Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Implementierung einer Datenbank-Schnittstelle für Curry

Mike Tallarek

Sommersemester 2014

Betreut durch Prof. Dr. Michael Hanus
und M. Sc. Björn Peemöller

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1. Einleitung

Für die Entwicklung von Software ist der Einsatz von Datenbanken häufig notwendig. Programmiersprachen sollten demnach Möglichkeiten bieten, mit Datenbanken kommunizieren zu können. Es gibt dabei verschiedene Varianten solch eine Kommunikation anzubieten, wobei vor allem die Ebene der Abstraktion zu bedenken ist. So kann eine Kommunikation auf niedriger Abstraktionsebene angeboten werden, indem SQL-Befehle direkt als String an eine Datenbank übergeben und dort ausgeführt werden. Der Vorteil einer solchen Umsetzung ist es, dass der volle Funktionsumfang von SQL zur Verfügung steht. Allerdings bieten solche Varianten keinen Schutz gegen Sicherheitslücken wie zum Beispiel *SQL injections*[1] und Typsicherheit ist nicht gegeben. Auf der anderen Seite gibt es Varianten mit sehr hoher Abstraktionsebene. Hier ist die direkte Verbindung zur Datenbank nicht mehr sichtbar und SQL-Befehle an die Datenbank werden in die Programmiersprache integriert und somit stark abstrahiert. Das Abfragen und Speichern von Daten auf Datenbanken wird streng typisiert. Diese Variante liefert Sicherheit und vereinfacht die Programmierung für Nutzer. Ein Nachteil von vielen Schnittstellen mit hoher Abstraktionsebene ist es, dass die Mächtigkeit im Gegensatz zu Schnittstellen mit niedriger Abstraktionsebene sehr viel geringer ausfällt.

Für die logisch-funktionale Programmiersprache Curry[2] gibt es bereits ein Modul namens `Database` [BHM08], welches eine Schnittstelle mit hoher Abstraktionsebene zur Verfügung stellt und Typsicherheit bietet. Mit dieser Schnittstelle ist es möglich, Datentypen, die aus einem ER-Modell automatisch generiert werden, direkt auszulesen und abzuspeichern. Es ist hier allerdings nicht möglich, komplexe Anfragen zu formulieren, die direkt auf der Datenbank ausgeführt werden. Um komplexe Suchen nach Daten durchzuführen, müssen zunächst alle Einträge der entsprechenden SQL-Tabelle gelesen werden, welche der Programmierer als Rückgabewert erhält. Dann erst können die Einträge nach bestimmten Kriterien gefiltert werden. Bei sehr komplexen Suchen kommt es so zu einem hohen Programmieraufwand und es werden mehr Datensätze als eigentlich notwendig aus der Datenbank gelesen, was durch komplexe Anfragen direkt auf der Datenbank verhindert werden würde.

Die Zielsetzung dieser Arbeit ist es nun, eine Datenbankschnittstelle für Curry zu entwickeln, die auf einer ähnlich hohen Abstraktionsebene wie die bisherige Schnittstelle arbeitet, aber zusätzlich komplexe Anfragen anbietet, die direkt auf der Datenbank ausgeführt werden. Die zu erstellende Schnittstelle soll auch in der Lage sein, mit aus einem ER-Modell automatisch generierten Datentypen zu arbeiten. Das automatische Generieren dieser Datentypen ist ebenso Teil dieser Arbeit.

[1] `http://en.wikipedia.org/wiki/SQL_injection`
[2] `http://www-ps.informatik.uni-kiel.de/currywiki/start`

Es werden zunächst alle technischen und theoretischen Grundlagen, die für diese Schnittstelle notwendig sind, vorgestellt. Daraufhin wird der Entwurf und dann die Implementierung erläutert. Es folgt am Schluss ein Fazit, welches erläutert, ob alle Ziele zufriedenstellend erfüllt worden sind und was in der Zukunft an der Schnittstelle noch erweitert werden könnte.

2. Technische und theoretische Grundlagen

2.1. Die Programmiersprache Curry

Curry ist eine universelle Programmiersprache, deren Ziel es ist, die wichtigsten deklarativen Programmierparadigmen, die funktionale Programmierung (verschachtelte Ausdrücke, Lazy Evaluation, Funktionen höherer Ordnung) und die logische Programmierung (logische Variablen, partielle Datenstrukturen, eingebaute Suchverfahren), zu vereinen [He12].

Ein Curry-Programm auszuführen bedeutet, dass ein Ausdruck so lange vereinfacht wird, bis ein Wert oder eine Lösung berechnet wurde. Um zu entscheiden, ob ein Ausdruck bereits zu einem Wert vereinfacht wurde, muss zwischen Konstruktoren und Funktionen beziehungsweise Operationen unterschieden werden. Datentypen in Curry werden mithilfe von Konstruktoren deklariert. Besteht ein Ausdruck nur noch aus Konstruktoren und den zugehörigen Werten (welche auch alle komplett ausgewertet wurden), dann wurde er komplett ausgewertet.

In Curry bestehen Programme also aus einer Menge von Datentyp- und Funktionsdeklarationen. Die Datentypdeklarationen definieren die Wertebereiche des Programms und die Funktionsdeklarationen die Operationen auf diesen Wertebereichen.

Curry ist eine stark getypte Programmiersprache, die Polymorphie unterstützt. Jedes Objekt besitzt einen eindeutigen Typen und beim Programmieren mit Curry können die Typen von Variablen und Funktionen in den meisten Fällen weggelassen werden, da diese durch einen Typ-Inferenz-Mechanismus rekonstruiert werden können.

Datentypen werden in Curry wie folgt deklariert:

$$\texttt{data A}\ a_1, ..., a_n = C_1\, v_{1_1}, ..., v_{1_{k_1}} \mid ... \mid C_m\, v_{m_1}, ..., v_{m_{k_m}}$$

Der Name des Datentyps ist A, $a_1, ..., a_n$ sind Typvariablen, $C_1, ..., C_m$ sind Konstruktoren und $v_{1_1}, ..., v_{m_{k_m}}$ sind Typ-Ausdrücke. Dabei dürfen die definierten Typvariablen in den Typausdrücken der Konstruktoren verwendet werden. Jeder Konstruktor C_m hat den Typ

$$v_{m_1} \to ... \to v_{m_{k_m}} \to \texttt{A}\ a_1, ..., a_n$$

Es gilt für alle $k_m >= 0$, wobei der Konstruktor C_m keine Argumente besitzt, wenn $k_m = 0$ gilt. So kann man zum Beispiel den polymorphen Datentyp `Either` definieren:

```
data Either a b = Left a | Right b
```

Dieser Datentyp wird entweder durch den Konstruktor `Left` und einen Wert vom Typ `a` oder durch den Konstruktor `Right` und einen Wert vom Typ `b` gebildet.

Datentypen können auch rekursiv aufgebaut werden. Das nächste Beispiel definiert polymorphe Binärbaume, die Werte zweier verschiedener Datentypen als Blätter besitzen können:

```
data Tree a b = Leaf (Either a b)
              | Node (Tree a b) (Tree a b)
```

Dieser Baum ist entweder ein Blatt vom Typ `Either a b` oder ein Knotenpunkt, der zwei weitere Bäume mit den Typen `Tree a b` besitzt. Ein Wert dieses Datentyps könnte zum Beispiel so aussehen:

```
myTree = Node (Node (Leaf (Left "A String"))
                    (Leaf (Right 23)))
              (Leaf (Left "Another String"))
```

Ohne explizit einen genauen Datentyp für `myTree` anzugeben, erkennt Curry durch seinen Typ-Inferenz-Mechanismus, dass `myTree` vom Typ `Tree String Int` ist.

Funktionen in Curry werden durch eine Typ-Signatur, die allerdings nicht zwingend notwendig ist, und eine Liste von Gleichungen, die die Funktion genau beschreiben, definiert. Typ-Signaturen sind wie folgt aufgebaut:

`f` :: $a_1 \rightarrow ... \rightarrow a_n$

f ist der Name der Funktion und $a_1, ..., a_n$ sind Datentypen. Wird f auf Werte der Typen $a_1, ..., a_{n-1}$ angewendet, dann liefert sie einen Wert vom Typ a_n als Ergebnis. Sie kann allerdings auch auf weniger Werte angewendet werden. Wird sie auf Werte der Typen $a_1, ..., a_m$ angewendet, wobei $m < n$ gilt, dann liefert f eine Funktion vom Typ $a_{m+1} \rightarrow ... \rightarrow a_n$. Diese Funktion agiert wie f, es sind nur die ersten m Werte schon festgelegt.

Ein Beispiel, dass dieses Prinzip zeigt:

```
add :: Int -> Int -> Int
add a b = a + b
addThree :: Int -> Int
addThree = add 3
```

Die Funktion `add` liefert einen Wert vom Typ `Int`, wenn man sie mit zwei `Int`-Werten ausführt. Führt man sie mit nur einem `Int`-Wert aus, dann liefert sie eine Funktion vom Typ `Int -> Int`. Die Funktion `addThree` ist definiert durch `add 3`, es ist also eine Funktion, die einen `Int`-Wert erwartet und diesen mit 3 addiert. Dieses Prinzip nennt man *Partial Application*.

Ein weiteres wichtiges Prinzip für Funktionen ist *Pattern Matching*. Funktionen können so implementiert werden, dass sie für unterschiedliche *Patterns* andere Definitionen haben. So kann für Werte, die einer Funktion übergeben werden, überprüft werden, ob sie einem bestimmten Muster entsprechen und dementsprechend wird ausgewählt, welche Gleichung für die Auswertung des Funktionsaufrufs in Frage kommt.

Eine Beispielsfunktion, die Pattern Maching nutzt:

```
isLeft :: Either _ _ -> Bool
isLeft (Left  _) = True
isLeft (Right _) = False
```

Die Funktion `isLeft` überprüft, ob ein Wert vom Typ `Either a b` mit dem Konstruktor `Left` erstellt wurde. Dies wird durch Pattern Matching überprüft. Stimmt ein Wert mit dem Pattern `(Left _)` überein, dann wird `True` zurückgegeben. Stimmt ein Wert allerdings mit `(Right _)` überein, dann wird `False` zurückgegeben. Das Zeichen `_` kann genutzt werden, wenn das Pattern an dieser Stelle beliebig ist. Es wird dadurch allerdings keine Variable gebunden. Auch in der Typdeklaration kann `_` genutzt werden, da die Typen von `Either` nicht gebraucht werden.

Pattern Matching ist auch nützlich, wenn man bestimmte Teile eines Datentyps in der Funktion nutzen möchte, was durch das folgende Beispiel demonstriert wird:

```
leftNode :: Tree a b -> Tree a b
leftNode (Leaf   x) = (Leaf x)
leftNode (Node x _) = x
```

Die Funktion `leftNode` liefert den linken Teilbaum eines Baumes, außer wenn es sich bei dem Baum um ein Blatt handelt. In dem Fall wird das Blatt zurückgegeben.

Beim Pattern Matching sind im Gegensatz zu Haskell auch Ausdrücke erlaubt, in denen Funktionen vorkommen. Da Curry eine logische Programmiersprache ist, wird außerdem nicht die erste Funktionsdefinition genutzt, welche passt, wie es zum Beispiel bei Haskell der Fall ist. Bei Haskell wird von oben nach unten die erste Funktionsdefinition gesucht, die passt, welche dann ausgeführt wird. Curry sucht bei Ausführungen nach allen möglichen Lösungen, weswegen im Falle von mehreren passenden Funktionsdefinitionen auch alle für eine jeweils andere Lösung genutzt werden. Aus diesem Grund kann man bei `isLeft` auch nicht

```
isLeft (Left _) = True
isLeft _        = False
```

geschrieben werden, wie es bei Haskell möglich wäre. Bei Curry wäre `isLeft` nun eine nicht-deterministische Funktion, da sie für Werte, die dem Pattern `(Left _)` entsprechen, `True` und `False` ergeben kann.

An diesem Beispiel sieht man einen ersten Unterschied von Curry zu Haskell. Curry ist, wie man bisher gesehen hat, von der Syntax und Semantik her sehr ähnlich zu Haskell. Es gibt aber dennoch – gerade da Curry neben Aspekten der funktionalen Programmierung auch logische Aspekte implementiert – viele Unterschiede. Auf den logischen Teil von Curry wird im Folgenden eingegangen.

In Curry sind nicht-deterministische Funktionen erlaubt, wodurch Ausdrücke mehrere Ergebnisse haben können. Curry sucht solange nach möglichen Vereinfachungen von Ausdrücken, bis es keine mehr findet. Curry hat für logische Berechnungen den Datentyp `Success`, welcher angibt, ob ein logischer Ausdruck eine Lösung besitzt oder nicht, und logische Variablen eingeführt. Logische Variablen dürfen nur auf der rechten Seite eines Ausdrucks auftauchen und sind demnach zunächst nicht gebunden. Sie werden durch den Ausdruck `where xs free` deklariert, wobei `xs` eine Liste von bisher ungebundenen Variablen ist, die von Kommas getrennt werden.

```
oneLeft :: [Either a b] -> a
oneLeft list | xs ++ ((Left x) : ys) =:= list = x
                where xs, x, ys free
```

Diese Funktion nimmt eine Liste vom Typ `[Either a b]` und gibt einen Wert vom Typ `a` zurück, welcher in der Liste in einem `Left`-Konstruktor als Wert vorkam. Der Operator `=:=` testet, ob es eine Belegung der deklarierten freien Variablen gibt, sodass der linke Term gleich dem rechten Term ist. Ist eine solche Belegung vorhanden, dann gibt `=:=` den Wert `success` zurück und der Rückgabewert von `oneLeft` ist `x`. Da es durchaus mehrere Belegungen geben kann, für die das gilt, ist diese Funktion nicht-deterministisch und Curry kann unter Umständen mehrere Lösungen finden. Ist die Liste leer oder es sind keine Einträge der Art `(Left x)` vorhanden, dann hat die Funktion keine Lösung.

Ausdrücke mit logischen Variablen werden in Curry je nach Art des Ausdrucks entweder durch *Residuation* oder *Narrowing* ausgewertet [Han13].

Funkionen in Curry, die Input/Output nutzen, haben als Ergebnis eine IO-Aktion. Eine IO-Aktion enthält einen Rückgabewert, welcher auch der `Unit type`, der keine Information enthält, sein kann. So gibt es zum Beispiel die Funktion `putStr`, die einen String zum Standard-Output schreibt, mit folgendem Typ:

```
putStr :: String -> IO ()
```

`IO ()` ist eine IO-Aktion mit `Unit type` als Rückgabewert. Die Funktion `getLine` liest eine Zeile Input und gibt diese als Wert zurück. Sie hat folgenden Typ:

```
getLine :: IO String
```

IO-Aktionen können durch `do`-Blöcke kombiniert werden. Durch ← wird der Rückgabewert einer IO-Aktion einer Variable zugewiesen. Eine Funktion, die zwei Zeilen einliest und diese dann kombiniert wieder ausgibt, könnte so wie folgt aussehen:

```
readAndPut :: IO ()
readAndPut = do line <- getLine
                line2 <- getLine
                putStr (line ++ line2)
```

Für die Programmiersprache Curry gibt es mittlerweile verschiedene Implementierungen. Durch `PAKCS`[1], das gemeinsam von der Portland State University, der RWTH Aachen und der Christian-Albrechts-Universität zu Kiel entwickelt wird, werden Curry-Programme in Prolog-Programme übersetzt. KICS2[2], welches an der Christian-Albrechts-Universität zu Kiel entwickelt wird, übersetzt Curry-Programme hingegen in Haskell-Programme. Die Datenbankschnittstelle dieser Arbeit wurde mit PAKCS entwickelt.

[1] `http://www.informatik.uni-kiel.de/~pakcs/`
[2] `http://www-ps.informatik.uni-kiel.de/kics2/`

2.2. Entity-Relationship-Diagramme

Entity-Relationship-Modelle (*ER-Modelle*) [Che76] werden genutzt, um Datenbankschemata in ihrer Struktur und ihren Einschränkungen zu modellieren. Die graphische Darstellung einen ER-Modells wird *Entity-Relationship-Diagramm* genannt. In ER-Modellen werden Entitäten definiert, die Attribute und Beziehungen zu anderen Entitäten besitzen. Beziehungen unterliegen dabei Einschränkungen in der Form von Kardinalitäten.

ER-Modelle können in Curry durch einen Datentyp, der im Folgenden vorgestellt wird, modelliert werden [BHM08]. Dieser sieht folgendermaßen aus:

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

Ein Wert vom Typ `ERD` (Entity-Relationship-Diagramm) hat einen Namen, eine Liste von Entitäten und eine Liste von Beziehungen. Entitäten sind dabei wie folgt definiert:

```
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)
               | UserDefined String (Maybe String)
               | KeyDom String
```

Eine Entität hat einen Namen und eine Liste von Attributen. Attribute haben auch einen Namen. Sie besitzen einen bestimmten Typ und man kann definieren, ob sie den Wert `NULL` annehmen dürfen oder nicht. Zusätzlich sind sie entweder ein Primärschlüssel, einzigartig oder kein Schlüssel. Für alle Attribute, außer die vom Typ `Key`, welcher später für Fremdschlüssel genutzt wird, kann ein Standard-Wert festgelegt werden. Das geschieht durch die `Maybe`-Ausdrücke im `Domain`-Datentyp. Auch vom Nutzer definierte Datentypen sind möglich.

Weiterhin sind Beziehungen wie folgt definiert:

```
data Relationship = Relationship String [REnd]
data REnd         = REnd String String Cardinality
data Cardinality  = Exactly Int | Range Int (Maybe Int)
```

Eine Beziehung hat einen Namen und eine Liste von Verbindungen zu Entitäten (`REnd`). Diese Verbindungen besitzen den Namen der entsprechenden Entität, den Namen der Rolle in dieser Beziehung und eine Angabe über die Kardinalität. Kardinalitäten können entweder als genauer Wert oder als ein Bereich zwischen zwei `Int`-Werten angegeben werden, wobei `Nothing` dafür steht, dass die Kardinalität nach oben hin unbeschränkt ist. Statt `Nothing` kann auch `Infinite` geschrieben werden. Wird im Folgenden von einem `ERD` gesprochen, dann ist damit eine Instanz von diesem Datentyp gemeint.

In Abbildung 2.1 ist ein ER-Diagramm zu sehen, das die Datenbank einer Universität darstellen könnte. Dieses ER-Diagramm wird in dieser Arbeit fortan häufig als Beispiel herbei genommen. Die Entität `Student` würde zum Beispiel so aussehen:

```
Entity "Student"
  [Attribute "MatNum" (IntDom Nothing) Unique False,
   Attribute "Name" (StringDom Nothing) NoKey False,
   Attribute "FirstName" (StringDom Nothing) NoKey False,
   Attribute "Email" (StringDom Nothing) Unique False,
   Attribute "Age" (IntDom Nothing) NoKey True]
```

Die Beziehung zwischen `Student` und `Lecture`:

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

Das gesamte `ERD` ist im Anhang (A.1) zu sehen.

2.3. Das Tool `erd2curry`

Das Tool `erd2curry`, ein Teil des Frameworks Spicey[3], welches an der Christian-Albrechts-Universität zu Kiel entwickelt wird, übersetzt ein `ERD` in ein verändertes `ERD`, wodurch es ein relationales Datenbankschema darstellt, und erzeugt Daten wie zum Beispiel die Datentypen für die Entitäten, damit dieses ERD mit dem Modul `Database` nutzbar ist (siehe [BHM08]). Für diese Arbeit ist nur das veränderte ERD relevant.

Um ein ER-Modell mit einer Datenbank zu nutzen, muss man dieses zunächst in ein relationales Datenbankmodell (siehe [Cod70]) umwandeln, da in Datenbanken lediglich Tabellen gespeichert werden können. Entitäten und Beziehungen müssen also in eine Form gebracht werden, sodass sie man sie als Tabellen darstellen kann.

Eine naive Variante wäre es, alle Entitäten so zu übernehmen, wie sie bisher sind, und ihre Namen als Tabellennamen zu nehmen. Ihre Attribute bilden dann die Spalten. Beziehungen könnten ebenfalls ihre Namen als Tabellennamen nehmen und ihre Spalten wären die Fremdschlüssel der teilnehmenden Entitäten. Bei dieser Variante würden allerdings sehr viele Tabellen erstellt werden, was durch eine andere Variante umgangen werden kann.

Eine bessere Variante ist es, Beziehungen, deren Kardinalitäten es erlauben, als Fremdschlüssel in einer der teilnehmenden Entitäten einzubauen. Die Relation `ExamPlace` (siehe Abbildung 2.1) kann zum Beispiel implementiert werden, indem ein Fremdschlüssel von der Entität `Place` als Attribut in der Entität `Exam` eingefügt wird. Bei manchen Beziehungen ist diese Art der Implementierung allerdings nicht möglich. Bei einer `(0..n)`-zu-`(0..n)` Beziehung, wie zum Beispiel `Participation`, muss eine neue Entität für die Beziehung eingeführt werden.

[3] `http://www.informatik.uni-kiel.de/~pakcs/spicey/`

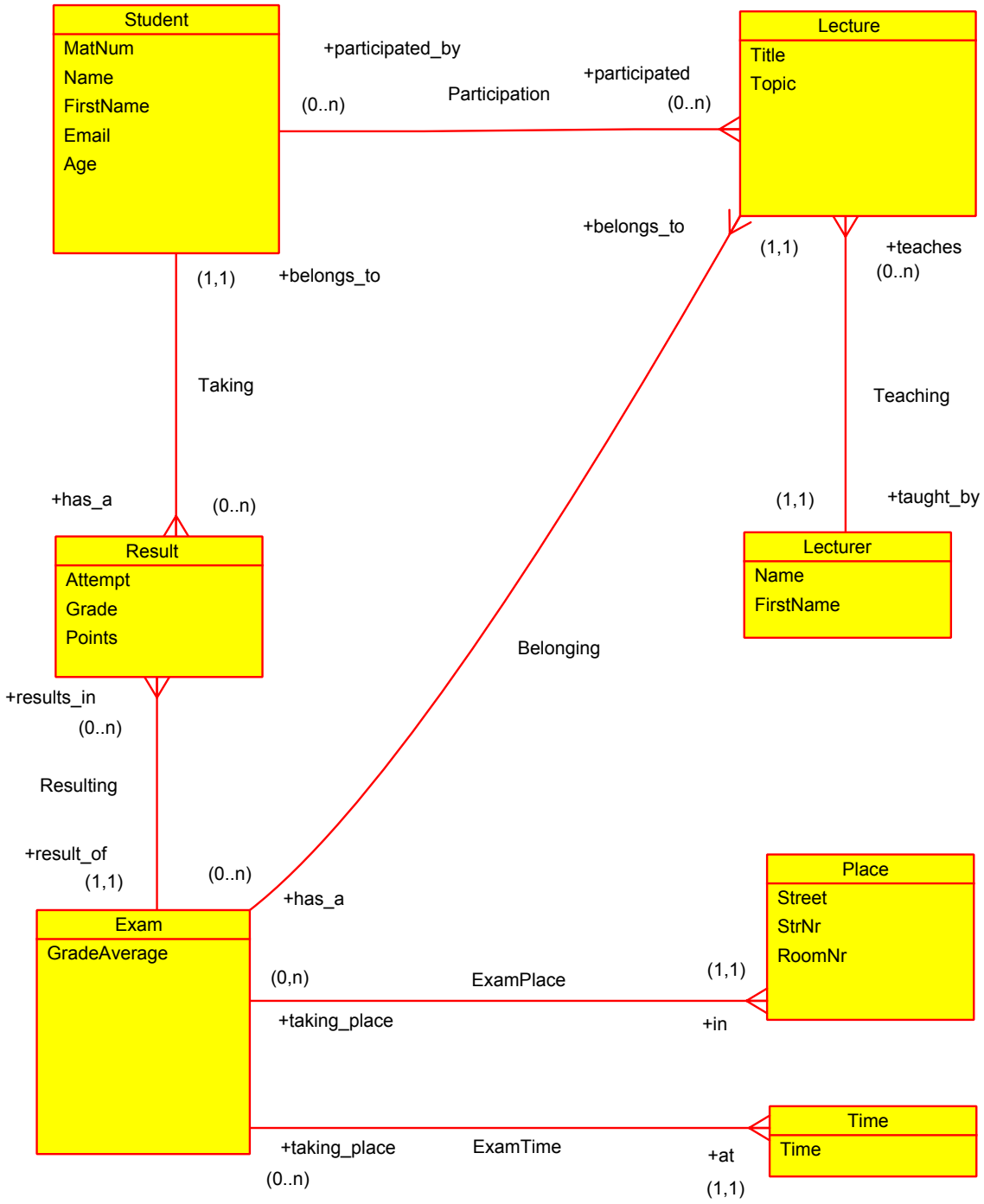

Abbildung 2.1.: ER-Diagramm der Datenbank einer Universität

Die genauen Übersetzungen der Beziehungen sind in der folgenden Auflistung genannt. Die linke Seite ist jeweils die Kardinalität der Entität A und die rechte Seite die Kardinalität der Entität B:

(0,1) - (1,1) Fremdschlüssel von B zur Entität A mit Unique-Bedingung, `NULL` nicht erlaubt

(0,1) - (0,1) Fremdschlüssel von B zur Entität A mit Unique-Bedingung, `NULL` erlaubt

(0,1) - (min,max$_{>1}$) Fremdschlüssel von A zur Entität B, `NULL` erlaubt

(1,1) - (0,max$_{>1}$) Fremdschlüssel von A zur Entität B, `NULL` nicht erlaubt

(min,max$_{>1}$) - (min,max$_{>1}$) Es muss eine neue Entität eingeführt werden, die einen Fremdschlüssel von A und einen Fremdschlüssel von B besitzt

Das Tool `erd2curry` übernimmt diese Übersetzung. Jede nicht durch diese Übersetzung neu entstandene Entität erhält dabei zudem einen Primärschlüssel namens `Key`. Waren zuvor Attribute als `PKey` gekennzeichnet, dann werden sie zu `Unique` umgewandelt. In dem Beispiel aus Abbildung 2.1 wird nach der Übersetzung eine Entität hinzugefügt:

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

Die Entität `Lecture` wurde wie folgt erweitert:

```
Entity "Lecture"
  [Attribute "Key" (IntDom Nothing) PKey False,
   Attribute "Title" (StringDom Nothing) NoKey False,
   Attribute "Topic" (StringDom Nothing) NoKey True,
   Attribute "LecturerTeachingKey"
             (KeyDom "Lecturer") NoKey False]
```

Das gesamte übersetzte `ERD` ist im Anhang (A.2) zu finden.

2.4. SQL

SQL ist eine Datenbanksprache, mit der relationale Datenbanksystem verwaltet werden können. Da SQL allgemein schon sehr bekannt ist, sollen hier nur noch einmal die wichtigsten Befehle erläutert werden, die für die Schnittstelle relevant sind.

- **Tabellen erstellen**

  ```
  CREATE TABLE table_name (column_name datatype constraint, ...);
  ```

Der `CREATE TABLE`-Befehl erstellt eine Tabelle in der aktuellen Datenbank. Es wird der Tabellenname gesetzt und dann in Klammern aufgezählt, welche Spalten die Tabelle haben soll. Spaltendeklarationen werden durch Kommas getrennt. In ihnen werden der Name der Spalte, der Datentyp und Einschränkungen gesetzt. Bei den Einschränkungen kann zum Beispiel deklariert werden, ob eine Spalte mit NULL-Werten beschrieben werden kann oder ob sie einzigartig, ein Schlüssel oder ein Fremdschlüssel ist.

- **Werte einfügen**

```
INSERT INTO table_name VALUES (value1, value2, ...);
```

Der `INSERT INTO`-Befehl fügt eine Reihe zu einer Tabelle hinzu. Es muss der Tabellenname angegeben werden und in Klammern die Werte, die die Reihe beinhalten soll. Dabei müssen die Werte in der Reihenfolge angeben werden, wie sie in der Reihe auftauchen sollen.

- **Reihen aktualisieren**

```
UPDATE table_name SET column1=value1,
                      column2=value2,... WHERE-Clause;
```

Der `UPDATE`-Befehle aktualisiert Reihen einer Tabelle. Es muss zunächst der Tabellenname angeben werden. Danach die Spalten, die aktualisiert werden sollen mit den Werten, die in sie geschrieben werden sollen. Danach folgt eine `WHERE`-Klausel. In `WHERE`-Klauseln werden Bedingungen gesetzt, die definieren, welche Reihen von dem Befehl betroffen sind. So werden nur die Reihen aktualisiert, die diese Bedingungen erfüllen. Die Beschreibung von `WHERE`-Klauseln folgt nach den Beschreibungen der Befehle.

- **Reihen löschen**

```
DELETE FROM table_name WHERE-CLAUSE;
```

Der `DELETE FROM`-Befehl löscht Reihen aus Tabellen. Welche Reihen gelöscht werden wird wie beim `UPDATE`-Befehl durch die `WHERE`-Klausel definiert.

- **Reihen auslesen**

```
SELECT * or column1, column2,... FROM table WHERE-CLAUSE OPTIONS;
```

Der `SELECT`-Befehle liest Reihen aus einer gewünschten Tabelle aus. Entweder werden alle Spalten ausgegeben, indem nach `SELECT` der Stern gesetzt wird oder es wird definiert, welche Spalten ausgeben werden sollen. Es sind hier auch noch weitere Möglichkeiten gegeben, die für die Schnittstelle aber nicht weiter relevant sind. Bei der gewünschten Tabelle kann ein einfacher Tabellenname angegeben werden, es ist aber auch möglich, dass mehrere Tabellen mit `JOIN`-Operatoren miteinander vereint werden. So vereint zum Beispiel `CROSS JOIN` Tabellen so miteinander, dass das kartesische Produkt gebildet wird. Jede Reihe der ersten Tabelle wird mit jeder

Reihe der zweiten Tabelle kombiniert. Dann wird mit einer `WHERE`-Klausel definiert, welche Bedingungen die Reihen erfüllen müssen, die ausgelesen werden sollen. Am Ende können noch Optionen gesetzt werden, wie zum Beispiel die Reihenfolge, in der die Reihen ausgegeben werden sollen. So können Reihen nach bestimmten Spalten aufsteigend oder absteigend sortiert werden. Es gibt noch weitere komplexere Möglichkeiten für diese Anfrage, welche für die Schnittstelle allerdings nicht relevant sind.

- **Tabellen löschen**

  ```
  DROP TABLE table_name;
  ```

  Löscht die angegebene Tabelle komplett aus der Datenbank.

Die `WHERE`-Klauseln, die in manchen Befehlen enthalten sind, werden über drei Arten von Operatoren aufgebaut.

- **Vergleiche**

  Es können als Bedingung Konstanten mit Spalten (beziehungsweise den Werten aus Spalten) und Spalten mit anderen Spalten verglichen werden. Theoretisch ist es auch möglich, Konstanten mit anderen Konstanten zu vergleichen, dies hat aber praktisch keinen Nutzen. Mehrere Vergleichsoperatoren sind nutzbar, wie zum Beispiel `=` (Gleich), `<` (Kleiner als) oder `BETWEEN`, bei dem überprüft wird, ob ein Werte/eine Spalte zwischen zwei anderen Werten/Spalten liegt.

- **Operatoren auf Konditionen**

  Konditionen, wie zum Beispiel ein Vergleich, können durch Operatoren miteinander verbunden und negiert werden. So gibt es die Operatoren `AND`, `OR` und `NOT`. Konditionen, die durch Anwendung dieser Operatoren entstehen, können wiederum durch weitere Operatoren verbunden/negiert werden.

- **Der `EXISTS`-Operator**

  Der `EXISTS`-Operator erhält eine SELECT-Anfrage. Wird bei dieser Anfrage mindestens eine Reihe ausgegeben, dann ist die Kondition erfüllt.

Zusätzlich sind noch sogenannte Transaktions-Befehle relevant für die Schnittstelle. Diese Befehle sehen bei verschiedenen SQL-Implementierungen allerdings häufig unterschiedlich aus. So kann man Transaktionen starten (bei SQLite `begin;`), abschließen (bei SQLite `commit;`) und zurücksetzen (bei SQLite `rollback;`). Transaktionen bestehen aus mehreren SQL-Befehlen, die durch das Starten und das Beenden der Transaktion eingeschlossen werden. Wird die Transaktion abgeschlossen (`commit;`), dann werden alle Änderungen durch die SQL-Befehle innerhalb der Transaktion permanent durchgeführt. Die zweite Variante ist es dann, dass alle Änderungen in der Transaktion zurückgesetzt werden (`rollback;`).

2.5. Die Datenbanksoftware `SQLite3`

SQLite3[4] ist eine Software, mit der man eine SQLite-Datenbank erstellen und verwalten kann. SQLite ist ein Datenbank-Management-System, das ohne Server läuft und nicht konfiguriert werden muss, damit es funktioniert. SQLite implementiert die meisten Funktionen des SQL-92 Standards [MS93].

SQLite3 ist sehr einfach zu nutzen. Nachdem SQLite3 installiert wurde, kann direkt eine neue Datenbank erstellt werden, indem SQLite3 mit dem gewünschten Namen der zu erstellenden Datenbank als Parameter gestartet wird.

```
sqlite3 Uni.db
```

Wird auf diese Weise einen SQLite3-Prozess gestartet, dann wird die Datenbank-Datei `Uni.db` erstellt und es kann sofort mit ihr gearbeitet werden. Gibt es die Datenbank-Datei `Uni.db` schon, dann kann mit dieser weitergearbeitet werden. Die gesamte Datenbank befindet sich in dieser einen Datei. Dem Prozess können dann SQL-Befehle übergeben werden und dieser führt sie auf der Datenbank aus. Bei Befehlen, die Ergebnisse liefern, werden diese vom Prozess in den Standard-Output geschrieben, sodass sie von anderen Programmen einfach gelesen werden können. Das Format der Ausgaben kann durch Optionen verändert werden, damit diese zum Beispiel für andere Programme leichter lesbar sind. So sorgt die Option `.mode line` dafür, dass für jede Spalte bei einer Ausgabe eine eigene Zeile genutzt wird und diese dabei aus dem Spaltennamen und dem Wert der Spalte getrennt durch ein Gleichheitszeichen besteht. Ohne diese Option sieht eine Ausgabe beispielsweise wie folgt aus:

```
sqlite> select * from Student;
1|23|Mustermann|Max|max@email.de|28
2|66|Arndt|Moritz|moritz@email.de|27
3|44|Ziegeldt|Stefan|stefan@email.de|20
```

Stellt man die Option ein, sieht eine Ausgabe so aus:

```
sqlite> .mode line
sqlite> select * from Student;
       Key = 1
    MatNum = 23
      Name = Mustermann
 FirstName = Max
     Email = max@email.de
       Age = 28

       Key = 2
    MatNum = 66
      Name = Arndt
 FirstName = Moritz
     Email = moritz@email.de
       Age = 27
```

[4] `http://www.sqlite.org/`

Die Option `.log stdout` sorgt dafür, dass Fehlermeldungen auch in den Standard-Output geschrieben werden, sodass andere Programme sie lesen können.

3. Entwurf der Schnittstelle

Dieses Kapitel befasst sich zunächst mit den funktionalen und nichtfunktionalen Anforderungen, die die Schnittstelle erfüllen soll, und daraufhin wird der gewünschte Aufbau der Schnittstelle dargestellt. Beim Aufbau werden die Aufgaben der Schnittstelle in verschiedene Komponenten beziehungsweise Schichten aufgeteilt und es wird erläutert, was diese leisten und wie sie miteinander interagieren. Als Name für die Schnittstelle wurde CDBI (Curry Database Interface) gewählt.

3.1. Anforderungen

3.1.1. Funktionale Anforderungen

Die Schnittstelle soll typsichere Funktionen implementieren, welche SQL-Befehle auf einer Datenbank ausführen, wobei auch komplexe Anfragen formuliert werden können. Zunächst soll lediglich die Verbindung zu SQLite-Datenbanken angeboten werden. Die Möglichkeit, die Schnittstelle um weitere Typen von Datenbanken zu erweitern, soll aber gegeben sein. Es soll zudem möglich sein, mit Entitäten, beschrieben in einem `ERD`, typsicher zu arbeiten. Die Datentypen und die Datenbank, welche dafür notwendig sind, sollen aus einem jeweiligen ERD automatisch erzeugt werden. Es sollen Funktionen bereitgestellt werden, mit denen Entitäten ausgelesen, gespeichert und aktualisiert werden können. Es muss dem Nutzer zudem möglich gemacht werden, dass er mehrere Entitäten zu einem neuen Datentyp kombinieren kann, welcher dann auch ausgelesen, gespeichert und aktualisiert werden kann.

3.1.2. Nichtfunktionale Anforderungen

Neben den funktionalen Anforderung soll die Schnittstelle auch nichtfunktionale Anforderungen erfüllen. So soll die Schnittstelle möglichst intuitiv nutzbar sein, wenn man sich mit SQL-Befehlen schon auskennt. Befehle in der Schnittstelle sollten also möglichst ähnlich aufgebaut sein wie Befehle in SQL.

Ebenso sollte im Allgemeinen die Syntax der Schnittstelle so simpel und übersichtlich wie möglich gehalten werden, so dass auch sehr komplexe Befehle noch übersichtlich verfasst werden können. Programme, die die Schnittstelle nutzen, sollen möglichst schnell verstanden und unter Umständen leicht erweitert werden können.

3.2. Komponenten

Die Aufgaben der Schnittstelle können gut in verschiedene Komponenten aufgeteilt werden. Drei Komponenten implementieren SQL-Befehle auf unterschiedlichem Abstraktionslevel, wobei die Komponenten mit höherem Abstraktionslevel die jeweils niedrigere als Basis nutzen. Diese drei Komponenten bilden als drei aufeinander aufbauende Schichten den Hauptteil der Schnittstelle. Eine vierte Komponente implementiert die Funktionalität, aus einem `ERD` alle notwendigen Daten und eine Datenbank zu erstellen, welche von der obersten Schicht der anderen drei Komponenten genutzt wird. Abbildung 3.1 stellt diesen Aufbau dar.

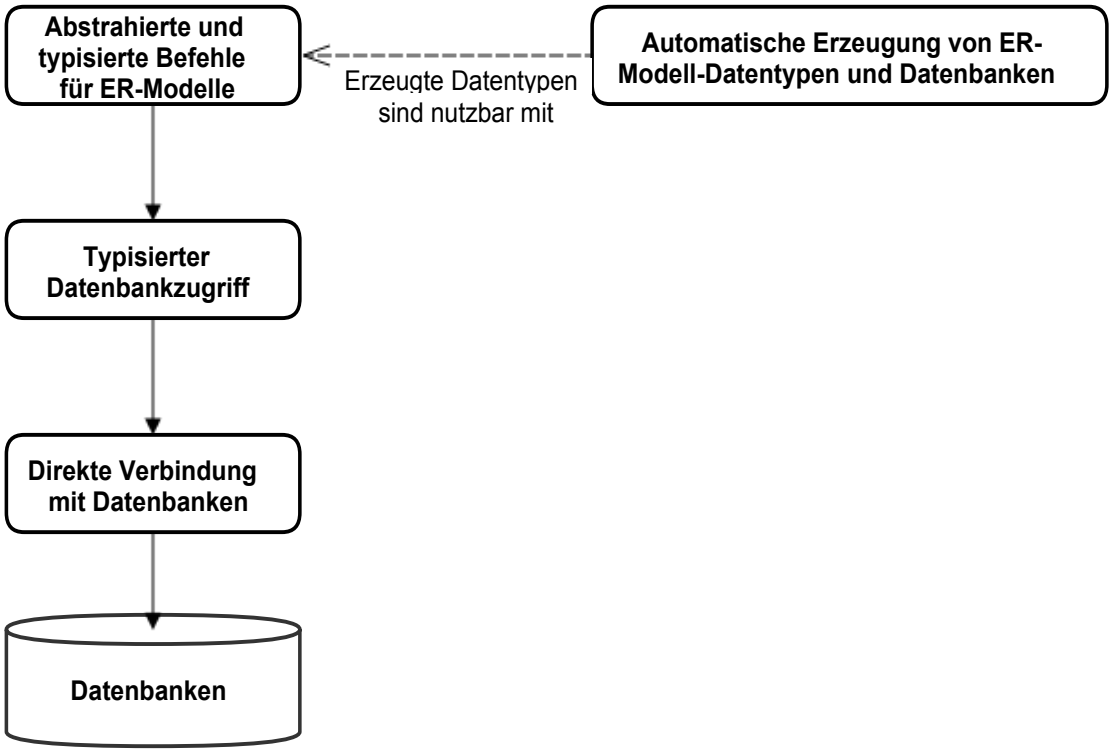

Abbildung 3.1.: Komponentendiagramm der Schnittstelle

Im Folgenden wird für jede Komponente erläutert, was sie genau an Funktionalität bieten soll und wie sie mit den anderen Komponenten interagiert.

3.2.1. Verbindung mit Datenbanken

Die erste Komponente ermöglicht es, dass eine Verbindung zu einer Datenbank aufgebaut werden kann. Durch diese Verbindung soll es möglich sein, SQL-Befehle in Form von

Strings auf Datenbanken auszuführen.

Zunächst wird lediglich eine Implementierung für SQLite-Datenbanken geschaffen. Die Komponente soll allerdings Verbindungen zu Datenbanken so abstrakt wie möglich behandeln, sodass die Implementierung von weiteren Arten von Datenbanken leicht möglich ist. Alle grundlegenden Funktionen dieser Komponente sollen also mit einem abstrakten Datentyp arbeiten, der Verbindungen zu Datenbanken beschreibt. Um einen neuen Datenbanktyp hinzuzufügen, müssen dann alle Funktionen, die datenbankspezifisch sind, für diesen implementiert werden.

Die grundlegende Funktion dieser Komponente ist es dann, einen SQL-Befehl als String auf Datenbanken ausführen zu können und alle Ausgaben der jeweiligen Datenbank bezüglich dieses Befehls als Liste von Listen von Strings zurückzugeben. Jede Liste von Strings dieses Rückgabewertes soll jeweils eine Zeile des Ergebnis der Ausführung darstellen. Meldet die Datenbank einen Fehler statt einem Ergebnis, dann soll dieser erkannt werden. Es muss also zwischen einem korrekten Ergebnis und einem Fehler unterschieden werden können.

Dem Programmierer soll es zudem leicht möglich gemacht werden, eine Reihe von Aktionen auf der Datenbank wieder rückgängig zu machen, falls ein Fehler in einer Aktion auftritt.

Diese Komponente bildet somit den Grundbaustein der Schnittstelle und könnte auch separat von den anderen Komponenten genutzt werden, wenn ein Nutzer lediglich an diesen grundlegenden Funktionen interessiert ist und direkt SQL-Befehle ausführen möchte.

3.2.2. Typisierter Datenbankzugriff

Die zweite Komponente kann man als eine zweite Schicht auf der ersten Komponente betrachten, welche als Basis dient. Es werden in dieser Schicht Befehle implementiert, die mit einer Reihe von verschiedenen Datentypen, anstatt nur mit Strings.

Befehle in dieser Schicht können Platzhalter enthalten, welche bei der Ausführung durch mitgegebene Werte ersetzt werden. Die Datentypen der Variablen, welche hierbei möglich sein sollen, werden in dieser Schicht genau definiert. Auf diese Weise wird eine erste Abstraktionsebene geschaffen, da hier schon mit eigenen Datentypen gearbeitet wird und die Komponente dafür verantwortlich ist, die SQL-Befehle aus diesen schon leicht abstrahierten Befehlen zusammenzubauen. Der Vorteil dabei ist zum einen, dass Nutzer der Schnittstelle die SQL-Befehle nicht komplett selber schreiben müssen. Dadurch wird ihnen vor allem die Umwandlung von Variablen zu Strings abgenommen, die einen anderen Datentyp als `String` besitzen. Bei Datentypen, die keine offensichtliche Übersetzung zu einem String besitzen, hat das zudem den Vorteil, dass das korrekte Auslesen von gespeicherten Daten sichergestellt werden kann. Die Schnittstelle kennt die Art der Umwandlung von allen Datentypen zu Strings und kann so die korrekte Umwandlung von Strings zurück zu den Datentypen sicherstellen. Zum anderen ist ein weiterer wichtiger Vorteil ist, dass die Schnittstelle so einen Schutz gegen das Auftauchen von unerwünschten Zeichen und Zeichenketten in SQL-Befehlen bieten kann. Strings, die einen Platzhalter ersetzen sollen, können im Voraus zum Beispiel auf Escape-Zeichen überprüft werden, wodurch Schutz gegen Angriffe wie SQL-Injections gegeben werden

kann.

Die Rückgabewerte von Anfragen sollen in dieser Schicht "leicht typisiert"werden. Anstatt einer Liste von Listen von Strings soll eine Liste von Listen mit vom Nutzer gewünschten Datentypen als Rückgabewert geliefert werden. Der Nutzer soll angeben können, welche Datentypen er bei einer Anfrage erwartet, woraufhin die Schnittstelle die Ergebnisse dementsprechend interpretiert und umwandelt. Auch hier wird damit dem Nutzer Arbeit abgenommen, da er die Umwandlungen von Strings zu den gewünschten Datentypen nicht selber übernehmen muss. Wurden Daten mithilfe dieser Komponente in die Datenbank eingetragen, dann kann der Nutzer sich sicher sein, dass sie durch diese Funktionalität auch wieder korrekt gelesen werden.

3.2.3. Abstrahierte und typisierte Befehle für ER-Modelle

Die dritte Komponente kann man nun ebenfalls als weitere Schicht betrachten, welche das höchste Abstraktionslevel der Schnittstelle bietet und auf der zweiten Schicht aufbaut. In dieser Schicht werden SQL-Befehle komplett abstrahiert, sodass der Nutzer Befehle nur noch aus Funktionen und Datentypen der Schnittstelle erstellt.

Hier gibt es nur noch Funktionen, die mit Datentypen arbeiten, die Tabellen einer Datenbank beziehungsweise Entitäten darstellen. Befehle sind hier also stark bezüglich dieser Datentypen typisiert. Es werden Funktionen bereitgestellt, um diese Datentypen zu speichern, zu aktualisieren und auszulesen.

Die meisten Funktionen geben die Möglichkeit, dass man Kriterien zu den Befehlen hinzufügen kann. Zum einen können Kriterien eine typsichere Abstraktion von *WHERE-Klauseln* beinhalten, bei der die Datentypen der zu vergleichenden Werte übereinstimmen müssen, und zum anderen eine Abstraktion von zusätzlichen Optionen wie zum Beispiel Optionen zum Sortieren der Ergebnisse.

Darüber hinaus ist es dem Nutzer möglich, schon vorhandene definierte Datentypen zu kombinieren. Auch diese kombinierten Datentypen können gespeichert, aktualisiert und ausgelesen werden. Gewünschte Kriterien können auch hier hinzugefügt werden.

In dieser Schicht müssen Nutzer also nicht einmal mehr selber SQL-Befehle schreiben und ihre Datentypen werden nach dem Auslesen von der Schnittstelle zusammengesetzt. Den Nutzern wird somit noch ein weiteres Stück Arbeit abgenommen. Darüber hinaus herrscht eine noch höhere Typsicherheit und schon zur Übersetzungszeit werden Fehler gemeldet, die sich sonst erst zur Laufzeit bemerkbar gemacht hätten. So wirft der Compiler zum Beispiel direkt einen Fehler, wenn Werte in WHERE-Klauseln mit unterschiedlichen Typen verglichen werden. Auch allgemein fehlerhafte SQL-Befehle, die unter Umständen erst zur Laufzeit bemerkt worden wären, können so vermieden werden.

3.2.4. Automatische Erzeugung von ER-Modell-Datentypen und Datenbanken

Die vierte Komponente agiert separat von den anderen drei Komponenten. Sie hat die Aufgabe, aus einem ERD alle notwendigen Daten und eine SQLite-Datenbank zu erstellen, um die Schnittstelle mit diesem ERD nutzbar zu machen. Es müssen also für alle

Entitäten des ERDs Daten erstellt werden, damit die Funktionen der dritten Komponente mit diesen Entitäten arbeiten können. Die automatisch erstellte SQLite3-Datenbank setzt alle Regeln (NoKey/Key/Unique, Nullable/Not Nullable, Foreign Keys) aus dem ERD um, sodass bei Verletzungen dieser Regeln auch Fehlermeldungen geworfen werden.

Kann ein Nutzer sein Datenbankschema als ERD darstellen, dann muss er die Datenbank nicht selber erstellen und es wird ihm auch das Definieren der Datentypen in Curry abgenommen. Nach dem Erstellen des ERDs und einem kurzen Übersetzen mit dieser Komponente kann der Nutzer sofort sein eigentliches Programm schreiben, welches diese Schnittstelle nutzen soll. Nur falls kombinierte Datentypen gewünscht sind, muss der Nutzer diese noch selber definieren.

4. Implementierung

Das folgende Kapitel beschreibt die Implementierung der Schnittstelle. Es wird Schritt für Schritt erläutert, wie die Anforderungen aus dem Entwurf umgesetzt werden und wie die Implementierung im Einzelnen funktioniert.

4.1. Verbindung mit Datenbanken

4.1.1. Der Datentyp `Connection`

Da die Schnittstelle komplett auf der Verbindung zu Datenbanken basiert, muss diese Funktionalität als erstes umgesetzt werden. Es wird gefordert, dass die Verbindung zu SQLite-Datenbanken implementiert wird und dass es möglich ist, die Schnittstelle im Nachhinein durch andere Datenbank-Typen zu erweitern. Da es in Curry keine Typklassen wie in Haskell gibt, müssen Verbindungen als Datentyp definiert werden:

```
data Connection = SQLiteConnection
```

Zunächst gibt es nur den Konstruktor `SQLiteConnection`. Falls die Schnittstelle erweitert werden soll, dann können weitere Konstruktoren für den jeweiligen Datenbank-Typ hinzugefügt werden. Bisher besitzt `SQLiteConnection` keine weiteren Argumente, da noch nicht geklärt wurde, wie eine Verbindung zu einer SQLite-Datenbank aussehen kann.

Wie in Abschnitt 2.5 beschrieben, kann sehr einfach mit SQLite3 kommuniziert werden. Es bietet sich also an, von Curry aus einen SQLite3-Prozess zu starten und mit diesem direkt zu kommunizieren. Das Modul `IOExts`[1] bietet die Möglichkeit, Prozesse zu starten, auf deren *Input- und Outputstreams* man in der Form des Datentyps `Handle` zugreifen kann. `Connection` kann also wie folgt erweitert werden:

```
data Connection = SQLiteConnection Handle
```

Jeder Verbindungstyp sollte eine Funktion bereitstellen, mit der eine Verbindung aufgebaut wird. Diese sieht für SQLite wie folgt aus:

```
connectSQLite :: String -> IO Connection
connectSQLite str = do
  h <- connectToCommand $ "sqlite3 " ++ str
  hPutAndFlush h ".mode line"
  hPutAndFlush h ".log stdout"
  return $ SQLiteConnection h
```

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

Diese Funktion nimmt einen String als Parameter, der den Dateinamen der SQLite-Datenbank enthält, zu der eine Verbindung aufgebaut werden soll. Es werden ein SQLite3-Prozess gestartet und noch ein paar Optionen von SQLite3 geschaltet. Die Funktion `hPutAndFlush` schreibt einen String zum Standard-Input eines `Handles` und führt einen `Flush` auf dem Buffer des `Handles` aus. Die Option `.mode line` macht das Lesen der Werte einfacher, da so für jeden Wert eine Zeile ausgegeben wird und Werte immer nach einem Gleichheitszeichen starten. Um einer Zeile den Wert zu entnehmen, muss also nur folgendes gemacht werden:

```
getValue (_ ++ "= " ++ b) = b
getValue (_ ++ "=") = ""
```

Die Option `.log stdout` ist notwendig, damit Fehler ausgelesen werden können.

Wenn eine Verbindung nicht mehr gebraucht wird, dann ist es in vielen Fällen sinnvoll, dass die Verbindung getrennt wird. Aus diesem Grund muss jeder Verbindungs-Typ die Funktion `disconnect` mit folgendem Typ implementieren:

```
disconnect :: Connection -> IO ()
```

Im Fall von SQLite ist `disconnect` wichtig, da die SQLite3-Prozesse sonst erst geschlossen werden, wenn das Curry-Programm beendet wird. Werden in einem Curry-Programm häufig Verbindungen aufgebaut, dann sind unnötig viele Prozesse geöffnet.

```
disconnect (SQLiteConnection h) = hClose h
```

Es wird hier lediglich die Funktion `hClose` aufgerufen, welche den Prozess beendet.

Auch muss jede Verbindung die Funktion implementieren, einen beliebigen Befehl auf ihr auszuführen.

```
writeConnection :: String -> Connection -> IO ()
```

Die SQLite-Implementierung dafür sieht wie folgt aus:

```
writeConnection str (SQLiteConnection h) = hPutAndFlush h str
```

Zusätzlich müssen die folgenden drei Funktionen implementiert werden, die eine SQL-Transaktion starten, abschließen und zurücksetzen.

```
begin :: Connection -> IO ()
begin conn@(SQLiteConnection _) = writeConnection "begin;" conn

commit :: Connection -> IO ()
commit conn@(SQLiteConnection _) = writeConnection "commit;" conn

rollback :: Connection -> IO ()
rollback conn@(SQLiteConnection _) = writeConnection "rollback;" conn
```

4.1.2. Ausführung unverarbeiteter SQL-Befehle

Mit diesen Grundlagen kann nun schon die erste Haupt-Funktionalität implementiert werden. Es sollen SQL-Befehle mit Verbindungen ausführbar sein und Ergebnisse sollen

als Liste von Liste von Strings zurückgeliefert werden. Da laut Entwurf in der nächsten Schicht mit Platzhaltern gearbeitet werden soll, kann man diese Funktionalität auch hier für Strings schon anbieten. Es wird also zusätzlich eine Liste von Strings als Parameter mitgegeben. Diese Strings sollen alle Platzhalter der Form `'?'` im SQL-Befehl ersetzen. Der Typ dieser Funktion sieht also wie folgt aus:

```
executeRaw :: String -> [String] -> Connection -> IO [[String]]
```

Diese Funktion kann nun beispielsweise wie folgt genutzt werden:

```
do connection <- connectSQLite "Uni.db"
   result <- executeRaw
             "select * from Student where Age = '?' and Name = '?';"
             ["25", "'Meier'"]
             connection
```

Sollen Platzhalter durch Strings ersetzt werden, dann ist es wichtig, dass man die Strings umschlossen von Apostrophen angibt. Sie werden sonst als Spaltenname interpretiert.

Wird nun allerdings von einer Datenbank ein Fehler geworfen, zum Beispiel aufgrund eines fehlerhaften SQL-Befehls, dann kann man ihn so nicht direkt von einem normalen Ergebnis unterscheiden. Aus diesem Grund wird der Typ `Error` eingeführt:

```
data Error = Error ErrorKind String
data ErrorKind
  = TableDoesNotExist
  | ParameterError
  | ConstraintViolation
  | SyntaxError
  | NoLineError
  | UnknownError
```

Dieser Datentyp besitzt zum einen einen genauen Fehlertypen und zum anderen einen String, der den Fehler genauer beschreiben kann. Um normale Ergebnisse und Fehler als Ergebnis von Datenbank-Aktionen unterscheiden zu können, wird der Typ `Result` definiert:

```
type Result a = Either Error a
```

Da viele Funktionen dieser Schnittstelle eine `Connection` als Eingabewert benötigen und als Ergebnis einen Wert vom Typ `Result a` besitzen, macht es zudem Sinn, den Typ `DBAction a` zu definieren:

```
type DBAction a = Connection -> IO (Result a)
```

Nun kann der Typ der Funktion `executeRaw` erneut definiert werden:

```
executeRaw :: String -> [String] -> DBAction [[String]]
```

Die Funktion muss alle Platzhalter ersetzen, den Befehl im String dann ausführen und die Ergebnisse beziehungsweise den Fehler zurückgeben. Da das Lesen von Ergebnissen von Verbindung zu Verbindung unterschiedlich gelöst wird, muss diese Funktionalität gekapselt werden, sodass jeder Verbindungs-Typ eine eigene Implementierung vornehmen kann.

```
executeRaw query para conn = do
  let queryInserted = (insertParams query para)
  case queryInserted of
    Left err -> return $ Left err
    Right qu -> do
      writeConnection qu conn
      parseLines conn
```

Die Funktion `insertParams` ersetzt die Platzhalter. Die Anzahl von Platzhaltern und Strings in der Liste müssen übereinstimmen, sonst wirft die Funktion einen Fehler. In dieser Schicht muss der Nutzer noch selber darauf achten, dass das eingehalten wird. Die Funktion `parseLines` ist nun eine weitere Funktion, die Verbindungs-Typen implementieren müssen. Sie hat folgenden Typ:

```
parseLines :: DBAction [[String]]
```

Die Funktion soll jede bisher ungelesene Ausgabe der Datenbank lesen. Stellt die Ausgabe das Ergebnis einer `select`-Anfrage dar, dann soll jede Liste von Strings eine Reihe repräsentieren. Jeder gelesene Wert soll dabei genau so gelesen und in seiner String-Repräsentation zurückgegeben werden, wie er in der Datenbank steht. `NULL`-Werte müssen als leerer String zurückgegeben werden, anstatt als `"NULL"`. Wird eine Fehlermeldung gelesen, dann wird ein entsprechender Fehler zurückgegeben.

Die Verwendung dieser Funktion könnte dann beispielsweise wie folgt aussehen:

```
printStudents :: IO ()
printStudents =
  do connection <- connectSQLite "Uni.db"
     result <- executeRaw
               "select * from Student;"
               []
               connection
     case result of
       Left _ -> putStr "Error"
       Right xs -> putStr $ show xs
```

Diese Funktion gibt eine Liste von Listen aus, die alle Studenten repräsentiert, wobei jede Liste alle Werte eines Studenten in String-Form enthält. Im Falle eines Fehlers wird `"Error"` ausgegeben.

Die SQLite-Implementierung dieser Funktion liest den Input-Stream der Verbindung. Durch die Optionen `.mode line` und `.mode stdout` ist es sehr leicht möglich, zwischen Werten und Fehlern zu unterscheiden und die Werte korrekt zu lesen. Soll eine Zeile eines Input-Streams gelesen werden, welcher allerdings bereits am Ende angekommen ist, dann stoppt die Ausführung des Programms so lange, bis es eine weitere Zeile zu lesen gibt. Aus diesem Grund setzt die SQLite-Implementierung zusätzlich eine zufällige 16-stellige Hexadezimalzahl ans Ende der Ausgabe und liest nur so weit, bis sie diese erreicht hat.

4.1.3. Der Datentyp `DBAction`

Im vorherigen Abschnitt wurde der Datentyp `DBAction` definiert:

```
type DBAction a = Connection -> IO (Result a)
```

Da die meisten Funktionen dieser Schnittstelle diesen Typ als Rückgabewert haben werden, macht es Sinn, das Arbeiten damit zu vereinfachen. Wenn mit der Schnittstelle gearbeitet wird, dann könnte es zum Beispiel häufig vorkommen, dass mehrere Funktionen hintereinander ausgeführt werden sollen, die alle den Rückgabewert `DBAction` besitzen. Es macht also Sinn, Operatoren bereitzustellen, die Werte dieses Typs verbinden können. Somit ist es einfacher übersichtlichen Code zu schreiben.

```
(>+=) :: DBAction a -> (a -> DBAction b) -> DBAction b
(m >+= f) conn = do
  v1 <- m conn
  case v1 of
    Right val -> f val conn
    Left  err -> return (Left err)
```

Der Infix-Operator `(>+=)` verbindet einen Wert vom Typ `DBAction a` mit einer Funktion vom Typ `a → DBAction b` und kombiniert diese zu einem Wert vom Typ `DBAction b`. Wird diese kombinierte `DBAction b` dann mit einer `Connection` ausgeführt, dann wird zunächst der Wert vom Typ `DBAction a` ausgeführt. Mit dem Ergebnis dieser Ausführung wird daraufhin die Funktion vom Typ `a → DBAction b` ausgeführt. Das Ergebnis ist ein Wert vom Typ `Result b`. Liefert die erste `DBAction a` allerdings einen Fehler, dann wird die `DBAction b` direkt mit einem Fehler abgebrochen. Seien beispielsweise zwei Funktionen folgendermaßen definiert:

```
getStudents :: DBAction [[String]]
alterStudents :: [[String]] -> DBAction ()
```

Die Funktion `getStudents` liest Studenten aus der Datenbank aus und die Funktion `alterStudents` verändert eine Liste von Studenten auf irgendeine Weise und speichert sie ab. Sollen dafür die Studenten genommen werden, die `getStudents` liefert, kann geschrieben werden:

```
do connection <- connectSQLite "Uni.db"
   let f = getStudents >+= alterStudents
   f connection
```

Ein zweiter Operator wird eingeführt für den Fall, dass der Rückgabewert nicht gebraucht wird:

```
(>+) :: DBAction a -> DBAction b -> DBAction b
(>+) x y = x >+= (\_ -> y)
```

Dieser kann zum Beispiel für das Zusammenfügen von Befehlen genutzt werden, die etwas an der Datenbank verändern, aber kein Ergebnis haben.

Mit diesen Operatoren werden Datenbankaktionen lediglich abgebrochen, wenn ein Fehler auftritt. Das heißt allerdings, dass alle bisherigen Änderungen, die durch diese

Aktionen vorgenommen wurden, bestehen bleiben. Es wäre gut, wenn Änderungen auch wieder rückgängig gemacht werden könnten, sobald in einer Reihe von Datenbankaktionen ein Fehler auftritt. Das wird durch die folgende Funktion implementiert:

```
runInTransaction :: DBAction a -> DBAction a
runInTransaction act conn = do
  begin conn
  res <- act conn
  case res of
    Left  _ -> rollback conn >> return res
    Right _ -> commit   conn >> return res
```

Es wird die Funktion `begin` aufgerufen, bevor die Datenbankaktion startet. Nur wenn die Aktion erfolgreich war, wird `commit` aufgerufen. Ist dies nicht der Fall, dann wird `rollback` ausgeführt. Sei beispielsweise die Funktion `saveStudent` gegeben, die folgenden Typ hat:

```
saveStudent :: [String] -> DBAction ()
```

Seien zudem `student1`, `student2` und `student3` Listen von Strings, die Studenten darstellen. Möchte man nun alle drei speichern, aber auch nur, wenn das Speichern für alle erfolgreich ist, dann kann man das wie folgt umsetzen:

```
do connection <- connectSQLite "Uni.db"
   let f = (saveStudent student1) >+
           (saveStudent student2) >+
           (saveStudent student3)
   result <- runInTransaction f connection
   case result of
     Left _ -> putStr "Error"
     Right _ -> putStr "Success"
```

Es werden erst die drei Datenbankaktionen zu einer zusammengefasst. Dann wird diese Datenbankaktion mit `runInTransaction` ausgeführt. Tritt auch nur in einer der kombinierten Aktionen ein Fehler auf, dann werden alle Änderungen zurückgesetzt.

4.2. Typisierter Datenbankzugriff

Bisher wird bei Datenbankaktionen nur mit Strings gearbeitet. Im Entwurf wurde für die nächste Schicht gefordert, dass es möglich ist, mit Variablen von unterschiedlichen Typen zu arbeiten. Es muss also definiert werden, welche Datentypen in der Schnittstelle genutzt werden können. Da Curry von sich aus keine Typklassen anbietet, müssen diese Typen in einem Datentypen zusammengefasst werden. Funktionen aus dieser Schicht arbeiten dann mit diesem zusammenfassenden Datentypen, anstatt mit Strings.

```
data SQLValue
  = SQLString String
  | SQLInt    Int
  | SQLFloat  Float
```

```
  | SQLChar   Char
  | SQLBool   Bool
  | SQLDate   ClockTime
  | SQLNull
```

Eine Funktion zum Ausführen von SQL-Befehlen ohne Rückgabewerte kann so schon fast implementiert werden.

```
execute :: String -> [SQLValue] -> DBAction ()
```

Der Nutzer schreibt wie bei `executeRaw` einen SQL-Befehl mit möglichen Platzhaltern. Dieses mal werden die Platzhalter durch Werte einer Liste vom Typ `SQLValue` ersetzt. Die Werte müssen lediglich zu Strings übersetzt werden, woraufhin `executeRaw` genutzt werden kann, um den Befehl auszuführen.

```
execute query values conn =
  (executeRaw query (valuesToString values) >+
  (\_ -> return (Right ()))) conn
```

Die Funktion `valuesToString` muss also implementiert werden, bei der für jeden Typ von `SQLValue` eine Übersetzung zu einem String definiert wird. Die Übersetzung von `SQLFloat` sieht beispielsweise folgendermaßen aus:

```
SQLBool a -> "'" ++ show a ++ "'"
```

Auf diese Weise können auch gleich erste Sicherheitslücken geschlossen werden. Die Übersetzung von `SQLString` sieht beispielsweise wie folgt aus:

```
SQLString a -> "'" ++ doubleApostrophes a ++ "'"
```

Wenn ein Wert vom Typ `SQLString` Apostrophe enthält, dann werden diese verdoppelt. Da Apostrophe Variablen umschließen, müssen sie doppelt geschrieben werden, wenn man sie in einem Wert nutzen möchte. Würde man einzelne Apostrophe zulassen, dann stellt dies eine Sicherheitslücke dar. Sei beispielsweise der SQL-Befehl

```
select * from Student where Name = '?';
```

gegeben. Würde nun der Platzhalter ohne Verdoppelung der Apostrophe durch den String

```
"Max'; drop table Student; select * from Lecturer where Name = 'Muster"
```

ersetzt werden, dann wäre der vollständige Befehl:

```
select * from Student where Name = 'Max';
drop table Student;
select * from Lecturer where Name = 'Muster';
```

Dadurch würde die gesamte Studenten-Tabelle gelöscht werden. Das Verdoppeln der Apostrophe verhindert solche sogenannten SQL-Injections, da nur einzelne Apostrophe eine Variable beenden können. Die Implementierung der Übersetzungen der anderen Typen von `SQLValue` sind dabei sehr ähnlich aufgebaut.

Nun ist es also möglich, mit einigen verschiedenen Typen zu arbeiten, allerdings nur für Befehle, die keinen Rückgabewert besitzen. Bei Befehlen mit Rückgabewerten ist

es notwendig zu wissen, welche Typen dabei erwartet werden. Eine erste Idee ist es, zu versuchen, die Information über die Typen der Rückgabewerte bei jeder Anfrage von der Datenbank zu erhalten. Dann müsste für jeden Datenbank-Typ eine Implementierung einer solchen Funktionalität getätigt werden. Dies würde zu zusätzlicher Arbeit bei der Implementierung neuer Datenbank-Typen führen. Zudem ist es bei manchen Datenbank-Typen nicht notwendig, für Spalten einen eindeutigen Typ anzugeben. Aus diesem Grund muss ein anderer Ansatz gewählt werden. Der Ansatz, der hier gewählt wird, ist, dass es dem Nutzer überlassen wird zu entscheiden, welche Datentypen als Rückgabe erwartet werden.

Es muss also bei jedem Befehl mit Rückgabewert vom Nutzer die Information mitgegeben werden, welche Typen erwartet werden. Dafür bieten sich Listen an, die die Typen beschreiben. Eine erste Idee ist es, dafür `SQLValues` zu nutzen, deren Wert ignoriert wird. Um das Ganze allerdings ein wenig übersichtlicher zu gestalten, wird ein neuer Datentyp eingeführt.

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

Mit einer Liste dieses Datentyps können die Rückgabetypen eines Befehls beschrieben werden. So hat zum Beispiel die Anfrage

```
select * from Student;
```

Studenten als Rückgabewert, die aus den Typen `Int`, `String`, `String`, `String` und `Int` bestehen. Somit kann man diesen Rückgabewert mit

```
[SQLTypeInt, SQLTypeInt, SQLTypeString, SQLTypeString,
 SQLTypeString, SQLTypeInt]
```

beschreiben. Es muss also eine Funktion mit folgendem Typ implementiert werden:

```
select :: String -> [SQLValue] -> [SQLType] -> DBAction [[SQLValue]]
```

Es kann `executeRaw` genutzt werden, um diese Funktion zu schreiben. Die Liste von Listen von Strings muss dann mit Hilfe der Liste von `SQLTypes` zu einer Liste von Listen von entsprechenden `SQLValues` umgewandelt werden.

```
select query values types conn =
  (executeRaw query (valuesToString values) >+=
  (\a _ -> return (convertValues a types))) conn
```

Dabei ist `convertValues` eine Funktion mit folgendem Typ:

```
convertValues :: [[String]] -> [SQLType] -> Result [[SQLValue]]
```

Sie wandelt jeden String von jeder Liste, wie beschrieben in der Liste vom Typ `SQLType`, um. Wenn die Längen der Listen nicht übereinstimmen, dann wird ein Fehler geworfen.

Für jeden Typ von `SQLValue` muss dabei eine Hilfsfunktion geschrieben werden, die das Übersetzen von String zu SQLValue definiert. Die Übersetzung von String zu `SQLFloat`:

```
convertValuesHelp (s, SQLTypeFloat) =
  if isFloat s
    then case (readsQTerm s) of
           []         -> SQLNull
           ((a,_):_)  -> SQLFloat a

    else SQLNull
```

Es wird hier auch geregelt, dass Strings, die nicht den entsprechenden Typ repräsentieren, zu SQLNull umgewandelt werden. Alle weiteren Umwandlungen wurden auf sehr ähnliche Weise implementiert.

Eine Anfrage als Beispiel, die diese neuen Funktionalitäten nutzt, sieht wie folgt aus:

```
do connection <- connectSQLite "Uni.db"
   select "select * from Student where Age = '?' and Name = '?';"
          [SQLInt 24, SQLString "Muster"]
          [SQLTypeInt, SQLTypeInt, SQLTypeString,
           SQLTypeString, SQLTypeString, SQLTypeInt]
          connection
```

Das Ergebnis ist ein Wert vom Typ `Result [[SQLValue]]`, wobei die `SQLValues` in den Listen genau der Liste von `SQLTypes` entsprechen, außer wenn ein Wert nicht übersetzt werden konnte. In dem Fall ist es `SQLNull`.

4.3. Abstrahierte und typisierte Befehle für ER-Modelle

Es wurde nun die Grundlage für die oberste Schicht geschaffen, in der SQL komplett abstrahiert werden soll. Die Ergebnis-Typen in dieser Schicht sollen keine Listen von Werten mehr sein, sondern Entitäten aus ER-Modellen. Entitäten werden in Curry einfach als Datentypen dargestellt.

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

Der Typ `StudentID` existiert, um die IDs von Studenten von anderen IDs unterscheiden zu können. Das Ziel ist es nun, Funktionen bereitzustellen, die es möglich machen, dass die Ausgabe auf jeden Fall der gewünschte Typ ist. Ein Befehl, der zum Beispiel Studenten ausgibt, soll typsicher sein.

Eine der Funktionalitäten, die es zu implementieren gilt, ist ein Befehl der `select` von SQL abstrahiert. Der Nutzer soll nur angeben müssen, nach welcher Entität er sucht und welche Kriterien bei dieser Suche angewandt werden sollen. Die Funktion hat also einen Typ in der Art:

```
a -> Criteria -> DBAction [a]
```

Wobei `Criteria` ein Datentyp ist, der alle Kriterien enthält. Das Problem ist, dass eine Entität alleine nicht genügend Informationen bietet, um eine Anfrage zu stellen. Es muss ein Datentyp implementiert werden, der Entitäten genügend beschreibt, so dass Anfragen gestellt werden können, die mit diesen Entitäten arbeiten. Auch andere Funktionen als die `select`-Funktion werden diese Informationen benötigen.

4.3.1. Der Datentyp `EntityDescription`

Die Aufgabe ist es nun, einen Datentyp zu implementieren, der alle notwendigen Informationen zu einer Entität enthält, um Anfragen zu erstellen. Die erste Information, die benötigt wird, ist der Tabellenname der Entität in der Datenbank.

```
data EntityDescription = ED String
```

Da die Anfragen in dieser Schicht direkt die Entitäten als Ergebnis haben sollen, muss es auch eine Information darüber geben, aus welchen Typen die Entität besteht. So können die Anfragen aus der zweiten Schicht so genutzt werden, dass direkt die richtigen Typen geliefert werden.

```
data EntityDescription = ED String [SQLType]
```

Die Anfragen aus der zweiten Schicht liefern Listen mit `SQLValues`. Es ist also notwendig zu wissen, wie aus einer Liste von SQLValues die Entität zusammengesetzt wird. Soll zudem eine Entität gespeichert werden, dann muss die Entität zunächst zu einer Liste von SQLValues aufgeteilt werden. Es muss also eine Funktion mit dem Typ `Entität → [SQLValue]` und eine Funktion mit dem Typ `[SQLValue] → Entität` gegeben sein. Der Datentyp `EntityDescription` braucht also noch einen Typparameter und zwar die entsprechende Entität.

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

Nun kann zum Beispiel die EntityDescription für die Entität Student geschrieben werden.

```
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, SQLInt age])
    (\[SQLInt key, SQLInt matNum, SQLString name,
        SQLString firstName, SQLString email, SQLInt age] ->
        Student (StudentID key) matNum name firstName email age)
```

Es gibt so aber kleines Problem. Der Datentyp `Student` hat für die Spalte `Age` den Typ `(Maybe Int)`. Bei der Funktion von `Student` zu `[SQLValue]` hat ein Student für `Age` entweder den Typ `Just Int` oder `Nothing`. Bei der Funktion von `[SQLValue]` zu `Student` wird `Age` entweder durch ein `SQLInt Int` repräsentiert oder durch `SQLNull`.

`Just Int` wird zu `SQLInt Int` umgewandelt, `Nothing` zu `SQLNull` und umgekehrt. Die bisherigen Funktionen können nicht mit allen Fällen umgehen. Um dies möglich zu machen, sind Hilfsfunktionen notwendig, die überprüfen, welcher Fall gerade zutrifft. Um solche Fälle allgemein abzudecken, müssen für alle Typen, die als `SQLValue` definiert sind, Funktionen folgender Art implementiert werden:

```
sqlIntOrNull :: (Maybe Int) -> SQLValue
sqlIntOrNull Nothing = SQLNull
sqlIntOrNull (Just a) = SQLInt a

intOrNothing :: SQLValue -> (Maybe Int)
intOrNothing SQLNull = Nothing
intOrNothing (SQLInt a) = Just a
```

Für die weiteren Typen sind diese Funktionen analog implementiert. Nun können die gewünschten Funktionen so geschrieben werden, dass alle Fälle abgedeckt werden.

```
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))
```

Es werden zum einfacheren Programmieren noch ein paar Hilfsfunktionen der folgenden Art implementiert:

```
getTable :: EntityDescription a -> String
getTable (ED s _ _ _) = s

getTypes :: EntityDescription a -> [SQLType]
getTypes (ED _ t _ _) = t
```

Was nun noch fehlt für die Implementierung von Befehlen in dieser Schicht ist der Datentyp `Criteria`.

4.3.2. Der Datentyp `Criteria`

Der Datentyp `Criteria` soll Kriterien bei Anfragen darstellen. Es müssen also zum einen `WHERE`-Klauseln abstrahiert werden und zum anderen Optionen wie zum Beispiel `ORDER BY`. Ein Wert vom Typ `Criteria` besteht also aus einem `Constraint` und einer Liste von Optionen.

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

Der Datentyp `Constraint` soll alle Möglichkeiten einer `WHERE`-Klausel bieten. Die meisten Operatoren vergleichen Spalten mit Werten oder anderen Spalten. Dazu kommen rekursive Operatoren und der `EXISTS`-Operator. Es wird also ein Datentyp gebraucht, der Werte darstellt, die hier benutzt werden können. Es macht zudem Sinn, für Spalten einen Typ einzuführen.

```
data Value = Val SQLValue | Col Column
data Column = Column String String
```

Der erste String steht hier für den einfachen Namen der Spalte, der zweite String für den Namen der Spalte inklusive Namen der Tabelle. So sieht die Spalte `Name` von der Entität `Student` zum Beispiel so aus:

```
studentColumnName = Column "\"Name\"" "\"Student\".\"Name\""
```

Es macht Sinn, die Namen der Spalten in Anführungszeichen anzugeben, da so keine Verwechslungsgefahr zu anderen Befehlen besteht. Nennt man eine Spalte zum Beispiel `Select`, setzt den Namen bei Anfragen aber nicht in Anführungszeichen, dann kommt es zu einem Fehler. Eine erste Variante, das Ganze nun zu implementieren sieht wie folgt aus. Der `EXISTS`-Operator wird dabei noch außen vor gelassen. `None` wird genutzt, wenn kein Kriterium eingesetzt werden soll. `None` wurde eingeführt, da der Datentyp `Criteria` sonst `Maybe Constraint` als Wert haben müsste. Da in den meisten Fällen ein `Constraint` genutzt wird, erspart man sich so, dass jedes mal `Just Constraint` geschrieben werden muss.

```
data Constraint
  = IsNull           Value
  | IsNotNull        Value
  | BinaryRel RelOp  Value Value
  | Between          Value Value Value
  | IsIn             Value [Value]
  | Not              Constraint
  | And              [Constraint]
  | Or               [Constraint]
  | None

data RelOp = Eq | Neq | Lt | Lte | Gt | Gte | Like
```

Es können Werte miteinander verglichen und Werte vom Typ `Constraint` durch `Not` negiert oder durch `And` und `Or` miteinander verknüpft werden. `And` und `Or` erhalten Listen von `Constraints`, die dann alle mit `And` beziehungsweise `Or` zusammengesetzt werden.

In dieser Implementierung können Werte von allen Typen miteinander verglichen werden und auch der Typ der Spalten ist hier nicht von Bedeutung. Es wäre hier also die Aufgabe des Nutzers zu überprüfen, dass alle `Constraints` auch Sinn machen. Schöner wäre es, hier Typsicherheit zu implementieren, so dass nur Werte vom gleichen Typ miteinander verglichen werden können. Zunächst müssen Spalten dafür einen Typ haben, was dazu führt, dass `Value` auch einen Typ haben muss.

```
data Column _ = Column String String
```

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

Da der Typ in der Definition von `Column` nicht vorkommt, kann `_` geschrieben werden. Die Spalte `Name` kann nun per Signatur mit einem Typ versehen werden.

```
studentColumnName :: Column String
studentColumnName = Column "\"Name\"" "\"Student\".\"Name\""
```

Nun muss ein Weg gefunden werden, die einzelnen Vergleiche in einem `Constraint` nur mit gleichen Typen zu erlauben. Es wäre wünschenswert, könnten Vergleiche folgendermaßen geschrieben werden:

```
BinaryRel RelOp (Value a) (Value a)
```

Das wäre allerdings nur möglich, wenn `Constraint` auch einen Typ bekommen würde. Dann wäre ein `Constraint` aber nur noch in der Lage Vergleiche mit einem einzigen Typ durchzuführen. Möchte man aber beispielsweise eine Bedingung schreiben, bei der die Spalte `Name` und die Spalte `Age` mit Werten verglichen werden, dann wäre das so nicht mehr möglich. Dieses Problem wird gelöst, indem der Typ `()`, der sogenannte `Unit type`, zu Hilfe genommen wird. Zwei neue Typen werden eingeführt:

```
type CColumn = Column ()
type CValue = Value ()
```

Beim Datentyp `Constraint` wird `Value` durch `CValue` ersetzt:

```
data Constraint
  = IsNull           CValue
  | IsNotNull        CValue
  | BinaryRel RelOp  CValue CValue
  | Between          CValue CValue CValue
  | IsIn             CValue [CValue]
  | Not              Constraint
  | And              [Constraint]
  | Or               [Constraint]
  | None
```

Nun ist es anscheinend nur noch möglich, Werte mit dem Typ `()` miteinander zu vergleichen. Der Trick ist nun, dass man Werte mit anderen Typen zu Werten mit dem Typ `()` umwandeln kann. Das wird genutzt, um Konstruktor-Funktionen zu schreiben, die nur Werte vom gleichen Typ annehmen, aber ein `Constraint` liefern, wobei `Constraints` keinen Typ besitzen. Die Konstruktoren vom Datentyp `Constraint`, außer `And`, `Or`, `Not` und `None`, werden dann nicht exportiert, so dass man diese Funktionen nutzen muss, um die Vergleiche zu nutzen. Es werden also zunächst die Funktionen zum Umwandeln geschrieben:

```
toCColumn :: Column a -> Column ()
toCColumn (Column s1 s2) = (Column s1 s2)

toCValue :: Value a -> CValue
toCValue (Col (Column s1 s2) n) = Col (Column s1 s2) n
toCValue (Val               v1) = Val v1
```

Es werden zudem Konstruktor-Funktionen für den Datentyp `Value` bereitgestellt. Zwei Beispiele dafür:

```
int :: Int -> Value Int
int = Val ∘ SQLInt

col :: Column a -> Value a
col c = Col c
```

Nun können die Konstruktor-Funktionen geschrieben werden. Die `equal`-Funktion sieht beispielsweise so aus:

```
equal :: Value a -> Value a -> Constraint
equal v1 v2 = BinaryRel Eq (toCValue v1) (toCValue v2)

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

Für die meisten Funktionen werden auch gleich Infix-Operatoren implementiert. Die Implementierungen der restlichen Funktionen wurden äquivalent umgesetzt.

Nun können nur noch Werte verglichen werden, die auch denselben Typ besitzen, ein `Constraint` kann aber trotzdem aus mehreren Vergleichen bestehen, die jeweils verschiedene Typen besitzen. Als Beispiel folgt ein `Constraint` für eine Anfrage nach der Entität `Student`:

```
And [col studentColumnAge .>. int 21,
      col studentColumnName .=. string "Mustermann"]
```

Es wird gefordert, dass die Studenten älter als 21 sind und den Namen "Mustermann" tragen. Ein Programm, das dieses `Constraint` nun nutzt, kompiliert nur, wenn die Spalte `studentColumnAge` den Typ `Int` besitzt und die Spalte `studentColumnName` den Typ `String`. Fehler bei den Typen werden also schon zur Übersetzungszeit gemeldet.

Was nun noch fehlt ist das `EXIST`-Constraint. Es muss die entsprechende Tabelle angegeben werden können und die Möglichkeit existieren, für das `EXIST`-Constraint ein `Constraint` anzugeben.

```
type Table = String
data Constraint
  = ...
  | Exists Table Constraint
  | None
```

Wenn dieses `Constraint` nun von der Schnittstelle übersetzt wird, dann gibt es ein Problem. Wird ein `EXIST-Constraint` genutzt, um nach einer Entität zu suchen und in diesem wird die Tabelle derselben Entität verwendet, dann sind in dem `Constraint` des `EXIST-Constraints` die Vergleiche nicht mehr eindeutig. Sei beispielsweise das folgende `Constraint` bei einer Suche nach Studenten gegegeben:

```
Exists "Student" (And [col studentColumnAge .=. col studentColumnAge,
                       col studentColumnKey ./=. col studentColumnKey]
```

Es ist gewünscht, alle Studenten zu bekommen, für die ein anderer Student existiert, der gleich alt ist. Es ist allerdings nicht möglich, die beiden Studenten in den Vergleichen zu unterscheiden. Es muss also eine Möglichkeit geben, eine der Tabellen umzubenennen. Dies wird möglich gemacht, indem der Datentyp `Value` und die Funktion `col` geändert werden und eine Funktion `colNum` hinzugefügt wird. Zudem wird das `EXIST-Constraint` ein wenig geändert.

```
data Value a = Val SQLValue | Col (Column a) Int
col :: Column a -> Value a
col c = Col c 0
colNum :: Column a -> Int -> Value a
colNum c n = Col c n
Exists Table Int Constraint
```

Ein `EXIST-Constraint` hat nun noch einen Integer-Wert. Dieser Wert benennt die Tabelle entsprechend der Zahl um. Sei `n` der Integer-Wert und `t` der Name der Tabelle. Ist $n = 0$, dann wird `t` nicht geändert. In allen anderen Fällen wird `t` zu `"n" ++ t` umbenannt. Spalten werden auf dieselbe Weise umbenannt, indem die Funktion `colNum` genutzt wird. So kann das `EXISTS-Constraint` umgeschrieben werden.

```
Exists "Student" 1 (And [col studentColumnAge .=.
                           colNum studentColumnAge 1,
                         col studentColumnKe) ./=.
                           colNum studentColumnKey 1]
```

`"Student"` wird dann zu `"1Student"` und die Spalten `"Student.Age"` und `"Student.Key"`, die mit der Funktion `colNum` zu einem `Value` gemacht werden, werden zu `"1Student.Age"` und `"1Student.Key"` umbenannt. Spalten können nun also eindeutig zugewiesen werden. Somit sind alle gewünschten `Constraints` implementiert.

SQL-Befehle können noch mit Optionen versehen werden. Es werden die Optionen `ORDER BY ASCENDING` und `ORDER BY DESCENDING` implementiert. Diese Befehle brauchen lediglich eine Spalte als Wert und können recht simpel geschrieben werden.

```
data Option = AscOrder CColumn | DescOrder CColumn
```

Auch hier gibt es Konstruktor-Funktionen:

```
ascOrder :: Column a -> Option
ascOrder (Column s1 s2) = AscOrder (Column s1 s2)
descOrder :: Column a -> Option
descOrder (Column s1 s2) = DescOrder (Column s1 s2)
```

Damit ist der Datentyp `Criteria` vollständig.

4.3.3. Befehle mit Entitäten

Nun können die meisten Befehle implementiert werden. Als erstes wird die Funktion implementiert, die den `SELECT`-Befehl abstrahiert. Diese Funktion braucht einen Wert vom Typ `EntityDescription`, um alle notwendigen Informationen über die gesuchte Entität zu haben. Weiterhin braucht sie einen Wert vom Typ `Criteria`, damit der

Nutzer alle gewünschten Kriterien angeben kann. Der Ergebnistyp ist ein Wert vom Typ `DBAction`.

```
getEntries :: EntityDescription a -> Criteria -> DBAction [a]
```

Die Funktion kann einfach mit den Informationen aus dem Wert des Typs `EntityDescription a` die Funktionen der zweiten Schicht nutzen. Zusammen mit dem Wert des Typs `Criteria` wird ein entsprechender SQL-Befehl zusammengesetzt. Es muss also eine Funktion zum Übersetzen von `Criteria`-Werten geben. Wenn man diese hat hat, kann man `getEntries` implementieren.

```
getEntries en c conn = do
  let query = "select * from '" ++ getTable en ++
              "' " ++ trCriteria c ++ ";"
  ((select query [] (getTypes en)) >+=
    (\vals _ -> return $ Right (map (getToEntity en) vals)))
    conn
```

Dabei ist `trCriteria` die Funktion zum Übersetzen von `Constraints` zu entsprechendem SQL. Nun kann zum Beispiel nach der Entität `Student` gesucht werden.

```
criteria = (Criteria
             (Exists "Student" 1
                     (And [col studentColumnAge .=.
                             colNum studentColumnAge 1,
                           col studentColumnKey ./=.
                             colNum studentColumnKey 1]))
             [ascOrder studentColumnAge])

f :: IO ()
f = do
  connection <- connectSQLite "Uni.db"
  result <- getEntries studentDescription criteria conn
  case result of
    Left _ -> putStr "Error"
    Right st -> putStr $ show st
```

Hier wird das `Constraint` aus dem vorherigen Kapitel genommen und die Ergebnisse werden aufsteigend nach dem Alter sortiert. Das Ergebnis wird danach ausgegeben, solange kein Fehler auftritt.

Entitäten abzuspeichern ist ein wenig einfacher, da es hier keine Rückgabewerte und keine Kriterien gibt.

```
saveEntry :: a -> EntityDescription a -> DBAction ()
saveEntry a en conn = do
  let query = "insert into '" ++ getTable en
              ++ "' values("++ questionmarks en ++");"
  execute query (getToValues en a) conn
```

Die Funktion `questionmarks` setzt die korrekte Anzahl an Fragezeichen-Platzhaltern in den SQL-Befehl.

Da man Entitäten nicht nur speichern, sondern auch bestehende Entitäten aktualisieren möchte, muss auch dafür eine Funktion geschrieben werden. Zunächst wird die Funktionalität des SQL-Befehls `UPDATE` direkt abstrahiert. Es muss also wieder ein Wert vom Typ `EntityDescription` gegeben sein. Dazu muss es eine Information darüber geben, welche Werte in welche Spalten geschrieben werden sollen. Es wird dafür folgender Datentyp mit Konstruktor-Funktion eingeführt:

```
data ColVal = ColVal CColumn CValue
colVal :: Column a -> Value a -> ColVal
colVal c v = ColVal (toCColumn c) (toCValue v)
```

So kann eine Liste von diesem Typ der Funktion übergeben werden und jeder Spalte wird beim Aktualisieren ihr entsprechender Wert zugewiesen. Dazu kommt ein Wert vom Typ `Constraint`, der einschränkt, welche Entitäten aktualisiert werden sollen.

```
updateEntries :: EntityDescription a -> [ColVal] ->
                 Constraint -> DBAction ()
```

Die Implementierung übersetzt diese Informationen einfach in den entsprechenden SQL-Befehl.

Es wäre wünschenswert, wenn man einzelne Entitäten aktualisieren könnte, indem man sie einfach einer Funktion übergibt und der entsprechende Eintrag mit dem zugehörigen Schlüssel wird aktualisiert.

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

Damit das funktioniert, besteht die Voraussetzung, dass die Entität einen primären Schlüssel als ersten Wert besitzt. Dann kann der `UPDATE`-Befehl von SQL genutzt werden, wobei in der `WHERE`-Klausel gefordert wird, dass die erste Spalte der Tabelle gleich dem ersten Wert der eingegeben Entität ist. Ist der erste Wert kein Schlüssel, dann kann es passieren, dass mehrere Entitäten aktualisiert werden. Da die Funktion `updateEntries` mächtiger ist, als die Funktion `updateEntry`, kann `updateEntry` mit der Hilfe von `updateEntries` implementiert werden. Die Funktion `updateEntries` fordert allerdings vom Nutzer alle Spalten, die aktualisiert werden sollen. Die Funktion `updateEntry` muss also eine Möglichkeit haben, alle Spaltennamen der Entität zu erfahren. Die erste Schicht der Schnittstelle muss also um eine Funktion erweitert werden, die alle Verbindungstypen implementieren müssen. Diese nimmt den Namen einer Tabelle und liefert die Namen aller Spalten in einer Liste.

```
getColumnNames :: String -> DBAction [String]
```

Die SQLite-Variante nutzt dazu den Befehl `pragma table_info`, welcher mehrere Informationen über Spalten ausgibt, und liest aus dem Ergebnis die Spaltennamen aus. Nun kann `updateEntry` implementiert werden.

```
updateEntry ent ed conn = do
  let table = (getTable ed)
  result <- getColumnNames table conn
  case result of
    Left err -> return (Left err)
```

```
Right columns -> do
  let values = getToValues ed ent
  let (SQLInt key) = head values
  let keycol = head columns
  let colvals = zipWith (colValAlt table) columns values
  let column = Column ("\"" ++ keycol ++ "\"")
                      ("\"" ++ table ++ "\".\"" ++ keycol ++ "\"")
  let const = col column .=. int key
  updateEntries ed colvals const conn
```

Es werden alle notwendigen Informationen geholt, damit `updateEntries` genutzt werden kann. Der erste Wert der Entität und der erste Spaltenname werden genutzt, um das `Constraint` zusammenzusetzen.

Es wurden noch weitere Funktionen implementiert, die aber alle recht simpel sind und mit den bisherigen Mitteln implementiert werden konnten. Die Signaturen und Beschreibungen dieser Funktionen sind im Anhang (A.1.3) zu sehen.

Es wurde gefordert, dass auch mit kombinierten Datentypen gearbeitet können werden soll, was häufig auch sinnvoll eingesetzt werden kann. So hat zum Beispiel die Entität `Exam` genau eine Beziehung zur Entität `Lecture`, genau eine zur Entität `Place` und genau eine zur Entität `Time`. Soll in einem Programm nun mit Klausuren gearbeitet werden, dann kann es Sinn machen, die Vorlesung, den Platz und die Zeit gleich dabei zu haben.

```
data CombinedExam = CombinedExam Exam Lecture Place Time
```

Es wäre also schön, Funktionen zu haben, mit denen man direkt mit zusammengesetzten Typen arbeiten kann.

4.3.4. Der Datentyp `CombinedDescription`

Wie auch bei Funktionen für normale Entitäten brauchen natürlich auch die Funktionen für zusammengesetzte Funktionen alle notwendigen Informationen, um die gewünschten Befehle auszuführen. Es muss also ein ähnlicher Datentyp wie `EntityDescription` her. Die notwendigen Informationen sind zum einen alle Informationen über die Entitäten, aus denen der kombinierte Typ zusammengesetzt ist und zum anderen die Art, wie der kombinierte Typ aus diesen Entitäten zusammengesetzt ist. Es soll auch möglich sein, einen kombinierten Typ zu schreiben, der nur manche Werte von den zu kombinierenden Entitäten beinhaltet. Ein Beispiel dafür wäre ein kombinierter Typ aus den Entitäten `Student` und `Lecture`, der von der Entität `Lecture` aber nur die `LectureID` und den Namen der Vorlesung beinhaltet.

Die erste Idee für die Informationen über die Entiäten wäre, dass einfach eine Liste von `EntityDescriptions` genommen wird, da diese alle notwendigen Informationen enthalten.

```
data CombinedDescription a = CD [EntityDescription a] ...
```

Es ist so nun allerdings nicht möglich, dass in der Liste `EntityDescriptions` mit unterschiedlichen Typen sind. Stattdessen wird eine Liste von Tripeln genutzt. Zwei Werte

davon sind der Tabellenname der Entität und die Liste von `SQLTypes`, die die Entität beschreiben. Da es auch möglich sein soll, dass die gleiche Entität kombiniert werden kann, muss man diese voneinander unterscheiden können. Es wird also ähnlich wie beim `Exists-Constraint` noch ein Integer-Wert benötigt. Dieser sagt wieder aus, wie Tabellen bei Befehlen umbenannt werden sollen. Dazu kommen zwei Funktionen. Die erste beschreibt wie der kombinierte Datentyp zu einer Liste von `SQLValues` umgewandelt wird und die zweite wie eine Liste von `SQLValues` zum kombinierten Datentyp zusammengesetzt werden.

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

Nun wäre es für den Nutzer schön, wenn es eine einfachere Weise geben würde, diesen Datentyp zusammen zu setzen, als ihn komplett von Hand zu schreiben. Aus diesem Grund werden Konstruktor-Funktionen implementiert. Nutzer sollen die jeweiligen `EntityDescriptions` der zu kombinierenden Typen angeben können und dazu zwei Funktionen. Die erste wandelt die Entitäten zum kombinierten Datentyp um und die zweite macht es andersherum. Auch die Integer-Werte zum Umbenennen müssen gegeben sein. Da auch hier das Problem mit den Typen von `EntityDescription` herrscht, können nur jeweils zwei Entitäten kombiniert werden. Eine zweite Funktion macht es dann möglich, noch weitere Entitäten hinzuzufügen. Die erste Funktion wurde wie folgt implementiert:

```
combineDescriptions :: EntityDescription a -> Int -> EntityDescription b ->
                       Int -> (a -> b -> c) -> (c -> (a, b)) ->
                       CombinedDescription c
combineDescriptions ed1 rename1 ed2 rename2 f1 f2 =
  CD [(getTable ed1, rename1, getTypes ed1),
      (getTable ed2, rename2, getTypes ed2)]
     createFunction1 createFunction2
    where createFunction1 xs = f1 ((getToEntity ed1)
                                    (take lengthEd1 xs))
                                  ((getToEntity ed2)
                                    (drop lengthEd1 xs))
             where lengthEd1 = length (getTypes ed1)
          createFunction2 combEnt =
            let (ent1, ent2) = f2 combEnt in
              ((getToValues ed1) ent1) : [(getToValues ed2) ent2]
```

Die Funktion setzt den kombinierten Datentyp zusammen, indem zunächst die Liste mit den benötigten Informationen erstellt wird. Dann wandelt sie die vom Nutzer gegebenen Funktionen zu Funktionen um, die für eine `combinedDescription` notwendig sind. Die Funktion `drop n xs` entfernt die ersten `n` Werte aus der Liste `xs`. Die Funktion zum Hinzufügen einer weiteren Entität muss die `EntityDescription` dieser Entität erhalten, den Integer-Wert zum Umbenennen und zwei Funktionen. Die erste Funktion ist zum Definieren, wie die Entität zu einem schon bestehenden Wert des kombinierten Datentyps

hinzugefügt wird und die zweite beschreibt, wie man die zu hinzufügende Entität wieder aus dem kombinierten Datentyp erhält.

```
addDescription :: EntityDescription a -> Int -> (a -> b -> b) -> (b -> a) ->
                  CombinedDescription b -> CombinedDescription b
addDescription ed1 rename f1 f2 (CD xs f1' f2') =
  CD ((getTable ed1, rename, getTypes ed1) : xs)
     createFunction1
     createFunction2
    where createFunction1 ys =
            f1 ((getToEntity ed1)
               (take lengthEd1 ys))
               (f1' (drop lengthEd1 ys))
            where lengthEd1 = length (getTypes ed1)
          createFunction2 combEnt =
            [(getToValues ed1) (f2 combEnt)] ++ (f2' combEnt)
```

Die Funktion fügt die Informationen der Entität der Liste hinzu und erweitert die Funktionen der bisherigen `combinedDescription`, sodass diese mit der hinzuzufügenden Entität richtig umgehen. Soll nun zum Beispiel der kombinierte Datentyp

```
data StudentStudentExam = StudentStudentExam Student Student Exam
```

beschrieben werden, kann das wie folgt gemacht werden:

```
cd = combineDescriptions studentDescription 0 examDescription 0
                         (\st ex -> (StudentStudentExam _ st ex))
                         (\(StudentStudentExam _ st ex) -> (st, ex))
```

Zunächst werden die erste `studentDescription` und `examDescription` miteinander kombiniert. Die Funktion von `Student` und `Exam` zu `StudentStudentExam` setzt diesen Wert noch nicht vollständig zusammen, da noch eine weitere Entität dazu kommt. Ebenso wird bei der zweiten Funktion der erste Wert von `StudentStudentExam` noch nicht genutzt. Dann kann die dritte Entität hinzugefügt werden.

```
sseDescription =
  addDescription studentDescription 1
  (\st1 (StudentStudentExam _ st2 ex) -> (StudentStudentExam st1 st2 ex))
  (\(StudentStudentExam st _ _) -> st) cd
```

Für kombinierte Datentypen werden äquivalente Funktionen zu denen für Entitäten implementiert.

```
getEntriesCombined :: CombinedDescription a -> Criteria -> DBAction [a]

saveEntryCombined :: a -> CombinedDescription a -> DBAction ()

updateEntryCombined :: a -> CombinedDescription a -> DBAction ()
```

Die Implementierungen dieser Funktionen sind sehr ähnlich zu denen für normale Entitäten. Beim Speichern und Aktualisieren muss alles lediglich für jede Entität gemacht werden, aus denen der kombinierte Datentyp besteht, anstatt nur für eine. Bei der Funktion

`getEntriesCombined` wird mit einem `CROSS JOIN` zwischen den kombinierten Tabellen gearbeitet.

4.4. Automatische Erzeugung von ER-Modell-Datentypen und Datenbanken

Die drei Schichten der Schnittstelle, die sich mit dem Ausführen von Befehlen beschäftigen, sind nun fertiggestellt. Was nun noch fehlt ist die Komponente, die die Datentypen für den Nutzer erstellt. Sie soll es Nutzern ermöglichen, dass sie nur ein ER-Modell erstellen müssen, um mit der Schnittstelle inklusive den Entitäten aus dem ER-Modell arbeiten zu können.

Da Datenbanken mit relationalen Datenbankschemata arbeiten, muss das ER-Modell zunächst übersetzt werden. Die Übersetzung wird von dem Tool `erd2curry` übernommen. Der Nutzer übergibt der Komponente also das relationale Datenbankschema, dass aus dem ER-Modell übersetzt worden ist. Die Komponente liest dann das relationale Datenbankschema ein und erzeugt eine neue Datei, in der alle Datentypen definiert werden. Die Hauptfunktion dieser Komponente hat den folgenden Typ:

```
writeCDBI :: ERD -> IO ()
```

Diese Funktion erstellt zunächst die Datei, in die die Datentypen und Funktionen geschrieben werden sollen. Ein paar grundlegende Sachen werden von ihr schon geschrieben. So werden zum Beispiel alle notwendigen Module importiert. Des Weiteren ruft die Funktion für alle Entitäten aus dem relationalen Schema eine Funktion auf.

```
writeEntityData :: Entity -> Handle -> IO ()
```

Die Funktion schreibt für eine Entität alle notwendigen Datentypen und Funktionen. Der Wert vom Typ `Handle` ist für die Datei, in die geschrieben wird. Es wird folgendes geschrieben:

- Der Datentyp, der die Entität darstellt, inklusive Datentyp für die ID
- Die entsprechende `EntityDescription`
- Der Tabellenname mit dem Typ `Table`
- Alle Spalten mit dem Typ `Column a`
- Funktionen zum einfachen Setzen und Erhalten von Werten der Entität

Die Funktion `writeCDBI` erstellt zudem eine `SQLite`-Datenbankdatei, indem ein `SQLite3`-Prozess gestartet wird und dabei der gewünschte Datenbankname angegeben wird. Dann wird folgende Funktion ausgeführt:

```
createDatabase :: [Entity] -> Handle -> IO ()
```

Diese bekommt eine Liste mit allen Entitäten und wieder einen Wert vom Typ `Handle`. Dieser ist hier für den `SQLite3`-Prozess zuständig, damit die gewünschten Tabellen erstellt werden können. Die Funktion erstellt nun für jede Entität eine Tabelle in der Datenbank. Dabei werden fast alle Eigenschaften, die in der Entität für die verschiedenen Attribute angegeben werden, auch in den Tabellen eingestellt. So werden zum Beispiel die Schlüssel- und Fremdschlüssel korrekt gesetzt und die Eigenschaft, ob NULL-Werte erlaubt sind in einer Spalte, wird gesetzt.

Nachdem diese Funktionen alle durchgelaufen sind, ist die Datei mit allen Daten und Funktionen fertig gestellt. Im Anhang (A.3) ist ein Ausschnitt der Datei zu sehen, die diese Komponente mit dem `ERD` der Universität erstellt hat.

5. Fazit

Das Ziel der Arbeit war es, eine Schnittstelle für Curry zum implementieren, die SQL komplett abstrahiert und es möglich macht, mit Entitäten aus ER-Modellen zu arbeiten. Es sollte dabei möglich gemacht werden, auch komplexe Anfragen zu stellen. Dieses Ziel wurde erreicht. Es wurden typsichere Befehle implementiert, die direkt mit Entitäten arbeiten und für Anfragen können komplexe Kriterien formuliert werden. Dabei basiert alles auf Datentypen und Funktionen, die in Curry implementiert sind, die dann in entsprechendes SQL übersetzt werden. Auch ist es möglich, dass ein Nutzer die Schnittstelle sofort nutzen kann, wenn ein ER-Modell geschrieben wurde. Es ist hier vom Nutzer kein weiterer Aufwand notwendig, als das ER-Modell übersetzen zu lassen.

Die Schnittstelle kann allerdings durchaus noch erweitert und verbessert werden. So wurde die Schnittstelle bisher noch nicht umfangreich genug getestet, um sicherstellen zu können, dass alles fehlerfrei läuft. Des Weiteren könnte die Schnittstelle noch um weitere Datenbank- und Variablentypen erweitert werden. So könnte versucht werden, dass der Variablentyp `BLOB` nutzbar ist. Der Datentyp `ErrorKind` kann ebenso noch ausgebaut werden, da noch nicht alle Arten von Fehlern abgedeckt sind. Das könnte gut in eine ausgiebige Testphase integriert werden, da man dort auf viele weitere Fehler stoßen könnte. Auch bisher fehlende SQL-Befehle wie `GROUP BY` oder `HAVING` könnten noch implementiert werden. Beim Testen der Schnittstelle ist aufgefallen, dass das Schreiben von komplexen Kriterien manchmal ein wenig umständlich ist. Aus diesem Grund wäre es eine gute Idee, die Syntax hier für den Nutzer einfacher und schöner zu gestalten. Es wäre auch in Betracht zu ziehen, dabei noch einen Schritt weiter zu gehen und zu versuchen, eine *Code Integration* von SQL für Curry zu implementieren. Dabei könnte ein Nutzer pure SQL-Befehle (oder Befehle, die SQL-Befehlen sehr ähnlich sind) schreiben, welche dann zu den Datentypen und Funktionen dieser Schnittstelle übersetzt werden. Beim Übersetzen von `ERDs` zu Datentypen für die Schnittstelle könnten die Übersetzungsschritte von `erd2curry` und der Übersetzungskomponente dieser Schnittstelle vereint werden, sodass der Nutzer nur einen Schritt ausführen muss. Die bisher notwendigen Schritte werden im Anhang (A.2) erläutert.

Die Entwicklung in Curry war durchgehend zufriedenstellend. Jede Funktionalität, die entwickelt werden sollte, konnte meistens sehr einfach umgesetzt werden. Gäbe es in Curry Typklassen, so wie in Haskell, dann hätten lediglich einige Dinge eleganter gelöst werden können. So hätten zum Beispiel für Verbindungstypen, `SQLValues` oder Entitäten Typklassen eingeführt werden können.

A. Anhang

A.1. Übersicht der Schnittstelle

In diesem Abschnitt wird eine Übersicht über die Schnittstelle gegeben. Für jede Komponente werden die Signaturen der exportierten Funktionen samt Erklärung dieser und alle wichtigen Datentypen angegeben. Da die Schnittstelle in Englisch verfasst ist, sind die Erklärungen die originalen englischen Kommentare.

A.1.1. Direkte Verbindung mit Datenbanken

- **Modul: CDBIConnection**

```
--- A DBAction takes a connection and
--- returns an `IO (Result a)`.
type DBAction a = Connection -> IO (Result a)

--- `Result`s are either an `Error` or a value.
type Result a = Either Error a

--- `Error`s are composed of an ErrorKind`
--- and a `String`
--- describing the error more explicitly.
data Error = Error ErrorKind String

--- The different kinds of errors.
data ErrorKind
  = TableDoesNotExist
  | ParameterError
  | ConstraintViolation
  | SyntaxError
  | NoLineError
  | UnknownError

--- Data type for database connections.
--- Currently, only connections to a SQLite3 database are supported,
--- but other types of connections could easily be added.
--- List of functions that would need to be implemented:
--- A function to connect to the database, disconnect,
--- writeConnection, readRawConnection, parseLines,
```

```
--- begin, commit, rollback and getColumnNames
data Connection = SQLiteConnection Handle

--- Run a `DBAction` as a transaction.
--- In case of an `Error` it will rollback all changes,
--- otherwise the changes
--- are committed.
runInTransaction :: DBAction a -> DBAction a

--- Connect two `DBAction`s.
--- When executed this function will execute the first `DBAction`
--- and then execute the second applied to the first result
--- An `Error` will stop either action.
(>+=) :: DBAction a -> (a -> DBAction b) -> DBAction b

--- Connect two DBActions, but ignore the result of the first.
(>+) :: DBAction a -> DBAction b -> DBAction b

--- Failing action.
fail :: Error -> DBAction a
fail err _ = return (Left err)

--- Successful action.
ok :: a -> DBAction a
ok val _ = return (Right val)

--- Connect to a SQLite Database
connectSQLite :: String -> IO Connection

--- Disconnect from a database.
disconnect :: Connection -> IO ()

--- Begin a Transaction.
begin :: Connection -> IO ()

--- Commit a Transaction.
commit :: Connection -> IO ()

--- Rollback a Transaction.
rollback :: Connection -> IO ()

--- Execute a SQL statement.
--- The statement may contain '?' placeholders and a list
--- of parameters which
--- should be inserted at the respective positions.
--- The result is a list of list of strings where
```

```
--- every single list represents a row of the result.
executeRaw :: String -> [String] -> DBAction [[String]]
```

A.1.2. Typisierter Datenbankzugriff

- **Modul: CDBIConnection**

```
--- Data type for SQL values, used during the
--- communication with the database.
data SQLValue
  = SQLString String
  | SQLInt    Int
  | SQLFloat  Float
  | SQLChar   Char
  | SQLBool   Bool
  | SQLDate   ClockTime
  | SQLNull

--- Type identifiers for ‘SQLValue‘s, necessary to
--- determine the type
--- of the value a column should be converted to.
data SQLType
  = SQLTypeString
  | SQLTypeInt
  | SQLTypeFloat
  | SQLTypeChar
  | SQLTypeBool
  | SQLTypeDate

--- execute a query where the value of the result is returned
select :: String -> [SQLValue] -> [SQLType]
          -> DBAction [[SQLValue]]


--- execute a query without the result having a value
execute :: String -> [SQLValue] -> DBAction ()

--- execute a query multiple times with different
--- SQLValues without the result having a value
executeMultipleTimes :: String -> [[SQLValue]]
                        -> DBAction ()
```

A.1.3. Abstrahierte und typisierte Befehle für ER-Modelle

- **Modul: CDBIDescription**

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

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

type Table = String

data Column _ = Column String String

--- A constructor for CombinedDescription.
combineDescriptions :: EntityDescription a -> Int ->
                       EntityDescription b -> Int ->
                       (a -> b -> c) -> (c -> (a, b)) ->
                       CombinedDescription c

--- Adds another ED to an already existing CD
addDescription :: EntityDescription a -> Int ->
                  (a -> b -> b) -> (b -> a) ->
                  CombinedDescription b ->
                  CombinedDescription b
```

- **Modul: CDBICriteria**

```
data Criteria = Criteria Constraint [Option]

data Option = AscOrder CColumn | DescOrder CColumn

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

data ColVal = ColVal CColumn CValue

data Constraint
  = IsNull           CValue
  | IsNotNull        CValue
  | BinaryRel RelOp  CValue CValue
  | Between          CValue CValue CValue
  | IsIn             CValue [CValue]
  | Not              Constraint
  | And              [Constraint]
  | Or               [Constraint]
  | Exists           Table Int Constraint
  | None

data RelOp = Eq | Neq | Lt | Lte | Gt | Gte | Like
```

```
--- An empty criteria
emptyCriteria :: Criteria

--- Constructor for a Value Val of type Int
int :: Int -> Value Int

--- Constructor for a Value Val of type Float
float :: Float -> Value Float

--- Constructor for a Value Val of type Char
char :: Char -> Value Char

--- Constructor for a Value Val of type String
string :: String -> Value String

--- Constructor for a Value Val of type Bool
bool :: Bool -> Value Bool

--- Constructor for a Value Val of type ClockTime
date :: ClockTime -> Value ClockTime

--- Constructor for a Value Col without a rename-number
col :: Column a -> Value a

--- Constructor for a Value Col with a rename-number
colNum :: Column a -> Int -> Value a

--- A constructor for ColVal needed for typesafety
colVal :: Column a -> Value a -> ColVal

--- IsNull construnctor
isNull :: Value a -> Constraint

--- IsNotNull construnctor
isNotNull :: Value a -> Constraint

--- Equal construnctor
equal :: Value a -> Value a -> Constraint

--- Infix Equal
(.=.) :: Value a -> Value a -> Constraint

--- NotEqual construnctor
notEqual :: Value a -> Value a -> Constraint
```

```
--- Infix NotEqual
(./=.) :: Value a -> Value a -> Constraint

--- GreatherThan construnctor
greaterThan :: Value a -> Value a -> Constraint

--- Infix GreaterThan
(.>.) :: Value a -> Value a -> Constraint

--- LessThan construnctor
lessThan :: Value a -> Value a -> Constraint

--- Infix LessThan
(.<.) :: Value a -> Value a -> Constraint

--- GreaterThanEqual construnctor
greaterThanEqual :: Value a -> Value a -> Constraint

--- Infix GreaterThanEqual
(.>=.) :: Value a -> Value a -> Constraint

--- LessThanEqual construnctor
lessThanEqual :: Value a -> Value a -> Constraint

--- Infix LessThanEqual
(.<=.) :: Value a -> Value a -> Constraint

--- Like construnctor
like :: Value a -> Value a -> Constraint

--- Infix Like
(.~.) :: Value a -> Value a -> Constraint

--- Between construnctor
between :: Value a -> Value a -> Value a -> Constraint

--- IsIn construnctor
isIn :: Value a -> [Value a] -> Constraint

--- Infix IsIn
(.<->.) :: Value a -> [Value a] -> Constraint

--- Constructor for the option: Ascending Order by Column
ascOrder :: Column a -> Option

--- Constructor for the option: Descending Order by Column
```

```
descOrder :: Column a -> Option
```

- **Modul: CDBI**

```
--- Saves an entry to the database
saveEntry :: a -> EntityDescription a -> DBAction ()

--- Saves multiple entries to the database
saveMultipleEntries :: [a] -> EntityDescription a ->
                       DBAction ()

--- Gets entries from the database
getEntries :: EntityDescription a -> Criteria -> DBAction [a]

--- Gets combined entries from the database
getEntriesCombined :: CombinedDescription a -> Criteria ->
                      DBAction [a]

--- Saves combined entries
saveEntryCombined :: a -> CombinedDescription a ->
                     DBAction ()

--- Updates entries depending on wether they fulfill
--- the criteria or not
updateEntries :: EntityDescription a -> [ColVal] ->
                 Constraint -> DBAction ()

--- Updates an entry by ID. Works for Entities that
--- have a primary key as first value.
--- Function will update the entry in the
--- database with the ID of the entry that is
--- given as parameter with the values of
--- the entry given as parameter
updateEntry :: a -> EntityDescription a -> DBAction ()

--- Same as updateEntry but for combined Data
updateEntryCombined :: a -> CombinedDescription a ->
                       DBAction ()

--- Deletes entries depending on wether they
--- fulfill the criteria or not
deleteEntries :: EntityDescription a ->
                 (Maybe Constraint) -> DBAction ()

--- Drops a table from the database
dropTable :: EntityDescription a -> DBAction ()
```

A.2. Gebrauchsanweisung

In diesem Abschnitt wird erläutert, wie die Schnittstelle mit einem ER-Modell genutzt werden kann. Wurde ein ER-Modell erstellt und dieses liegt als `ERD` in einer `.term`-Datei vor, dann muss zunächst das Tool `erd2curry`[1] genutzt werden.

```
erd2curry erd.term
```

Das Tool erzeugt nun einige Dateien, wobei nur die Datei mit dem Namen `ERDName_ERDT.term` relevant ist. `ERDName` ist dabei der Name, der im `ERD` für das ER-Modell angegeben wurde. Nun wird das Modul `ERD2CDBI` geladen. Hier muss folgendes ausgeführt werden:

```
main "ERDName_ERDT.term"
```

`ERD2CDBI` erzeugt nun eine Datei namens `CDBIERDName.curry`, in der alle Datentypen enthalten sind, die für die Schnittstelle notwendig sind. Nun kann ein Programm geschrieben werden, dass die Schnittstelle nutzt, indem die Module `CDBI` und `CDBIERDName` importiert werden.

A.3. ERD der Universität

Listing A.1: ERD einer Universität vor dem Übersetzen mit `erd2curry`

```
ERD "Uni"
   [Entity "Student"
           [Attribute "MatNum" (IntDom Nothing) PKey False,
            Attribute "Name" (StringDom Nothing) NoKey False,
            Attribute "FirstName" (StringDom Nothing) NoKey False,
            Attribute "Email" (StringDom Nothing) Unique False,
            Attribute "Age" (IntDom Nothing) NoKey True],
    Entity "Lecture"
           [Attribute "Title" (StringDom Nothing) NoKey False,
            Attribute "Topic" (StringDom Nothing) NoKey True],
    Entity "Lecturer"
           [Attribute "Name" (StringDom Nothing) NoKey False,
            Attribute "FirstName" (StringDom Nothing) NoKey False],
    Entity "Place"
           [Attribute "Street" (StringDom Nothing) NoKey False,
            Attribute "StrNr" (IntDom Nothing) NoKey False,
            Attribute "RoomNr" (IntDom Nothing) NoKey False],
    Entity "Time"
           [Attribute "Time" (DateDom Nothing) Unique False],
    Entity "Exam"
           [Attribute "GradeAverage" (FloatDom Nothing) NoKey True],
    Entity "Result"
```

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

```
        [Attribute "Attempt" (IntDom Nothing) NoKey False,
         Attribute "Grade" (FloatDom Nothing) NoKey True,
         Attribute "Points" (IntDom Nothing) NoKey True]]
  [Relationship "Teaching"
                [REnd "Lecturer" "taught_by" (Exactly 1),
                 REnd "Lecture" "teaches" (Between 0 Infinite)],
   Relationship "Participation"
                [REnd "Lecture" "participates" (Between 0 Infinite),
                 REnd "Student" "participated_by" (Between 0 Infinite)],
   Relationship "Taking"
                [REnd "Result" "has_a" (Between 0 Infinite),
                 REnd "Student" "belongs_to" (Exactly 1)],
   Relationship "Resulting"
                [REnd "Exam" "result_of" (Exactly 1),
                 REnd "Result" "results_in" (Between 0 Infinite)],
   Relationship "Belonging"
                [REnd "Exam" "has_a" (Between 0 Infinite),
                 REnd "Lecture" "belongs_to" (Exactly 1)],
   Relationship "ExamDate"
                [REnd "Exam" "taking_place" (Between 0 Infinite),
                 REnd "Time" "at" (Exactly 1)],
   Relationship "ExamPlace"
                [REnd "Exam" "taking_place" (Between 0 Infinite),
                 REnd "Place" "in" (Exactly 1)]]
```

Listing A.2: ERD einer Universität nach dem Übersetzen mit `erd2curry`

```
ERD "Uni"
  [Entity "Participation"
          [Attribute "LectureParticipationKey" (KeyDom "Lecture")
           PKey False,
           Attribute "StudentParticipationKey" (KeyDom "Student")
           PKey False],
   Entity "Student"
          [Attribute "Key" (IntDom Nothing) PKey False,
           Attribute "MatNum" (IntDom Nothing) Unique False,
           Attribute "Name" (StringDom Nothing) NoKey False,
           Attribute "FirstName" (StringDom Nothing) NoKey False,
           Attribute "Email" (StringDom Nothing) Unique False,
           Attribute "Age" (IntDom Nothing) NoKey True],
   Entity "Lecture"
          [Attribute "Key" (IntDom Nothing) PKey False,
           Attribute "Title" (StringDom Nothing) NoKey False,
           Attribute "Topic" (StringDom Nothing) NoKey True,
           Attribute "LecturerTeachingKey" (KeyDom "Lecturer")
           NoKey False],
   Entity "Lecturer"
```

```
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "Name" (StringDom Nothing) NoKey False,
        Attribute "FirstName" (StringDom Nothing) NoKey False],
 Entity "Place"
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "Street" (StringDom Nothing) NoKey False,
        Attribute "StrNr" (IntDom Nothing) NoKey False,
        Attribute "RoomNr" (IntDom Nothing) NoKey False],
 Entity "Time"
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "Time" (DateDom Nothing) Unique False],
 Entity "Exam"
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "GradeAverage" (FloatDom Nothing) NoKey True,
        Attribute "LectureBelongingKey" (KeyDom "Lecture") NoKey False,
        Attribute "TimeExamDateKey" (KeyDom "Time") NoKey False,
        Attribute "PlaceExamPlaceKey" (KeyDom "Place") NoKey False],
 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]]

[Relationship "ExamPlace"
              [REnd "Exam" "taking_place" (Between 0 Infinite),
               REnd "Place" "in" (Exactly 1)],
 Relationship "ExamDate"
              [REnd "Exam" "taking_place" (Between 0 Infinite),
               REnd "Time" "at" (Exactly 1)],
 Relationship "Belonging"
              [REnd "Exam" "has_a" (Between 0 Infinite),
               REnd "Lecture" "belongs_to" (Exactly 1)],
 Relationship "Resulting"
              [REnd "Exam" "result_of" (Exactly 1),
               REnd "Result" "results_in" (Between 0 Infinite)],
 Relationship "Taking"
              [REnd "Result" "has_a" (Between 0 Infinite),
               REnd "Student" "belongs_to" (Exactly 1)],
 Relationship ""
              [REnd "Lecture" [] (Exactly 1),
               REnd "Participation" "participated_by"
               (Between 0 Infinite)],
 Relationship ""
              [REnd "Student" [] (Exactly 1),
```

```
                REnd "Participation" "participates"
                (Between 0 Infinite)],
    Relationship "Teaching"
                [REnd "Lecturer" "taught_by" (Exactly 1),
                 REnd "Lecture" "teaches" (Between 0 Infinite)]]
```

A.4. Beispiel von automatisch erzeugten Datentypen und Funktionen

Listing A.3: Alle notwendigen Datentypen und Funktionen für die Entität `Lecture`

```
data Lecture = Lecture LectureID String (Maybe String) LecturerID
data LectureID = LectureID Int
lectureDescription :: EntityDescription Lecture
lectureDescription = ED "Lecture"
    [SQLTypeInt, SQLTypeString, SQLTypeString, SQLTypeInt]
    (\(Lecture (LectureID key) title topic
      (LecturerID lecturerTeachingKey))
        -> [SQLInt key, SQLString title,
            (sqlStringOrNull topic), SQLInt lecturerTeachingKey])
    (\[SQLInt key, SQLString title, topic, SQLInt lecturerTeachingKey]
        -> Lecture (LectureID key) title (stringOrNothing topic)
        (LecturerID lecturerTeachingKey))
lectureTable :: Table
lectureTable = "Lecture"
lectureColumnKey :: Column Int
lectureColumnKey = Column "\"Key\"" "\"Lecture\".\"Key\""
lectureColumnTitle :: Column String
lectureColumnTitle = Column "\"Title\"" "\"Lecture\".\"Title\""
lectureColumnTopic :: Column String
lectureColumnTopic = Column "\"Topic\"" "\"Lecture\".\"Topic\""
lectureColumnLecturerTeachingKey :: Column Int
lectureColumnLecturerTeachingKey =
  Column "\"LecturerTeachingKey\""
  "\"Lecture\".\"LecturerTeachingKey\""
lectureKey (Lecture a _ _ _ ) = a
setLectureKey (Lecture _ b3 b2 b1 ) a = (Lecture a b3 b2 b1 )
lectureTitle (Lecture _ a _ _ ) = a
setLectureTitle (Lecture a2 _ b2 b1 ) a = (Lecture a2 a b2 b1 )
lectureTopic (Lecture _ _ a _ ) = a
setLectureTopic (Lecture a3 a2 _ b1 ) a = (Lecture a3 a2 a b1 )
lectureLecturerTeachingKey (Lecture _ _ _ a ) = a
setLectureLecturerTeachingKey (Lecture a4 a3 a2 _ ) a =
  (Lecture a4 a3 a2 a )
```

A.5. Beispielmodul

In einem kleinen Beispielmodul soll einmal eine etwas komplexere Anfrage gezeigt werden. Es wird zunächst der zusammengesetzte Datentyp `StudentStudentExam` implementiert und daraufhin eine Anfrage nach diesem Typ gestartet. Damit die Anfrage auf die Seite passt, wurde das `Constraint` ein wenig aufgeteilt. Dabei stehen stehen `ie` und `oe` für *inner constraint* und *outer constraint*.

```
import CDBIUni
import CDBI
import Time

cd = combineDescriptions studentDescription 0 examDescription 0
                         (\st ex -> (StudentStudentExam _ st ex))
                         (\(StudentStudentExam _ st ex) -> (st, ex))

data StudentStudentExam = StudentStudentExam Student Student Exam
sseDescription =
  addDescription studentDescription 1
  (\st1 (StudentStudentExam _ st2 ex) -> (StudentStudentExam st1 st2 ex))
  (\(StudentStudentExam st _ _) -> st) cd

-- Print StudentStudentExams where the first Student has ID = 1 and the
-- other Student wrote the exam with him ascending by examID
testFour :: IO ()
testFour = do
  conn <- connectSQLite "Uni.db"
  let ie = Exists resultTable 1
             (And [colNum resultColumnStudentTakingKey 1
                          .=. colNum studentColumnKey 1,
                   colNum resultColumnExamResultingKey 1
                          .=. col examColumnKey])
  let oe = Exists resultTable 0
             (And [col resultColumnStudentTakingKey
                       .=. int 1,
                   col resultColumnExamResultingKey
                       .=. col examColumnKey,
                   ie])
  let const = And [col studentColumnKey
                       ./=. colNum studentColumnKey 1,
                   col studentColumnKey
                       .=. int 1,
                   oe]
  result <- getEntriesCombined sseDescription
            (Criteria const
                      [ascOrder examColumnKey]) conn
  case result of
```

```
  Left err  -> putStrLn $ show err
  Right res -> printList res
putStrLn ""
```

Literaturverzeichnis

[BHM08] 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.

[Che76] Peter Pin-Shan Chen. The entity-relationship model toward a unified view of data. In *ACM Transactions on Database Systems 1*, pages 9–36, 1976.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. In *Communications of the ACM.*, pages 377–387. ACM Press, New York 13, 1970.

[Han13] 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.

[He12] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[MS93] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.