
   (getColumnTuple
     [Union]
     [TupleCS
        Distinct
        (tupleCol
          (singleCol studentFirstnameColDesc 0 none)
          (singleCol studentNameColDesc 0 none))
        (TC studentTable 0 Nothing)
        (Criteria None Nothing)
      ,TupleCS
        Distinct
        (tupleCol
          (singleCol lecturerFirstnameColDesc 0 none)
          (singleCol lecturerNameColDesc 0 none))
        (TC lecturerTable 0 Nothing)
        (Criteria None Nothing)]
     [ascOrder (colNum lecturerColumnFirstname 0)]
     (Just 3))
```

The statement below inserts two entities into the database,the first one given as embedded Curry expression, the second one given as list of values.

```
insertEx :: Student -> IO (SQLResult ())
insertEx student1 =
 ``sql Insert Into Student
         Values
           {student1}
           ,(8, 6828, "Julia", "Krone", "julia@mail.de", 26);''

insertEx :: Student -> IO (SQLResult ())
insertEx student1 =
   runWithDB
     "Uni.db"
     (saveMultipleEntries  [student1
                            ,Student (StudentID 42)
                                     6828
                                     "Julia"
                                     "Krone"
                                     "julia@mail.de"
                                     (Just 26)]
                           studentDescription )
```

The last example updates the column "Age" in the `Student` table dependent on the value of an embedded Curry expression.

```
updateEx :: Int -> Int -> IO (SQLResult ())
updateEx x y  =
 ``sql Update Student Set Age = 20 Where Age < {1+x+6*y};''

updateEx :: Int -> Int -> IO (SQLResult ())
updateEx x y  =
  runWithDB
    "Uni.db"
    (updateEntries studentDescription
                   [colVal studentColumnAge (int 20)]
                   (lessThan (colNum studentColumnAge 0)
                             (int (1+x+6*y))))
```

Bibliography

[1] B. Brassel, M. Hanus, and M. Müller. High-level database programming in curry. In *Proc. of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

[2] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[3] P. P.-S. Chen. The entity-relationship-model - towards a unified view of data. volume 1, pages 9–36, 1976.

[4] O. Chitil. Pretty Printing with Delimited Continuations. Technical Report 4-06, University of Kent, June 2006.

[5] E. Codd. A relational model of data for large shared data banks. volume 13, pages 377–387, 1970.

[6] C. Date. *SQL and Relational Theory: How to write Accurate SQL Code.* O'Reilly, 2 edition, 2012.

[7] N. F. Fischer, K. C. Cyton, and R. J. LeBlanc Jr. *Crafting a Compiler.* Pearson, 2010.

[8] S. Fischer. A functional logic database library. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 54–59, New York, NY, USA, 2005. ACM Press.

[9] M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

[10] M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.

[11] M. Hanus and S. Antoy. Set functions for functional logic programming. In *Proc. 11th Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82, 2009.

[12] M. Hanus and C. Prehofer. A needed narrowing strategy. volume 47, pages 776–822, 2000.

[13] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[14] K. Muneeswaran. *Complier Design*. Oxford University Press, 2013.

[15] J. P. Sikorra. Foreign Code Integration in Curry. Bachelorthesis, March 2014.

[16] M. Tallarek. Implementierung einer Datenbank-Schnittstelle für Curry. Bachelorthesis, 2014.

[17] R. F. van der Lans. *Introduccion to SQL:mastering the relational database language*. Addison-Wesley, 4 edition, 2007.