Christian-Albrechts-Universität zu Kiel

Diplomarbeit

Generierung von Curry-Programmen aus ER-Diagrammen

Marion Müller

28. Februar 2007

Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

betreut durch:
Prof. Dr. Michael Hanus
Dr. Bernd Braßel

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel, 28. Februar 2007

_____

Inhaltsverzeichnis

1. Einleitung

In der Praxis wird für die Implementierung einer Datenbank oft ein Werkzeug verwendet, um dem Programmierer Arbeit abzunehmen. Aus einer vorgegebenen Spezifikation, die beispielsweise grafisch als ER-Diagramm erstellt wird, werden dann automatisch die Tabellen der Datenbank generiert. Dazu und für den Zugriff auf die Daten wird meistens SQL verwendet, das von allen großen Datenbanksystemen verstanden wird.
Mit dem Werkzeug Hibernate[1] für Java implementiert man für jede Tabelle eine Java-Klasse mit speziellen Javadoc-Kommentaren, aus denen mithilfe von XDoclet[2] sogenannte Mapping-Dateien (XML) generiert werden, die Informationen über die Tabellen und ihre Beziehungen untereinander enthalten und aus denen automatisch die Tabellen der Datenbank generiert werden können. Diese Verbindung der Java-Objekte mit einer relationalen Datenbank nennt man objekt-relationales Mapping (ORM). Der Zugriff auf die Daten ist dadurch direkt über die Java-Objekte möglich.
Auch für Haskell gibt mit HaSQL[3] und HaskellDB [7] schon Implementierungen, mit denen der Zugriff auf relationale Datenbanken realisiert wird. HaskellDB bietet Kombinatoren, um Datenbankabfragen mit relationaler Algebra zu erstellen. Diese werden automatisch in SQL-Abfragen übersetzt.

Diese Diplomarbeit hat das Ziel, die Implementierung einer Datenbank mit funktional-logischer Programmierung in Curry[4] durch automatische Codegenerierung zu unterstützen. Curry ist für die Implementierung einer Datenbank besonders gut geeignet, weil die Aussagen der Logikprogrammierung eine natürliche Schnittstelle zu Datenbanken sind und somit den Zugriff auf einem hohen Niveau ermöglichen. Zusammen mit der funktionalen Programmierung und dem Modulsystem von Curry lässt sich alles in einer Sprache implementieren, d.h. interne Hilfsbibliotheken, die Codegenerierung, der generierte Code selber und alle späteren Erweiterungen des Benutzers sind in Curry geschrieben. Das vereinfacht die Einarbeitung für den Benutzer.

ER-Diagramme sind für den grafischen Datenbank-Entwurf weit verbreitet und einfach zu verstehen. Auch gibt es viele Werkzeuge, um sie zu erstellen. Deswegen werden ER-Diagramme hier verwendet. Dem Programmierer soll möglichst viel und besonders fehleranfällige Programmierung durch automatische Generierung von erweiterbaren Curry-Programmen direkt aus grafisch erstellten ER-Diagrammen abgenommen werden.

[1] `http://www.hibernate.org`
[2] `http://xdoclet.sourceforge.net`
[3] `http://members.tripod.com/~sproot/hasql.htm`
[4] `http://www.informatik.uni-kiel.de/~curry`

Die einzelnen Aufgaben sind also:

- Erstellung einer Spezifikation für ER-Diagramme und deren Umsetzung in Curry
- Verwendung eines grafischen Werkzeuges zur Erstellung von ER-Diagrammen und Übersetzung in Curry
- Generierung von Datenbank-Operationen
- Generierung von Konsistenztests

In Kapitel 2 werden die Programmiersprache Curry und das ER-Modell zusammen mit einer Curry-Darstellung vorgestellt. Kapitel 3 begründet die Wahl des verwendeten Werkzeuges und gibt eine Einführung in dessen Benutzung. Kapitel 4 beschreibt die einzelnen Schritte der Implementierung. Dabei werden Vorüberlegungen und die eigentliche Codegenerierung getrennt behandelt. Ein Anwendungsbeispiel für den generierten Code wird gegeben. Kapitel 5 fasst die Ergebnisse dieser Arbeit zusammen und bietet einen Ausblick auf zukünftige Arbeiten.

2. Grundlagen

2.1. Curry

In diesem Abschnitt soll eine kleine Einführung in Curry gegeben werden im Hinblick auf die hier verwendeten Elemente der Sprache. Alle Einzelheiten werden in [2] genau beschrieben. Als Interpreter wird PAKCS (Portland Aachen Kiel Curry System) [5] verwendet, der die Curry-Programme in SICStus Prolog übersetzt.

Ein Curry-Programm ist ein funktionales Programm in Haskell-ähnlicher Syntax erweitert um die Möglichkeit, freie (logische) Variablen zu verwenden. Es besteht also aus Datentyp- und Funktionsdefinitionen.

2.1.1. Datentypen

Datentypen sind die Grundlage für Berechnungen in Funktionen. Definiert wird ein Datentyp durch das Schlüsselwort `data`, einen Namen, eventuell Typparametern für polymorphe Typen und den durch `|` getrennten Konstruktoren.

```
data Bool    = True    | False
data Maybe a = Nothing | Just a
data [a]     = []      | a : [a]
```

Der Typ `Bool` für boole'sche Werte ist ein Aufzählungstyp mit den beiden Konstruktoren `True` und `False`. Optionale Werte lassen sich durch den polymorphen Typ `Maybe` darstellen. Die Typvariable `a` steht für alle verfügbaren Typen. Alle Typen können also mit `Maybe` zu einem Typ für optionale Werte werden. Gibt es einen Wert `x`, so ist der Wert des Ausdrucks `Just x`, sonst `Nothing`. Der rekursive Typ `List` implementiert eine Liste. Üblicherweise wird der Listentyp als `[a]` geschrieben. Entweder ist eine Liste leer (`[]`) oder sie enthält ein Element und eine Restliste. Der Listenkonstruktor `(:)` hängt ein Element vorne an eine Liste an. Als abkürzende Schreibweise kann man statt der Liste `0:(1:(2:[]))` auch `[0,1,2]` verwenden.

Typsynonyme werden durch das Schlüsselwort `type` definiert und dienen zur Abkürzung bereits bestehender Typen, zur besseren Lesbarkeit des Programms und zur Austauschbarkeit einzelner Typen ohne sonst viel im Programm ändern zu müssen. Beispielsweise ist das Typsynonym

```
type MyList = [Maybe Int]
```

eine Abkürzung für eine Liste, die Werte vom Typ `Maybe Int` enthält. Strings werden in Curry als Typsynonym

```
type String = [Char]
```

definiert. Dadurch können alle Funktionen für Listen auch für Strings verwendet werden.

2.1.2. Funktionen

Eine Funktionsdefinition besteht aus einer optionalen Typdeklaration, die die Typen der Parameter und des Ergebnisses der Funktion angibt, und einer Menge von Regeln. Jede Regel hat auf der linken Seite Pattern, durch die festgelegt wird, für welchen Fall die jeweilige Regel gültig ist. Die rechte Seite einer Regel ist ein Ausdruck. Beim Aufruf einer Funktion wird ein „Pattern-Matching“ der Parameter gemacht, d.h. es wird geprüft, welche Regeln für die Parameter passen. Es kann passieren, dass mehr als eine Regel passt. Die Funktion ist dann nicht-deterministisch und es kann mehrere Lösungen geben. Dies ist ein Unterschied zu Haskell, wo immer die erste passende Regel genommen wird.

Die Funktion

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

nimmt zwei Listen des gleichen Typs und hängt diese hintereinander. Da sie für polymorphe Listen implementiert ist, kann sie beispielsweise auch für die Konkatenation von Strings verwendet werden.

Curry erlaubt Funktionen höherer Ordnung, d.h. Funktionen können als Parameter übergeben werden. Beispiele dafür sind die oft verwendeten Funktionen

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Jedes Element einer Liste von `Int`-Werten kann man dann leicht inkrementieren durch die partielle Applikation von `map`:

```
incList :: [Int] -> [Int]
incList = map (1+)
```

Mit `foldr` lassen sich die Elemente einer Liste durch eine Funktion mit einem neutralen Element verknüpfen.

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

berechnet die Summe aller Elemente einer `Int`-Liste. Die Funktion `(++)` von oben lässt sich nun auch so implementieren:

```
xs ++ ys = foldr (:) ys xs
```

Wie in Haskell kann man Funktionen auch mit Guards definieren, die es erlauben, beliebige boole'sche Bedingungen auf die linke Seite einer Regel zu schieben. Die Guards werden der Reihe nach ausgewertet. Die rechte Seite des ersten erfüllten Guards wird ausgeführt. Die Funktion zur Berechnung der Fakultät lässt sich mit oder ohne Guards wie folgt implementieren, wobei `otherwise` eine Funktion ist, die zu `True` ausgewertet wird.

```
fac n | n == 0    = 1
      | otherwise = n * fac (n-1)

fac n = if n == 0 then 1 else n * fac (n-1)
```

In Curry gibt es auch Funktionen mit nur einem Guard, der dann ein *Constraint* vom Typ `Success` ist.

Constraints werden zu `success` ausgewertet, wenn Bindungen für alle freie Variablen gefunden werden, so dass das Constraint erfüllt ist. Prädikate der logischen Programmierung können als Funktionen mit Ergebnistyp `Success` betrachtet werden. Ein Prädikat, das erfüllt ist, falls sein Argumente eine Mutter mit ihrem Kind ist, kann durch die folgende Funktion mit einigen Fakten für dieses Prädikat implementiert werden.

```
istMutterVon :: String -> String -> Success
istMutterVon "Lieselotte" "Monika" = success
istMutterVon "Monika" "Marion" = success
istMutterVon "Monika" "Michael" = success
istMutterVon "Monika" "Martin" = success
istMutterVon "Marion" "Kalle" = success
```

Das folgende Prädikat ist erfüllt für alle Großmütter und ihre Enkelkinder.

```
istGroßmutterVon :: String -> String -> Success
istGroßmutterVon g e = istMutterVon g m & istMutterVon m e where m free
```

Hier werden die Constraints `(istMutterVon g m)` und `(istMutterVon m e)` durch die Konjunktion `(&)` kombiniert, die dadurch beide nebenläufig berechnet werden. Lokale Funktions- oder Variablendefinitionen werden mit dem Schlüsselwort `where` eingeleitet. So lassen sich auch freie Variablen für logische Programmierung definieren, z.B. im Aufruf der Funktion `istGroßmutterVon`.

```
> istGroßmutterVon "Monika" e where e free
Free variables in goal: e
Result: success
Bindings:
e="Kalle" ? ;
No more solutions.
```

Die Ein- und Ausgabe wird wie in Haskell durch die IO-Monade realisiert. Wichtig ist die Reihenfolge der IO-Aktionen. Mit den Funktionen

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>) :: IO a -> IO b -> IO b
```

können Aktionen sequentialisiert werden. Wenn man nicht an Zwischenergebnissen interessiert ist, verwendet man `(>>)`, z.B. für die Ausgabe von Zeichen mit `putChar`, einer IO-Aktion, die als Argument ein Zeichen von Typ `Char` bekommt.

```
putChar 'a' >> putChar 'b'
```

Werden Zwischenergebnisse in der weiteren Berechnung verwendet, so benutzt man dafür `(>>=)`. Die folgende IO-Aktion liest ein Zeichen ein und gibt dieses wieder aus.

```
getChar >>= putChar
```

Als vereinfachte Schreibweise wird die do-Notation für die IO-Monade unterstützt. Statt `(>>=)` und `(>>)` werden Sequenzen verwendet. Das Beispiel von oben wird in der do-Notation so geschrieben:

```
do c <- getChar
   putChar c
```

2.1.3. Module

Curry-Programme können in Modulen organisiert werden. Ein Modul `M` wird in einer Datei `M.curry` gespeichert. Es enthält Datentyp- und Funktionsdefinitionen, die exportiert werden können, um in anderen Modulen importiert zu werden. Was nicht exportiert wird, ist nur lokal in einem Modul verfügbar. Um abstrakte Datentypen zu definieren, exportiert man nur den Datentyp ohne die Konstruktoren.
Funktionsnamen müssen innerhalb eines Moduls eindeutig sein, können aber in anderen Modulen mit ihrem qualifizierten Namen, d.h. dem Modulnamen und dem Funktionsnamen durch einen Punkt verbunden, unterschieden werden.
Das Modul

```
module M (D1, D2(..), f) where
import List

data D1 = C1 | C2

data D2 = C3 | C4 | C5

f :: [a] -> Bool
f xs = length xs == length (nub xs)
```

hat den Namen `M` und exportiert die Datentypen `D1` und `D2`, aber nur die Konstruktoren von `D2`, und die Funktion `f`, die mit der Funktion `nub` aus dem importierten Modul `List` prüft, ob eine Liste doppelte Elemente enthält.

2.2. Dynamische Prädikate

Die Prädikate der funktional-logischen Programmierung ermöglichen den Datenbank-Zugriff auf einem hohen Niveau. Dafür ist es notwendig, dass Daten dauerhaft über mehrere Programmläufe hinweg erhalten bleiben. Ein Prädikat ist

- *dynamisch*, wenn zur Laufzeit Fakten hinzugefügt und entfernt werden können, und
- *persistent*, wenn es dynamisch ist und seine Fakten extern gespeichert werden, um mehrere Programmläufe zu überdauern.

Die Curry-Bibliothek `Dynamic` [6] bietet eine Implementierung für die Speicherung von dynamischen Prädikaten in Dateien, die in dieser Arbeit verwendet wird. Eine Erweiterung der Bibliothek auf eine relationale Datenbank zur Speicherung der Daten wird in [3] vorgestellt.

Ein dynamisches Prädikat ist eine Menge von Fakten, die nicht im Programm definiert werden und von denen angenommen wird, dass sie keine freien Variablen enthalten. Definiert wird ein dynamisches Prädikat im Programm nur durch die Typsignatur, das Schlüsselwort `persistent` und den Namen des Verzeichnisses, in dem die Daten gespeichert werden sollen. In Abschnitt 2.1.2 wurde das Prädikat `istMutterVon` definiert. Die persistente Version davon ist

```
mutterVon :: Int -> Dynamic
mutterVon persistent "file:mutterVonDB"
```

Um neue Fakten hinzufügen zu können, wird eine Aktion benötigt.

```
assert :: Dynamic -> IO ()
```

tut dies für Argumente ohne freie Variablen, sonst suspendiert es. Auch das Entfernen von Fakten ist mit einer Aktion möglich:

```
retract :: Dynamic -> IO Bool
```

Der Typ `Bool` im Ergebnis der Aktion zeigt, ob etwas entfernt wurde oder nicht. Der Zugriff auf die für ein dynamisches Prädikat gespeicherten Fakten wird durch die Aktion

```
getDynamicSolutions :: (a -> Dynamic) -> IO [a]
```

realisiert.

Ein Beispiel ist der folgende Programmausschnitt:

```
do assert (istMutterVon "Lieselotte" "Monika")
   assert (istMutterVon "Monika" "Marion")
   assert (istMutterVon "Lieselotte" "Kalle")
   m1 <- getDynamicSolutions (\(m,k) -> istMutterVon m k)
   retract (istMutterVon "Lieselotte" "Kalle")
   assert (istMutterVon "Monika" "Michael")
   assert (istMutterVon "Monika" "Martin")
   assert (istMutterVon "Marion" "Kalle")
   m2 <- getDynamicSolutions (\(m,k) -> istMutterVon m k)
   return (m1, m2)
```

Führt man diese Aktionen aus, so erhält man

```
([("Lieselotte","Monika"),("Monika","Marion"),("Lieselotte","Kalle")],
 [("Lieselotte","Monika"),("Monika","Marion"),("Monika","Michael"),
  ("Monika","Martin"),("Marion","Kalle")]).
```

2.2.1. Transaktionen

Eine *Transaktion* ist eine Folge von Datenbank-Operationen, die hinsichtlich gewisser Integritätsanforderungen als atomar anzusehen sind, d.h. Transaktionen müssen entweder vollständig oder gar nicht ausgeführt werden. Nur nach einer erfolgreichen Transaktion dürfen die Daten verändert sein. Damit soll erreicht werden, dass die Datenbank vor und nach der Ausführung einer Transaktion in einem konsistenten Zustand ist. Für Transaktionen werden zwei Aktionen bereitgestellt.

```
transaction :: IO a -> IO (Maybe a)

abortTransaction :: IO a
```

Die Funktion `transaction` nimmt eine IO-Aktion als Argument, die Änderungen von dynamischen Prädikaten enthält, und startet eine Transaktion. Das Ergebnis benutzt den Typ `Maybe`, um zwischen erfolgreichen und fehlgeschlagenen Transaktionen zu unterscheiden. Eine Transaktion kann mit `abortTransaction` abgebrochen werden. So kann man selbst entscheiden, in welchen Fällen eine Transaktion fehlschlagen soll.

2.2.2. Kombination dynamischer Prädikate

Um komplexe Anfragen zu formulieren ist es nützlich, mehrere dynamische Prädikate zu kombinieren. Mit der Funktion

```
(<>) :: Dynamic -> Dynamic -> Dynamic
```

lassen sich zwei dynamische Prädikate verbinden. Zur Einschränkung von dynamischen Prädikaten gibt es zwei Möglichkeiten.

```
(|>) :: Dynamic -> Bool -> Dynamic

(|&>) :: Dynamic -> Success -> Dynamic
```

Dazu kann man entweder mit `(|>)` eine boole'schen Bedingung oder mit `(|&>)` ein Constraint verwenden. Wichtig für die Programmierung mit diesen Kombinatoren ist, dass `(<>)` stärker bindet als `(|>)` und `(|&>)`.

Das Beispiel von oben kann man nun auch so schreiben:

```
do assert (istMutterVon "Lieselotte" "Monika"
           <> istMutterVon "Monika" "Marion"
           <> istMutterVon "Lieselotte" "Kalle")
   m1 <- getDynamicSolutions (\(m,k) -> istMutterVon m k)
```

```
retract (istMutterVon "Lieselotte" "Kalle")
assert (istMutterVon "Monika" "Michael"
         <> istMutterVon "Monika" "Martin"
         <> istMutterVon "Marion" "Kalle")
m2 <- getDynamicSolutions (\(m,k) -> istMutterVon m k)
return (m1, m2)
```

Ebenso kann man mehrere Fakten kombinieren, die gelöscht werden sollen, oder diese durch Bedingungen einschränken. Der folgende Aufruf liefert Paare von Großmüttern und Enkeln, für die eine Einschränkung gilt:

```
getDynamicSolutions (\(g,e) ->
  let m free
  in
  istMutterVon g m <> istMutterVon m e |> head e == 'M')
```

Damit kann man auch komplexe Anfragen stellen. In SQL entspricht `(g,e)` der Projektion (`SELECT`), `(istMutterVon g m <> istMutterVon m e)` enthält die beteiligten Relationen (`FROM`) und die Bedingung `(head e == 'M')` schränkt das Ergebnis ein (`WHERE`).

2.3. Das Entity/Relationship-Modell

Eine der am weitesten verbreiteten Methoden zur konzeptuellen Beschreibung von Datenbanken stellt das Entity/Relationship-Modell (ER-Modell) dar, das auf [1] zurückgeht.

Bei diesem Modell geht es darum, die reale Welt durch Entitäten (Entities) und den zwischen ihnen bestehenden Beziehungen (Relationships) grafisch darzustellen. Aufgrund der grafischen Struktur des ER-Modells eignet es sich besonders gut zum Entwurf von Datenbanken. Stellt man ein ER-Modell grafisch dar, so spricht man auch von einem Entity/Relationship-Diagramm (ER-Diagramm).

Abbildung 2.1 zeigt ein Beispiel für ein ER-Diagramm. Ein Entitätstyp legt die Eigenschaften von Entitäten des gleichen Typs fest. Zu einem Entitätstyp gehört ein eindeutiger Bezeichner und eine Menge von Attributen, die die Eigenschaften der Entitäten festlegen. Jedem Attribut wird ein eindeutiger Wertebereich (Domain) zugeordnet. Um die Entitäten eindeutig identifizieren zu können, werden ein oder mehrere Attribute aus der Attributmenge oder ein zusätzliches sogenanntes künstliches Attribut als Schlüssel bestimmt.

Ein Beziehungstyp beschreibt eine Beziehung zwischen mindestens zwei Entitätstypen. Im allgemeinen ER-Modell können auch mehrere Entitäten miteinander in Beziehung stehen. Zu einem Beziehungstyp gehört neben einem Bezeichner für jeden beteiligten Entitätstypen eine Rolle, die die eindeutige Zuordnung der Entitätstypen ermöglicht.

Abbildung 2.1.: ER-Diagramm

Repräsentiert wird ein ER-Diagramm durch den folgenden Curry-Datentyp.

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

Ein ER-Diagramm besteht aus einem Namen als String, einer Menge von Entitäten und einer Menge von Beziehungen zwischen diesen Entitäten, die jeweils in einer Liste gespeichert werden.

```
data Entity = Entity EName [Attribute]

type EName = String

data Attribute = Attribute AName Domain Key Null

type AName = String

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 EName

data Key = NoKey
         | PKey
         | Unique

type Null = Bool
```

Eine Entität hat einen Namen als String und eine Liste von Attributen. Jedes Attribut hat einen Namen als String, einen Wertebereich, Angaben über Schlüsseleigenschaften und ob sein Wert weggelassen bzw. undefiniert (Null) sein darf. Der Datentyp `Domain` ordnet einem Attribut einen Wertebereich zu, der einer der Curry-Typen `Int`, `Float`, `Char`, `String`, `Bool` oder `ClockTime` oder auch ein benutzerdefinierter Typ (`UserDefined`) sein kann. Außerdem ermöglicht er, einen optionalen Standardwert für einen Typ anzugeben. Der Konstruktor `KeyDom` ist für die in Abschnitt 4.3 beschriebene Transformation des ER-Diagramms vorgesehen. So werden intern verwendete Fremdschlüssel zusammen mit ihrer Herkunft gekennzeichnet.
Ein Attribut ist entweder kein Schlüssel (`NoKey`), Teil des Primärschlüssels (`PKey`) oder eindeutig (`Unique`).

```
data Relationship = Relationship RName [REnd]

type RName = String

data REnd = REnd EName Role Cardinality

type Role = String

data Cardinality = Exactly Int
                 | Range Int (Maybe Int)
```

Eine Beziehung hat einen Namen und eine Liste von Verbindungen zu Entitäten (`REnd`), die in Beziehung zueinander stehen. Die Anzahl der Entitäten, die an einer Beziehung beteiligt sind, wird als *Grad* einer Beziehung bezeichnet. Für jede beteiligte Entität wird ihr Name, ein Rollenbezeichner (`Role`) und zur Beschreibung der Komplexität einer Beziehung die Kardinalität (`Cardinality`) gespeichert. Die Kardinalität ist entweder eine bestimmte Zahl (z.B. `(Exactly 1)` für die Zuordnung genau einer Entität) oder ein Bereich (z.B. `(Range 1 (Just 4))` für den Bereich von 1 bis 4 oder `(Range 0 Nothing)` für den Bereich von 0 bis beliebig viele). Man kann Beziehungen grob in drei Typen unterteilen:

- einfach-einfache Beziehungen (1:1)
- einfach-komplexe Beziehungen (1:n)
- komplex-komplexe Beziehungen (n:m)

In Abbildung 2.1 ist beispielsweise ein Entitätstyp mit Namen „Student“ und den Attributen „MatrikelNr“, „Name“, „Vorname“ und „Email“ dargestellt. Durch Unterstreichen wird das Attribut „MatrikelNr“ als Schlüssel gekennzeichnet. Die anderen Eigenschaften wie z.B. Wertebereiche sind in der Grafik nicht sichtbar. Die Entitätstypen „Student“ und „Veranstaltung“ sind über die n:m-Beziehung „Teilnahme“ miteinander verbunden, d.h.

ein Student darf an beliebig vielen Veranstaltungen teilnehmen und eine Veranstaltung darf von beliebig vielen Studenten besucht werden.

3. Auswahl eines Werkzeuges

Das Ziel dieser Arbeit ist, die Programmierung einer Datenbank in Curry zu unterstützen. Zur Eingabe der Daten soll ein existierendes Werkzeug verwendet werden. Gesucht ist also ein grafischer Editor für den konzeptionellen Entwurf einer Datenbank unter Verwendung des ER-Modells, der die dort üblichen Symbole bereit stellt, so dass Diagramme direkt am Rechner gezeichnet werden können.

Die Auswahl solcher Werkzeuge ist groß. Viele sind kommerziell, aber es gibt auch eine Reihe frei verfügbarer. Meist sind diese sogenannten CASE-Tools (Computer Aided Software Engineering-Tools) konzipiert für den Entwurf und oft auch für die Erzeugung und Verwaltung einer Datenbank mit einem oder mehreren speziellen Datenbank-Managementsystemen (DBMS). Es ist dann nicht mehr notwendig, ein sogenanntes SQL-DDL-Skript zu schreiben, d.h. eine Beschreibung der Datenbank in der SQL Data Definition Language, einem Teil von SQL zum Anlegen von Tabellen. Ebenso soll es hier nicht mehr notwendig sein, alle Datenstrukturen und Funktionen, die für eine Datenbank gebraucht werden, von Hand zu schreiben. So weit möglich sollen diese automatisch generiert werden.
Speichern lässt sich ein Entwurf abhängig von dem verwendeten Werkzeug in verschiedenen XML- oder Textformaten. Leider gibt es keine standardisierte textuelle Repräsentation für ER-Diagramme. SQL wurde zwar als ANSI-Standard definiert, aber die Hersteller der DBMS implementieren SQL nicht streng nach diesem Standard, so dass es zwischen den einzelnen Systemen Unterschiede im Sprachumfang von SQL gibt. Daten im XML-Format sind leichter weiterzuverarbeiten als anderer Text. Gespeichert wird von den Werkzeugen oft nicht nur das reine ER-Diagramm, sondern zusätzlich für die Codegenerierung überflüssige Layout-Daten.

3.1. Beurteilung einiger Werkzeuge

In diesem Abschnitt sollen verschiedene Werkzeuge genannt und ihre Eignung für die Erstellung von ER-Diagrammen nach den folgenden nach Relevanz geordneten Kriterien beurteilt werden:

1. Die Erstellung von ER-Diagrammen muss möglich sein.

2. Das Werkzeug muss kostenfrei sein.

3. Die Erfassung von Eigenschaften wie Wertebereiche oder Schlüsseleigenschaften für Attribute muss unterstützt werden.
4. Die Erfassung von Kardinalitäten für Beziehungen muss unterstützt werden.
5. Das Speicherformat muss zur Weiterverarbeitung geeignet sein.
6. Lauffähigkeit unter Linux und nicht nur unter Windows ist erforderlich.
7. Keine Spezialisierung auf bestimmte DB-Systeme und damit eventuell verbundene automatische Anpassung des Diagramms schon bei der Erstellung.
8. Benutzerfreundlichkeit
9. Ein lebendes Projekt wird bevorzugt.
10. Eine weite Verbreitung ist wünschenswert.

Der Oracle Designer ist eines der bekanntesten kostenpflichtigen Werkzeuge. Außerdem gibt es beispielsweise noch den PowerDesigner von Sybase oder ERWIN von Computer Associates.
Die vier folgenden Werkzeuge sind alle kostenfrei und ermöglichen die Erstellung von ER-Diagrammen.

Das Werkzeug Dia[1] ist weit verbreitet und für alle gängigen Betriebssysteme verfügbar. Die Darstellung der Diagramme ist allerdings nicht sehr benutzerfreundlich, denn jedes Attribut wird in einem eigenen Kreis dargestellt, der mit dem zugehörigen Entitäts-Rechteck verbunden werden muss. Das ist umständlich und bei einem großen ER-Diagramm mit vielen Entitäten und Attributen unübersichtlich. Beziehungen werden durch Rauten dargestellt. Eine einfache Linie, die mit dem Namen der Beziehung beschriftet werden kann, würde ausreichen und wäre übersichtlicher.
Für Attribute ist keine Angabe des Wertebereichs möglich. Das ist aber sehr wichtig für die automatische Codegenerierung.
Dia ist eher ein reines Zeichen-Werkzeug und die XML-Ausgabe enthält nicht nur die Information über das ER-Diagramm, sondern auch viel überflüssige Information über das Layout der Grafik, die hier nicht benötigt wird.
Aus diesen Gründen ist Dia nur eingeschränkt geeignet.

Der DBDesigner4[2] unterstützt die Speicherung in SQL oder einem eigenen XML-Format. Für die SQL-Ausgabe sind verschiedene vom Benutzer wählbare Optionen möglich, die für verschiedene DBMS passend sind. Das weist schon auf die Spezialisierung dieses Werkzeuges auf bestimmte wenn auch mehrere DBMS hin. Auch werden beim Erstellen eines Diagramms automatische Anpassungen vorgenommnen, z.B. für Beziehungen automatisch Fremdschlüssel in die beteiligten Relationen eingetragen und bei komplex-komplexen Beziehungen eine zusätzliche Relation eingefügt.

[1] `http://www.gnome.org/projects/dia`
[2] `http://www.fabforce.net/dbdesigner4`

Die Zukunft des DBDesigner4 ist ungewiss, da er nicht weiterentwickelt wird und bald durch seinen Nachfolger MySQL-Workbench ersetzt werden soll. Der Download ist eventuell nur noch möglich, bis der Nachfolger fertiggestellt ist.
Dadurch kommt dieses Werkzeug nicht in Frage.

Das Tool DBMain[3] gibt es nur in einer eingeschränkten Version frei, und außerdem nur für Windows. Deswegen ist es ungeeignet.

Umbrello (Umbrello UML Modeller)[4] ist ein UML-Werkzeug für KDE, das ab Version 1.4 (KDE 3.4) auch ER-Diagramme unterstützt. Die Daten werden im XMI-Format gespeichert, das ein XML-Format ist und in Abschnitt 3.3 näher beschrieben wird. Der Export eines Diagramms als Grafik ist auch möglich. Als Teil von KDE wird Umbrello bei Sourceforge als Open-Source-Implementierung laufend weiterentwickelt. Es ist jedem leicht verfügbar und wird viel verwendet. Zusätzlich zur grafischen Darstellung sind Informationen wie Wertebereiche, Schlüsseleigenschaften oder Standardwerte für Attribute erfassbar. Die Unterstützung von benutzerdefinierten Wertebereichen hält die ER-Modellierung mit Umbrello flexibel. Ein Ziel der Entwickler ist, durch die Verwendung von XMI als Speicherformat den Datenaustausch mit anderen Werkzeugen zu ermöglichen.
Aufgrund dieser Eigenschaften ist Umbrello am besten geeignet.

3.2. Erstellen eines ER-Diagramms mit Umbrello

In diesem Abschnitt soll die Erstellung eines ER-Diagramms mit Umbrello 1.5.52 anhand eines kleinen Beispiels beschrieben werden.
Umbrello läßt sich unter KDE durch das Kommando umbrello oder unter Entwicklung im Menü öffnen. Abbildung 3.1 zeigt Umbrello nach dem Start. Das Hauptfenster ist aufgeteilt in die Baumansicht, den Arbeitsbereich und das Dokumentationsfenster. Die Baumansicht befindet sich am oberen linken Rand des Fensters und zeigt für die ER-Modellierung die Datentypen, Entitäten und Attribute. Im Arbeitsbereich wird ein erstelltes Diagramm grafisch dargestellt und kann bearbeitet werden. Das Dokumentationsfenster ist am unteren linken Rand des Fensters.

Außer Menüs und Werkzeugleisten nutzt Umbrello sehr stark über die rechte Maustaste erreichbare Kontextmenüs. Man kann auf jedes Element im Umbrello-Arbeitsbereich oder der Baumansicht mit der rechten Maustaste klicken, um für das gewählte Element sinnvolle Funktionen zu erreichen.

Durch Rechtsklicken auf „Entity Relationship Model → New → Entity Relationship Diagram“ in der Baumansicht legt man zuerst ein neues ER-Diagramm an, für das man direkt nach einem Namen gefragt wird, der später automatisch als Modulname verwendet wird.

[3] `http://www.db-main.be`
[4] `http://uml.sourceforge.net`

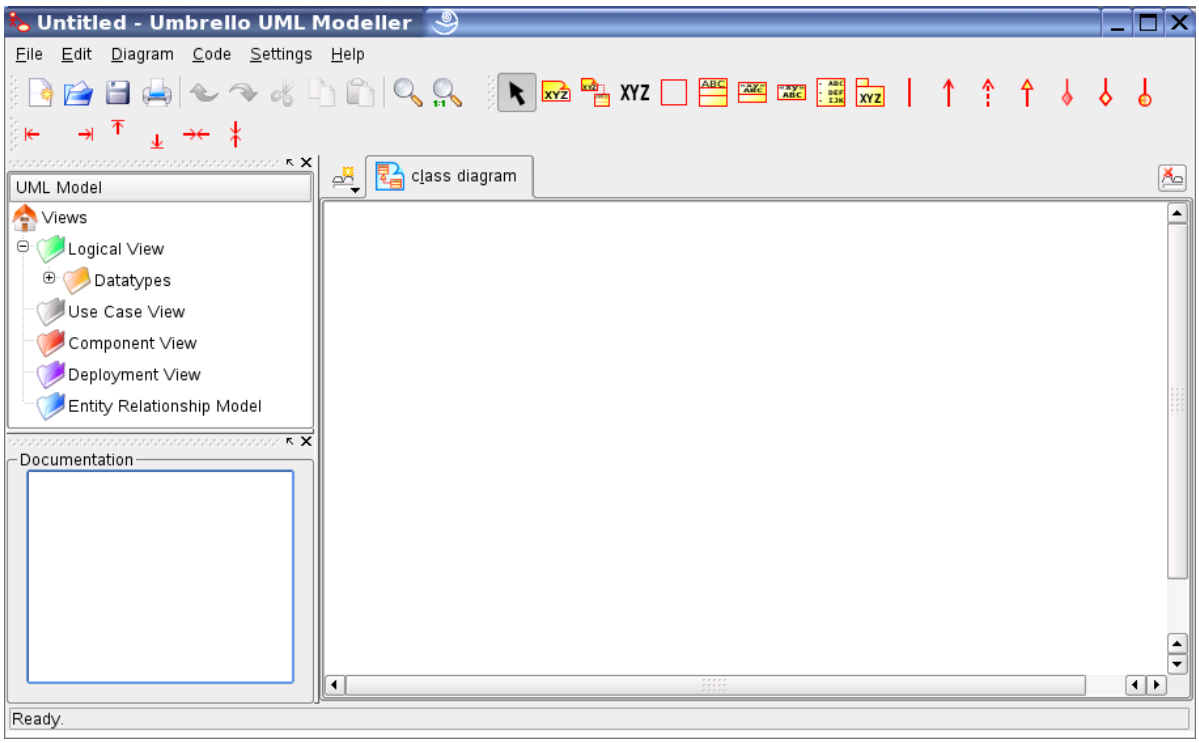

Abbildung 3.1.: Ansicht nach Start

Nun ändert sich die Werkzeugleiste am oberen Rand, so dass nur noch die Erstellung von Elementen des ER-Modells möglich ist (siehe Abbildung 3.2). Eine neue Entität kann man durch Auswahl des markierten Menüpunktes und Klicken an die gewünschte Stelle anlegen. Dabei wird man automatisch nach einem Namen der Entität gefragt. Es gibt zwei Möglichkeiten, Attribute zu einer bestehenden Entität hinzuzufügen: Durch Rechtsklicken auf die entsprechende Entität in der Grafik und der Wahl von „New → Entity Attribute“ im Kontextmenü läßt sich ein Attribut hinzufügen oder durch Doppelklicken auf den Namen der Entität öffnet sich ein Fenster zu Verwaltung aller Attribute einer Entität. Für ein Attribut kann man Eigenschaften festlegen, wie in Abbildung 3.3 zu sehen ist.

- Für den Wertebereich kann man einen der Basistypen von Curry (`Int`, `Float`, `Char`, `String`, `Bool`) oder einen benutzerdefinierten Datentypen direkt in das Auswahlfeld schreiben. Dabei ist für die Code-Generierung zu beachten, dass dieser Typ dann in der Form *Modulname.Typ* angegeben werden muss.
- Ein Attributname ist notwendig.
- Ein Standardwert aus dem gewählten Wertebereich ist optional. Das Feld kann auch freigelassen werden.
- Die Entscheidung, ob Nullwerte erlaubt sind.

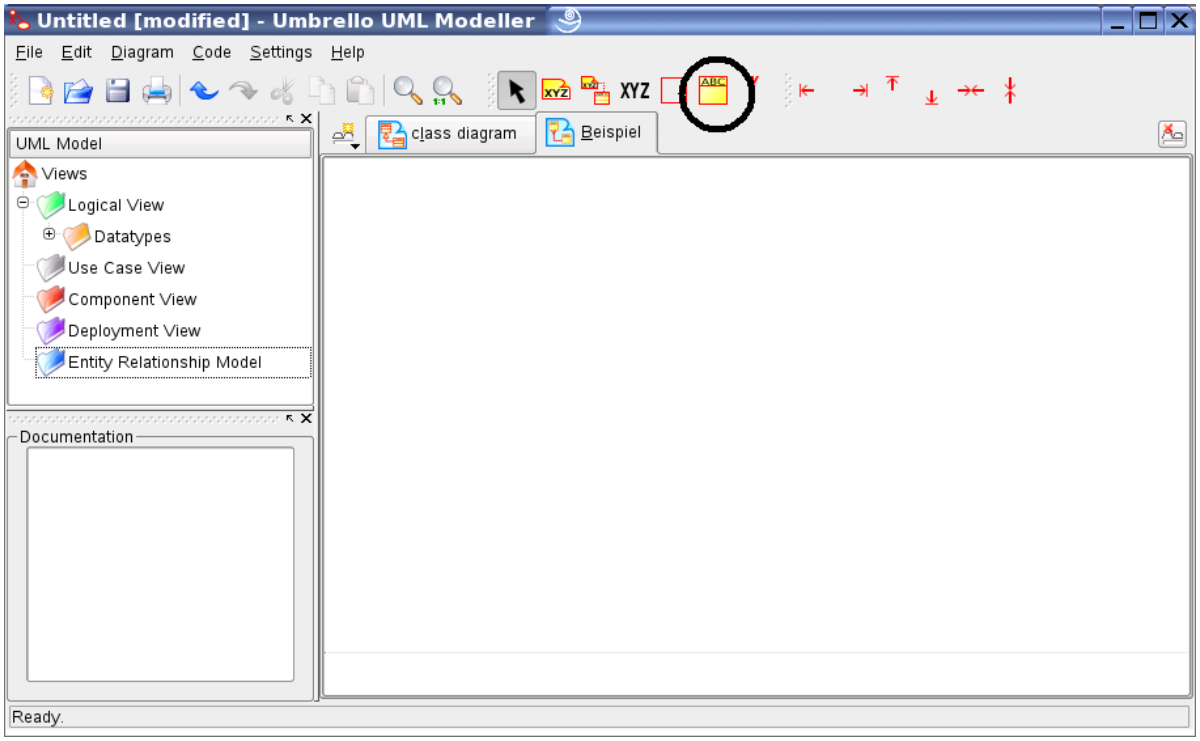

Abbildung 3.2.: Leeres ER-Diagramm

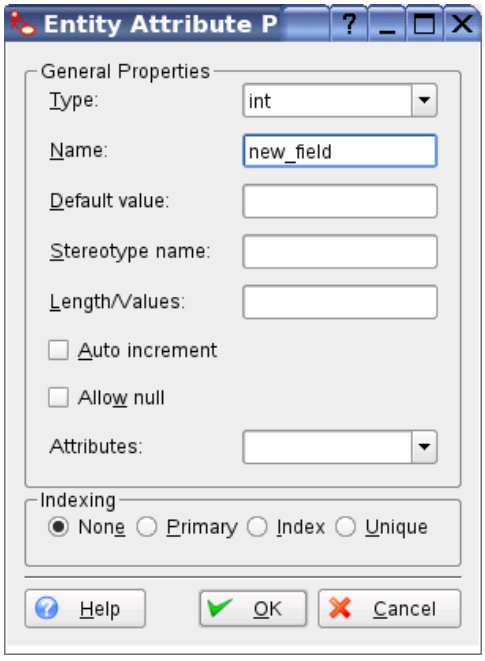

Abbildung 3.3.: Attribut-Eigenschaften

- Die Indizierung: keine, Teil des Primärschlüssels, Eindeutigkeit

Die restlichen Angaben werden nicht ausgewertet.
In der Grafik werden Primärschlüssel unterstrichen dargestellt, alle anderen Eigenschaften eines Attributes sind nur im Attribut-Menü sichtbar.

Zu beachten ist, dass Elemente in der Baumansicht über den Eintrag Löschen aus dem Kontextmenü gelöscht werden müssen, um sie aus dem Modell zu entfernen. Löscht man ein Element direkt im Diagramm, so wird es nicht aus dem Modell entfernt.

Durch Wahl des markierten Menüpunktes lassen sich Beziehungen zwischen je zwei Entitäten erstellen, indem man die zu verbindenden Entitäten nacheinander anklickt. Die Reihenfolge ist dabei egal und der „Pfeil“ an der einen Seite der entstehenden Verbindungslinie hat im ER-Modell keine Bedeutung. Abbildung 3.4 zeigt eine Beziehung als Beispiel. Durch einen Rechtsklick auf den Strich einer Beziehung öffnet sich ein Menü

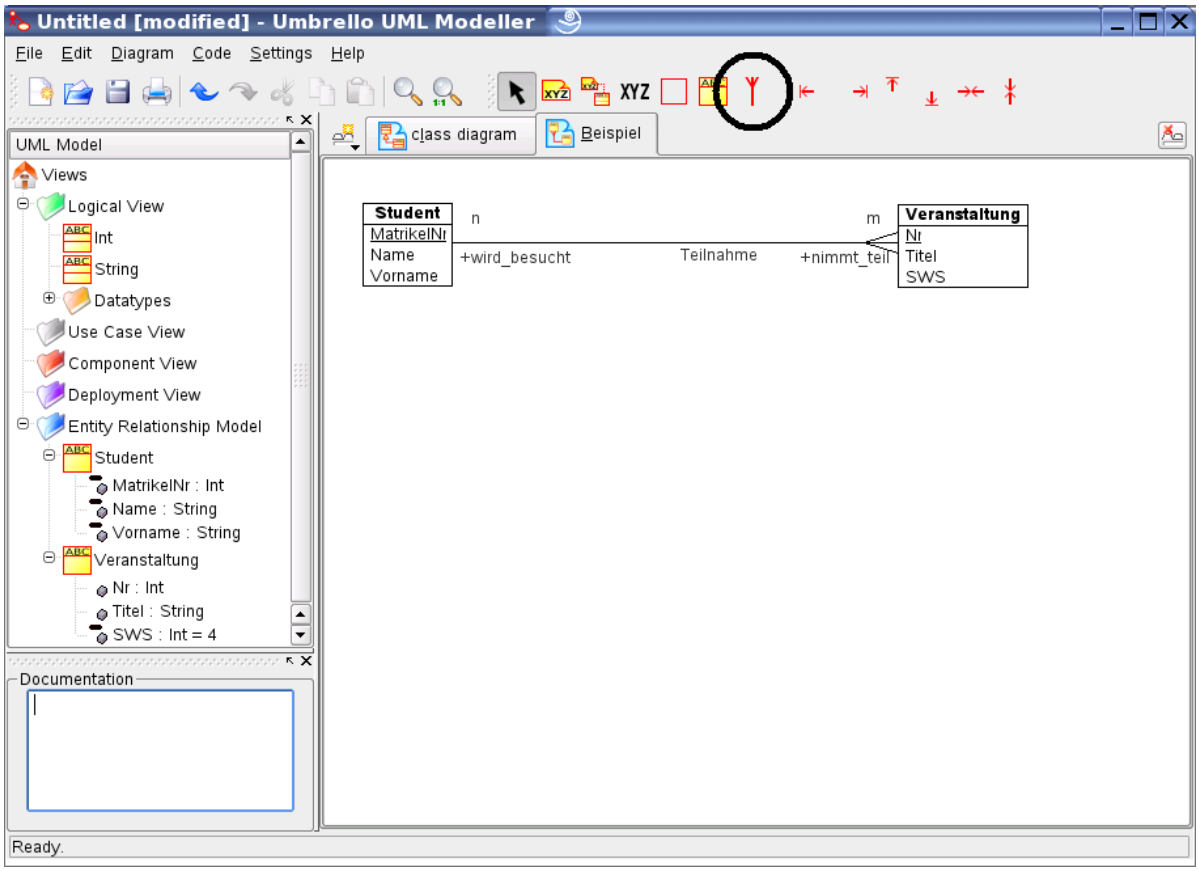

Abbildung 3.4.: Beziehung

für die Angabe eines Namens der Beziehung, Rollenbezeichnern und Kardinalitäten. Beziehungen werden standardmäßig als durchgezogene Linie zwischen den zu verbindenden Entitäten im Diagramm gezeichnet. Man kann durch Doppelklick auf die Beziehung an einer beliebigen Stelle einen sogenannten Ankerpunkt einfügen, der durch einen blauen

Punkt dargestellt wird, wenn die Beziehung ausgewählt ist. Der Ankerpunkt ist verschiebbar, um der Beziehung die gewünschte Form zu geben. Ein Doppelklick auf einen Ankerpunkt entfernt diesen wieder.

Gespeichert wird ein Diagramm im XMI-Format über das Menü („File → Save as“). Außerdem kann man es über den Menüpunkt „Diagram → Export as Picture“ als Grafik speichern.

3.2.1. Anforderungen an ein ER-Diagramm für die Codegenerierung

Zusätzlich zu den durch Umbrello gesicherten Regeln bei der Erstellung von ER-Diagrammen sind für die Generierung eines verwendbaren Codes einige Vorgaben zu beachten. Umbrello stellt sicher, dass ein ER-Diagramm und jede Entität einen Namen hat und eine Beziehung immer zwischen genau zwei Entitäten besteht. Ein Attribut hat immer einen Namen und einen Typ, der sich auf die Curry-Typen abbilden lässt (siehe Abschnitt 4.1), d.h. entweder aus der Menge {`Int`, `Float`, `Char`, `String`, `Bool`, `Date`} ist oder in einem existierenden Modul definiert ist. Weitere Anforderungen an ein ER-Diagramm sind:

- Die Bezeichner für Entitäten, Beziehungen und Rollen müssen für das gesamte Diagramm eindeutig sein.
- Attributnamen müssen eindeutig für eine Entität sein.
- Jede Beziehung hat einen Namen und zwei Rollennamen.
- Für jede Beziehung sind Kardinalitäten aus der folgenden Menge anzugeben:
  $\mathbb{N} \cup \{\mathtt{n}, \mathtt{m}\} \cup \{(min, max), min \in \mathbb{N}, max \in \mathbb{N} \cup \{\mathtt{n}, \mathtt{m}\}, min \leq max\}$
  Dabei stehen `n` und `m` für beliebig viele.
- Einschränkung der Kardinalitäten: Minimum höchstens auf einer Seite zulässig (siehe 4.2.5)
- Rollennamen sollten klein, Entitäts-, Beziehungs- und Attributsnamen groß geschrieben werden.
- Der Attributname „Key“ ist nicht erlaubt.
- Ein Initialwert für ein Datum muss wie folgt angegeben werden:
  dd.mm.yyyy_hh:mm:ss
- Benutzerdefinierte Datentypen dürfen nicht polymorph sein und müssen zusammen mit ihrem Modul angegeben werden: *Modulname.Typ*.

3.3. Umbrello XMI-Formatbeschreibung

Das XMI-Format (XML Metadata Interchange) ist eine Standard-UML-Repräsentation in XML, das entwickelt wurde, um den Austausch von UML-Modellen zwischen verschiedenen Werkzeugen zu ermöglichen.

Der Aufbau des von Umbrello zur Speicherung von ER-Diagrammen verwendeten XML-Formats wird im Folgenden an einem Beispiel erläutert, das mit Umbrello 1.5.52 erstellt wurde. Auf oberster Ebene enthält das XML-Dokument drei Elemente. In dem Element `XMI.header` steht das verwendete Werkzeug mit Versionsnummer. Die hier benötigten Informationen über das ER-Diagramm findet man in dem Element `XMI.content`. Das Element `XMI.extensions` enthält interne Angaben des Werkzeuges und hat hier keine Bedeutung. Die Elemente `UML:Model` und `UML:Namespace.ownedElement` klammern die einzelnen Teile wie die logische Sicht (`Logical View`) und das ER-Diagramm (`Entity Relationship Model`). Genauer eingehen möchte ich nur auf den Teil, in dem das reine ER-Diagramm beschrieben wird. Tabelle 3.1 zeigt den Inhalt dieses Teiles verkürzt auf die in der Implementierung verwendeten Elemente und Attribute.

XML-Element	Attribute
`UML:DataType`	`xmi.id`, `name`
`UML:Class`	`xmi.id`, `name`
`UML:Entity`	`xmi.id`, `name`
`UML:EntityAttribute`	`dbindex_type`, `allow_null`, `initialValue`, `type` (`xmi.id` des `DataType`s bzw. der `Class`), `name`
`UML:Association`	`name`
`UML:AssociationEnd`	`type` (`xmi.id` der `Entity`), `name`, `multiplicity`

Tabelle 3.1.: Verwendete XML-Elemente und Attribute

Die Elemente `DataType` und `Class` definieren beide Datentypen und unterscheiden sich nur dadurch, dass mit `DataType` häufig verwendete Standardtypen von Umbrello vordefiniert werden, auch wenn sie nicht im Diagramm auftauchen, und alle benutzerdefinierten Typen mit dem Element `Class` erst nach Bedarf definiert werden.
Beziehungen innerhalb der im Metamodell dargestellten Daten (Assoziationen im Metamodell) werden durch den XML-Mechanismus der idref ausgedrückt. Zunächst wird jedem Modellelement eine eindeutige Identifikation (`xmi.id`) zugeordnet, die im gesamten Dokument als Referenz-Ankerpunkt zur Verfügung steht.
In Tabelle 3.2 werden mögliche Werte für die einzelnen Attribute dargestellt.

Das in Abbildung 3.4 vorgestellte Beispiel als XML (ebenfalls verkürzt):

```
<XMI.content>
 <UML:Model>
  <UML:Namespace.ownedElement>
   <UML:Model name="Logical View">
```

Attribut	Wert	Bedeutung
`xmi.id`	Zahl	eindeutige Identifikation von zugehörigen Elementen
`name`	String	Bezeichner
`dbindex_type`	1100 1101 1103	keine Angabe Primärschlüssel eindeutiges Attribut (unique) jeder andere Wert hat keine Bedeutung
`allow_null`	0 1	Nullwerte werden nicht erlaubt Nullwerte werden erlaubt
`initialValue`	String	Initialwert eines Attributes
`type`	Zahl	Verbindung zu anderen Elementen über deren `xmi.id`
`multiplicity`	Zahl n oder m (min, max)	Kardinalität (i, i), exakte Zahl Kardinalität (0, n) bzw. (0, m), beliebig viele Kardinalität mit Minimum und Maximum, siehe Abschnitt 3.2.1

Tabelle 3.2.: Mögliche Werte der Attribute

```
 <UML:Namespace.ownedElement>
  <UML:Package name="Datatypes">
   <UML:Namespace.ownedElement>
    <UML:DataType xmi.id="3" name="varchar" />
    <UML:DataType xmi.id="9" name="float" />
    <UML:DataType xmi.id="10" name="double" />
   </UML:Namespace.ownedElement>
  </UML:Package>
  <UML:Class xmi.id="45" name="String" />
  <UML:Class xmi.id="49" name="Int" />
  <UML:Class xmi.id="59" name="MyModule.Email" />
 </UML:Namespace.ownedElement>
</UML:Model>
<UML:Model name="Entity Relationship Model">
 <UML:Namespace.ownedElement>
  <UML:Entity xmi.id="29" name="Student">
   <UML:EntityAttribute dbindex_type="1101" allow_null="0"
                        initialValue="" type="49" name="MatrikelNr" />
   <UML:EntityAttribute dbindex_type="1100" allow_null="0"
                        initialValue="" type="45" name="Name" />
   <UML:EntityAttribute dbindex_type="1100" allow_null="0"
                        initialValue="" type="45" name="Vorname" />
   <UML:EntityAttribute dbindex_type="1100" allow_null="1"
                        initialValue="" type="59" name="Email" />
  </UML:Entity>
```

```
<UML:Entity xmi.id="30" name="Veranstaltung">
 <UML:EntityAttribute dbindex_type="1101" allow_null="0"
                      initialValue="" type="49" name="Nr" />
 <UML:EntityAttribute dbindex_type="1103" allow_null="0"
                      initialValue="" type="45" name="Titel" />
 <UML:EntityAttribute dbindex_type="1100" allow_null="0"
                      initialValue="4" type="49" name="SWS" />
</UML:Entity>
<UML:Entity xmi.id="31" name="Dozent">
 <UML:EntityAttribute dbindex_type="1101" allow_null="0"
                      initialValue="" type="49" name="Nr" />
 <UML:EntityAttribute dbindex_type="1100" allow_null="0"
                      initialValue="" type="45" name="Name" />
 <UML:EntityAttribute dbindex_type="1100" allow_null="0"
                      initialValue="" type="45" name="Vorname" />
</UML:Entity>
<UML:Entity xmi.id="32" name="Gruppe">
 <UML:EntityAttribute dbindex_type="1100" allow_null="0"
                      initialValue="" type="45" name="Termin" />
</UML:Entity>
<UML:Association name="Veranstalten">
 <UML:Association.connection>
  <UML:AssociationEnd type="31" name="wird_gehalten"
                      multiplicity="(1,1)" />
  <UML:AssociationEnd type="30" name="haelt"
                      multiplicity="(0,n)" />
 </UML:Association.connection>
</UML:Association>
<UML:Association name="Teilnahme">
 <UML:Association.connection>
  <UML:AssociationEnd type="29" name="wird_besucht"
                      multiplicity="(0,n)" />
  <UML:AssociationEnd type="30" name="nimmt_teil"
                      multiplicity="(0,m)" />
 </UML:Association.connection>
</UML:Association>
<UML:Association name="Zugehoerigkeit">
 <UML:Association.connection>
  <UML:AssociationEnd type="29" name="besteht_aus"
                      multiplicity="(3,3)" />
  <UML:AssociationEnd type="32" name="ist_in"
                      multiplicity="(0,n)" />
 </UML:Association.connection>
</UML:Association>
```

```
      </UML:Namespace.ownedElement>
    </UML:Model>
   </UML:Namespace.ownedElement>
  </UML:Model>
 </XMI.content>
```

In der logischen Sicht sind einige Standardtypen zu sehen (`DataType`), die aber nicht verwendet werden. Die hier verwendeten Wertebereiche `String`, `Int` und `MyModule.Email` werden in `Class`-Elementen definiert. Die Entitäten `Student`, `Veranstaltung`, `Dozent` und `Gruppe` mit ihren Attributen und die Beziehungen `Veranstalten`, `Teilnahme` und `Zugehoerigkeit` lassen sich leicht erkennen.

4. Implementierung

Die Implementierung lässt sich in drei Aufgaben unterteilen. Eine mit einem Werkzeug erstellte XML-Datei ist in die in Abschnitt 2.3 vorgestellte Datenstruktur umzuwandeln, die eine vom verwendeten Tool unabhängige Repräsentation eines ER-Diagramms darstellt. Ab hier soll die weitere Implementierung unabhängig vom Werkzeug bleiben, sodass nur der erste Teil angepasst werden muss, falls man ein anderes Werkzeug verwenden möchte. Des Weiteren sollen die Daten aus dem ER-Diagramm in eine vorbereitende Darstellung für eine Art relationales Datenbankschema transformiert werden, um schließlich daraus die Datenbank-Funktionen zu generieren.

4.1. Konvertierung von XML in eine Curry-Datenstruktur

Eine Aufgabe ist, die mit Umbrello wie in Abschnitt 3.2 beschrieben erstellte XML-Darstellung eines ER-Diagramms in die Curry-Datenstruktur `ERD` zu konvertieren. Dadurch wird eine gewisse Unabhängigkeit von dem verwendeten Werkzeug erreicht, da bei einem Wechsel des Werkzeuges nur dieser Teil verändert werden muss und die weitere Transformation dann unabhängig vom Speicherformat des Werkzeuges ist. Hier wird überprüft, ob alle Vorgaben aus Abschnitt 3.2.1 eingehalten wurden. Noch mehr Unabhängigkeit würde eine Trennung der Konvertierung von der Überprüfung bringen. Das könnte man als zukünftige Aufgabe ändern.
Durch Umbrello ist sichergestellt, dass es keine Beziehung ohne zugehörige Entitäten gibt, Entitäten einen Namen und Attribute einen Namen und einen Wertebereich haben.

Im Modul `XML2ERD` wird die Konvertierung mittels der Curry-Bibliothek `XML`, die eine Datenstruktur für XML-Dokumente und einige Funktionen dazu bereitstellt, implementiert.

```
data XmlExp = XText String
            | XElem String [(String, String)] [XmlExp]
```

Ein XML-Ausdruck enthält einfachen Text oder ein XML-Element mit einem Tag-Namen, einer Liste von Attribut-Wert-Paaren und einer Liste von enthaltenen XML-Ausdrücken.
Die Funktion

```
readXmlFile :: String -> IO XmlExp
```

liest ein XML-Dokument aus einer Datei und gibt die zugehörige `XmlExp` zurück.

Die Konvertierung wird gestartet mit der Funktion

```
convert :: XmlExp -> ERD
```

die einen XML-Ausdruck bekommt und daraus zuerst die benötigten Teile herausfiltert und in drei Listen aufteilt. Erstens die Elemente `UML:Entity` für die Entitäten, zweitens die Elemente `UML:Association` für die Beziehungen und drittens einen Liste von ID-Name-Paaren, um einer ID Entitäten oder Datentypen zuordnen zu können. Eine Verbesserung wäre, diese Zuordnung statt in einer Liste von Paaren in einem Suchbaum zu speichern, um die Paare schneller suchen zu können. Außerdem wird noch der Name des Diagramms gesucht. Die Bezeichner für Entitäten, Beziehungen und Rollen werden auf Eindeutigkeit geprüft.

In `convert` werden die Funktionen

```
convertE :: [(String, String)] -> XmlExp -> Entity
```

```
convertR :: [(String, String)] -> XmlExp -> Relationship
```

aufgerufen, um mit Hilfe der Liste von ID-Name-Paaren Entitäten bzw. Beziehungen in der XML-Darstellung in die Datenstruktur `Entity` bzw. `Relationship` zu konvertieren. In `convertE` wird der Name der Entität aus der Liste der Attribut-Wert-Paare geholt und geprüft, ob es Attribute gibt. Die Funktion

```
convertAttr :: [(String, String)] -> XmlExp -> Attribute
```

wird verwendet, um aus den Attributen der Entität in der XML-Darstellung eine Liste des Typs `Attribute` zu erhalten. Dazu werden der Name des Attributs (`name`), der Typ als ID (`type`), der Standardwert (`initialValue`), der Index-Typ (`dbindex_type`) und die Zulässigkeit von Nullwerten (`allow_null`) ausgewertet. Der Name des Typs ist in der Liste der ID-Name-Paaren enthalten und wird zusammen mit einem eventuell vorhandenen Standardwert durch die Funktion

```
convertDomain :: Maybe String -> Maybe String -> Domain
```

in den Typ `Domain` für den Wertebereich konvertiert. Der Index-Typ ist wie in Tabelle 3.2 durch eine Zahl kodiert: 1101 steht für Primärschlüssel (`PKey`) und 1103 für Eindeutigkeit (`Unique`). Alles Andere wird nicht weiter unterschieden (`NoKey`). Jede Liste von Attributen wird mit der Funktion `checkAttrs` kontrolliert, ob die Vorgaben eingehalten werden, d.h. ob der Name des Attributs nicht „Key“ ist, und ob bei angegebenem

Standardwert nicht Nullwerte zugelassen werden oder die Eindeutigkeit des Attributs gefordert wird.

In `convertR` wird ebenfalls der Name der Beziehung aus der Liste der Attribut-Wert-Paare geholt und die Verbindungen zu den beiden beteiligten Entitäten jeweils mit der Funktion

```
convertREnd :: [(String, String)] -> XmlExp -> REnd
```

in die Datenstruktur `REnd` konvertiert. Es wird über die im Attribut `type` gespeicherte ID die verbundene Entität aus der Liste der ID-Name-Paare herausgesucht, der Rollenbezeichner aus dem Attribut `name` und die Kardinalität aus dem Attribut `multiplicity` geholt. Zur Umwandlung der Kardinalität wird die Funktion

```
convertCard :: Maybe String -> Cardinality
```

verwendet, die den String, der nach den Vorgaben aus Abschnitt 3.2.1 aufgebaut sein muss, in den Datentyp `Cardinality` umwandelt. Überprüft wird für eine Beziehung, ob ein Name und zwei Rollenbezeichner angegeben sind und es keine zwei Minima gibt.

Ruft man nun die Funktion `convert` mit dem Beispiel auf und lässt sich das Ergebnis ausgeben,

```
test = do
  xml <- readXmlFile "./Uni.xmi"
  print (convert xml)
```

so erhält man die Daten des ER-Diagramms aus Abbildung 2.1 in der Datenstruktur `ERD`.

```
(ERD "Uni"
  [(Entity "Student"
     [(Attribute "MatrikelNr" (IntDom Nothing) PKey False),
      (Attribute "Name" (StringDom Nothing) NoKey False),
      (Attribute "Vorname" (StringDom Nothing) NoKey False),
      (Attribute "Email" (UserDefined "MyModule.Email" Nothing) NoKey True)]),
   (Entity "Veranstaltung"
     [(Attribute "Nr" (IntDom Nothing) PKey False),
      (Attribute "Titel" (StringDom Nothing) Unique False),
      (Attribute "SWS" (IntDom (Just 4)) NoKey False)]),
   (Entity "Dozent"
     [(Attribute "Nr" (IntDom Nothing) PKey False),
      (Attribute "Name" (StringDom Nothing) NoKey False),
      (Attribute "Vorname" (StringDom Nothing) NoKey False)]),
   (Entity "Gruppe"
```

```
    [(Attribute "Termin" (StringDom Nothing) NoKey False)])]
 [(Relationship "Veranstalten"
    [(REnd "Dozent" "wird_gehalten" (Exactly 1)),
     (REnd "Veranstaltung" "haelt" (Range 0 Nothing))]),
  (Relationship "Teilnahme"
    [(REnd "Student" "wird_besucht" (Range 0 Nothing)),
     (REnd "Veranstaltung" "nimmt_teil" (Range 0 Nothing))]),
  (Relationship "Zugehoerigkeit"
    [(REnd "Student" "besteht_aus" (Exactly 3)),
     (REnd "Gruppe" "ist_in" (Range 0 Nothing))])])
```

4.2. Umsetzung von Entitäten und Beziehungen

Im ER-Modell gibt es zwei Strukturierungskonzepte: Entitätstypen und Beziehungstypen. Im relationalen Modell werden beide auf Relationen, d.h. Tabellen, abgebildet.
Aus einem Entitätstyp wird leicht eine Relation, indem der Name als Name der Relation und die Attribute mit ihren Eigenschaften als Spalten übernommen werden.
Grundsätzlich gibt es zwei Möglichkeiten, Beziehungstypen in Relationen nachzubilden [8]. Durch Verwendung einer neuen Relation, die die beteiligten Entitäten verbindet, lässt sich jede Relation umsetzen. Um möglichst wenige Relationen zu erhalten, kann man einige Beziehungstypen auch durch Fremdschlüssel nachbilden. Eine Relation kann mehrere Fremdschlüssel besitzen, die den gleichen oder verschiedene Beziehungstypen realisieren.
In Tabelle 4.1 werden alle möglichen Kardinalitäten von Beziehungen aufgelistet. In den folgenden Abschnitten werden die einzelnen Fälle und ihre Umsetzungsmöglichkeiten näher erläutert.

Typ	Spezialfall
1:1	(0,1):(1,1) (0,1):(0,1)
1:n	(0,1):(0,max_2), $max_2 \in \mathbb{N}_{>1} \cup \{\mathtt{n}\}$ (0,1):(i,max_2), $i \in \mathbb{N}, max_2 \in \mathbb{N}_{>1} \cup \{\mathtt{n}\}, i \leq max_2$ (1,1):(0,max_2), $max_2 \in \mathbb{N}_{>1} \cup \{\mathtt{n}\}$
n:m	(0,max_1):(0,max_2), $max_1, max_2 \in \mathbb{N}_{>1} \cup \{\mathtt{n}, \mathtt{m}\}$ (0,max_1):(i,max_2), $i \in \mathbb{N}_{>1}$,max$_1$, $max_2 \in \mathbb{N}_{>1} \cup \{\mathtt{n}, \mathtt{m}\}, i \leq max_2$ (0,max_1):(i,i), $max_1 \in \mathbb{N}_{>1} \cup \{\mathtt{n}\}, i \in \mathbb{N}_{>1}$

Tabelle 4.1.: Mögliche Kardinalitäten von Beziehungen

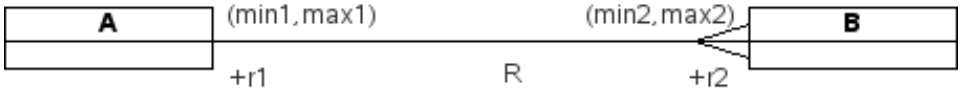

Abbildung 4.1.: Entitäten A und B verbunden durch Beziehung R mit den Kardinalitäten (min_1, max_1) und (min_2, max_2), wobei $0 \leq min_i \leq max_i$, $min_i \in \mathbb{N}$, $max_i \in \mathbb{N} \cup \{\mathtt{n}, \mathtt{m}\}$

4.2.1. Integritätsbedingungen

Integritätsbedingungen sind Anforderungen, die der modellierte Datenbestand während seiner gesamten Lebensdauer zu erfüllen hat, d.h. die die Integrität der Daten gewährleisten.

Im ER-Modell gibt es zwei verschiedene Arten von Integritätsbedingungen [9]:

- Entitäts-Integrität (Entity Integrity)
- Referenz-Integrität (Referential Integrity)

Die Entitäts-Integrität bedeutet, dass es Schlüssel zur eindeutigen Identifizierung von Entitäten eines Typs im modellierten Datenbestand geben muss. Ein Schlüssel besteht aus einem oder mehreren Attributen, den Schlüsselattributen, die keine Nullwerte enthalten dürfen. Also kann eine mit dem ER-Modell entworfene Datenbank keine verschiedenen Entitäten mit identischen Schlüsselattributen enthalten.
Die Referenz-Integrität ist die Integrität auf Beziehungsebene, d.h. wenn eine Relation einen Fremdschlüssel enthält, dann muss jeder der Werte der Fremdschlüssel entweder Null sein oder als Wert eines Primärschlüssels in der zugehörigen Relation existieren. Ausserdem gehört der Grad von Beziehungen dazu, der festlegt, wie oft eine Entität an einer Beziehung eines gegebenen Typs teilnehmen darf bzw. muss.
Auch die Wertebereiche der Attribute kann man als Integritätsbedingungen auffassen. Da hierfür Curry-Typen verwendet werden, wird die Einhaltung dieser Integritätsbedingung schon von der verwendeten Programmiersprache garantiert.

4.2.2. Einfach-einfache Beziehungen (1:1)

(0,1):(1,1)

Fügt man den Schlüssel der B-Seite als Fremdschlüssel auf der A-Seite hinzu, so ist sichergestellt, dass jedem Eintrag der A-Seite genau ein existierender Eintrag der B-Seite zugeordnet wird. Alternativ könnte man auch den Schlüssel der A-Seite als Fremdschlüssel auf der B-Seite hinzufügen, aber dann müssten Nullwerte für den Fremdschlüssel zugelassen werden, was man hier vermeiden kann.

Der Fremdschlüssel muss eindeutig (`Unique`) sein, damit auch die andere Richtung garantiert wird.
Es wäre auch möglich, beide Seiten um Fremdschlüssel zu erweitern. Das führt allerdings zu Redundanz, die möglichst vermieden werden soll.

(0,1):(0,1)

Dieser Fall wird umgesetzt wie (0,1):(1,1), bis auf dass hier jedem Eintrag der A-Seite ein oder kein Eintrag der B-Seite zugeordnet wird. Deswegen ist der eingefügte Fremdschlüssel vom Typ `Maybe`, d.h. Nullwerte werden für den Fremdschlüssel zugelassen. Zur Vereinheitlichung wird der Fremdschlüssel auf der Seite eingefügt, auf die der Pfeil zeigt (hier B), aber beide Seiten wären zulässig.

4.2.3. Einfach-komplexe Beziehungen (1:n)

(0,1):(0,max$_2$)

Auf der B-Seite wird der Schlüssel von A als Fremdschlüssel vom Typ `Maybe` hinzugefügt, d.h. Nullwerte werden zugelassen. Ist max_2 eine Zahl, so muss bei Operationen, die die Datenbank ändern, überprüft werden, ob es den Fremdschlüssel schon max_2-mal gibt, um die Beschränkung einzuhalten.

(0,1):(i,max$_2$)

Eine Möglichkeit ist die Umsetzung wie im Fall (0,max_1):(i,max_2) mit $max_1 = 1$.
Hier wird diese Beziehung umgesetzt durch Einfügen des Schlüssels der A-Seite als Fremdschlüssel vom Typ `Maybe` auf der B-Seite. Die Einhaltung von Minimum und Maximum muss bei Operationen, die die Datenbank ändern, überprüft werden.

(1,1):(0,max$_2$)

Die Umsetzung ist genau wie im Fall (0,1):(0,max$_2$), nur dass der eingefügte Fremdschlüssel nicht Null sein darf und deswegen nicht vom Typ `Maybe` ist.

4.2.4. Komplex-komplexe Beziehungen (n:m)

Zur Veranschaulichung sind hier grafische Beispiele auf Ebene der ER-Diagramme (nicht auf Ebene des Werkzeuges) aufgeführt.

(0,max$_1$):(0,max$_2$)

Für die Umsetzung dieser Art von Beziehung benötigt man eine zusätzliche Relation, die mit den beiden Entitäten jeweils durch eine 1:n-Beziehung verbunden wird. So erhält man statt einer n:m-Beziehung zwei 1:n-Beziehungen, die sich nun wie oben beschrieben durch die Schlüssel der beiden ursprünglichen Relationen als Fremdschlüssel in der neuen Relation umsetzen lassen. Diese beiden Fremdschlüssel bilden zusammen den Schlüssel der neuen Relation, d.h. sie müssen zusammen eindeutig sein.

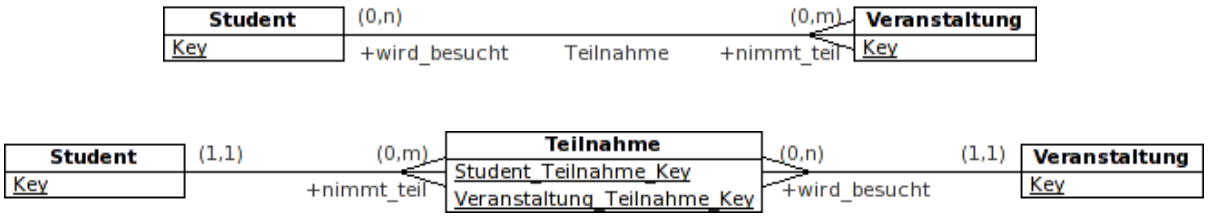

Abbildung 4.2.: Beispiel für (0,n):(0,m)-Beziehung

Abbildung 4.2 zeigt die Umsetzung am Beispiel. Ein Student kann an beliebig vielen Veranstaltungen teilnehmen und an einer Veranstaltung können ebenfalls beliebig viele Studenten teilnehmen. Als Name der Relation zur Umsetzung der Beziehung wird der Name der ursprünglichen Beziehung gewählt und die Rollenbezeichner an der entsprechenden Stelle in den 1:n-Beziehungen weiterverwendet. Zusätzliche Rollenbezeichner werden nicht benötigt, denn der Benutzer soll möglichst nur das zu sehen bekommen, was er selbst angegeben hat.
Wenn Maxima angegeben werden, die Beziehung also beschränkt ist, muss die Einhaltung der Maxima durch Operationen garantiert werden.

(0,max$_1$):(i,max$_2$)

Abbildung 4.3.: Beispiel für (0,10):(3,4)-Beziehung

Bis auf das Minimum i entspricht dieser Fall dem vorigen. Einträge in A dürfen nur zusammen mit i Einträgen in R existieren, was bei der Codegenerierung berücksichtigt werden muss.

Das Beispiel in Abbildung 4.3 hat nicht nur ein Minimum sondern auch Maxima in den Kardinalitäten der Beziehung. Eine Gruppe kann aus 3 bis 4 Studenten bestehen und ein Student kann in 0 bis 10 Gruppen sein. Die Einhaltung dieser Vorgaben kann nur

durch entsprechenden Funktionen garantiert werden, die eine Beziehung über mehrere Einträge in die neue Relation *Zugehoerigkeit* umsetzen und auch die Einhaltung der Maxima garantieren.

(0,max_1):(i,i)

Diesen Fall könnte man durch i-faches Hinzufügen des Schlüssels von B zur A-Seite als i Fremdschlüssel des gleichen Typs, die alle nicht Null werden dürfen, abbilden. Ihre Werte müssen allerdings unterschiedlich sein. Falls max_1 eine Zahl ist, muss bei Änderungen für jeden dieser Fremdschlüssel einzeln geprüft werden, ob es ihn schon i-mal gibt.
Die Kardinalität i kann sehr groß werden und der Umgang mit mehreren Fremdschlüsseln, die sich nicht unterscheiden lassen und keine Reihenfolge haben, ist schwierig. Deswegen ist es sinnvoller, wie im vorigen Fall doch eine zusätzliche Relation zu verwenden.

4.2.5. Zusammenfassung der betrachteten Fälle

Alle in Tabelle 4.1 aufgelisteten Fälle sind Spezialfälle des allgemeinsten hier betrachteten Falles (0,max_1):(i,max_2) mit $i \in \mathbb{N}$, $max_1, max_2 \in \mathbb{N} \cup \{n, m\}$ und $i \leq max_2$.

Der einzige Fall, der bewusst ausgeschlossen wird, ist also eine Beziehung, die in beide Richtungen ein Minimum hat. Wenn es solch eine Beziehung mit beidseitigen Abhängigkeiten geben würde, ist nicht klar, wie das Eintragen in die Datenbank funktioniert. Welche Einträge müssen zuerst existieren, um davon abhängige Einträge machen zu können? Sind die beiden Minima gleich, könnte man die erforderliche Anzahl Einträge in beide Relationen in einer Transaktion unterbringen. Wenn dies beabsichtigt ist, kann man das zweite Minimum, ähnlich wie in Abschnitt 4.5.8 für Zyklen beschrieben, auch selbst durch eine Funktion ergänzen. Hier wird so eine Funktion jedoch nicht implementiert.

Bei allem Umsetzungen von n:m-Beziehungen wird als Kardinalität der neuen Beziehungen (1,1) statt (0,1) auf der Seite der ursprünglichen Relationen verwendet. So spart man sich einerseits das `Maybe` in den Fremdschlüsseln und stellt andererseits automatisch sicher, dass mit jedem Eintrag in eine zusätzliche Relation auch wirklich zwei Einträge miteinander verbunden werden.

4.3. Transformation

Im ER-Modell gibt es zwei Strukturierungskonzepte: Entitäts- und Beziehungstypen. Im Relationalen Modell werden beide auf Relationen, d.h. Tabellen, abgebildet. Die Entitätstypen entsprechen Relationen und die Beziehungstypen werden durch Fremdschlüssel und zugehörige Primärschlüssel nachgebildet. Dadurch ist die Transformation vom ER-Modell ins Relationale Modell direkt auf der in Abschnitt 2.3 vorgestellten Da-

tenstruktur `ERD` möglich. Die transformierten Entitäten entsprechen dann den Tabellen und die Beziehungen werden nur noch zur Generierung von Funktionen aufgehoben. Das Modul `Transformation` implementiert die Transformation. Mit der Funktion

```
transform :: ERD -> ERD
```

lässt sich die Transformation starten.

4.3.1. Primärschlüssel

Zur Vereinfachung wird in dieser ersten Version der Transformation und Codegenerierung erstmal für jeden benutzerdefinierten Entitätstypen ein künstlicher Primärschlüssel vom Typ `Int` eingefügt (Funktion `addKey`), der den Namen „Key“ erhält. Der vom Benutzer gekennzeichnete Primärschlüssel wird (falls vorhanden) auf `Unique` gesetzt (Funktion `deleteKey`), um den Zugriff auf die Einträge durch dieses Attribut weiterhin zu ermöglichen. Ein Beispiel für diesen Transformationsschritt ist

```
Entity "Veranstaltung"
       [Attribute "Nr" (IntDom Nothing) PKey False, ...]
⇝
Entity "Veranstaltung"
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "Nr" (IntDom Nothing) Unique False, ...]
```

4.3.2. Beziehungen

Grundsätzlich lässt sich eine Beziehung immer durch eine Tabelle darstellen, die die beteiligten Entitäten über Fremdschlüssel verbindet. Wie in Abschnitt 4.2 beschrieben, kann man in bestimmten Fällen auch Fremdschlüssel direkt in eine der beteiligten Entitäten einfügen und so die Tabellen für diese Beziehungen sparen. Die Funktion

```
transformRel :: ([Entity], [Relationship]) -> [Relationship]
             -> ([Entity], [Relationship])
```

realisiert die Transformation der Beziehungen und liefert ein Paar aus transformierten Entitäten und Beziehungen. Für jede vom Benutzer angelegte Beziehung wird die passende Funktion zur Transformation aufgerufen. Hierbei werden alle in Abschnitt 4.2 beschriebenen Fälle berücksichtigt. Fremdschlüsseln wird ein intern verwendeter Name gegeben, der sich aus der Herkunft, dem Beziehungsnamen und dem Schlüsselnamen zusammensetzt, z.B. „Dozent_Veranstalten_Key“. Die Herkunft wird verwendet, da Attributnamen nur für eine Entität eindeutig sind, und der Beziehungsname sichert die Eindeutigkeit des neuen Namens bei mehreren Beziehungen zwischen zwei Entitäten.

4.3.3. Beispiele

(1,1):(0,1)-Beziehung: Ein Student hat kein oder ein Vordiplom, ein Vordiplom ist genau einem Studenten zugeordnet.

```
Entity "Vordiplom"
       [Attribute "Key" (IntDom Nothing) PKey False]
⇝
Entity "Vordiplom"
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "Student_R_Key" (KeyDom "Student") Unique True]
```

(1,1):(0,n)-Beziehung: Ein Dozent hält beliebig viele Veranstaltungen, einer Veranstaltung ist genau ein Dozent zugeordnet.

```
Entity "Veranstaltung"
       [Attribute "Key" (IntDom Nothing) PKey False]
⇝
Entity "Veranstaltung"
       [Attribute "Key" (IntDom Nothing) PKey False,
        Attribute "Dozent_Veranstalten_Key" (KeyDom "Dozent") NoKey False]
```

(0,n):(0,m)-Beziehung: Ein Student nimmt an beliebig vielen Veranstaltungen teil, eine Veranstaltung wird von beliebig vielen Studenten besucht.

```
[Entity "Student"
        [Attribute "Key" (IntDom Nothing) PKey False],
 Entity "Veranstaltung"
        [Attribute "Key" (IntDom Nothing) PKey False]]
[Relationship "Teilnahme"
              [REnd "Student" "wird_besucht" (Range 0 Nothing),
               REnd "Veranstaltung" "nimmt_teil" (Range 0 Nothing)]]
⇝
[Entity "Student" [...],
 Entity "Veranstaltung" [...],
 Entity "Teilnahme"
        [Attribute "Student_Teilnahme_Key" (KeyDom "Student") NoKey False,
         Attribute "Veranstaltung_Teilnahme_Key" (KeyDom "Veranstaltung")
                   NoKey False]]
[Relationship []
              [REnd "Student" [] (Exactly 1),
```

```
               REnd "Teilnahme" "nimmt_teil" (Range 0 Nothing)],
 Relationship []
              [REnd "Veranstaltung" [] (Exactly 1),
               REnd "Teilnahme" "wird_besucht" (Range 0 Nothing)]]
```

4.4. Transaktionen und das Modul `Trans`

Zur Vermeidung von Inkonsistenzen muss das Erstellen oder Ändern (auch das Löschen) eines Eintrags eine atomare Operation sein. Da aber jeweils mehrere Zugriffe auf die Datenbank erforderlich sind, werden diese durch eine Transaktion (siehe auch Abschnitt 2.2.1) gekapselt. Die von der Bibliothek `Dynamic` bereitgestellte Funktion `transaction` für Transaktionen kann hier jedoch nicht verwendet werden, da Transaktionen nicht geschachtelt werden können und deswegen der Benutzer diese Funktionen nicht mehr in eigenen Transaktionen verwenden kann. Eine Lösungsmöglichkeit ist, einen Typ

```
type Trans a = IO (TransResult a)

data TransResult a = OK a
                   | TransError Error String

data Error = KeyNotExistsError
           | DuplicateKeyError
           | UniqueError
           | MinError
           | MaxError
           | UserDefinedError
```

zu definieren, der dem Benutzer sagt, dass um Funktionen dieses Typs eine Transaktion gehört. `Trans` ist eine Monade, also können Funktionen dieses Typs sequentialisiert werden. Eine Transaktion liefert entweder ein Ergebnis (`OK`) oder einen Fehler (`TransError`) mit einem bestimmten Fehlertyp und einem `String` zur genauen Beschreibung des Fehlers. Die Fehler sind bei Bedarf erweiterbar. Tritt ein Fehler in einer Sequenz von Funktionen auf, so wird dieser in der Monade versteckt und „durchgeschleift“. Dazu werden die folgenden Monaden-Funktionen zur Verfügung gestellt:

```
(>>~) :: Trans a -> (a -> Trans b) -> Trans b
t1 >>~ f = do
  r1 <- t1
  case r1 of TransError e s -> return r1
             OK x           -> f x
```

```
(>>-) :: Trans a -> Trans b -> Trans b
t1 >>- t2 = t1 >>~ \_ -> t2

returnTrans :: a -> Trans a
returnTrans x = return (OK x)

failTrans :: String -> Trans a
failTrans s = return (TransError UserDefinedError s)
```

Mit der Bind-Funktion `(>>~)` können Funktionen vom Ergebnistyp `Trans` mit Weitergabe der Zwischenergebnisse sequentialisiert werden. Zur Abkürzung kann man auch die Funktion `(>>-)` verwenden, wenn man die Zwischenergebnisse nicht benötigt. Ein Ergebnis liefert `returnTrans`, einen Fehler `failTrans`.

Zusätzlich werden einige nützliche Funktionen implementiert. Es ist bei der Benutzung der Datenbank wichtig, an das Ergebnis einer Transaktion heranzukommen. Dafür gibt es die Funktion

```
result :: TransResult a -> a
result (OK x) = x
result (TransError e s) = error (show e ++ " " ++ show s)
```

Um eine Liste von `Trans`-Aktionen zu sequentialisieren und alle Ergebnisse in einer Liste zu sammeln bzw. zu ignorieren, sind die folgenden Funktionen implementiert.

```
sequenceTrans :: [Trans a] -> Trans [a]
sequenceTrans [] = returnTrans []
sequenceTrans (c:cs) =
  c >>~ \x ->
  sequenceTrans cs >>~ \xs ->
  returnTrans (x:xs)

sequenceTrans_ :: [Trans _] -> Trans ()
sequenceTrans_ = foldr (>>-) (returnTrans ())
```

Eine `Trans`-Aktion kann auch auf jedes Element einer Liste angewendet werden.

```
mapTrans :: (a -> Trans b) -> [a] -> Trans [b]
mapTrans f = sequenceTrans . map f

mapTrans_ :: (a -> Trans _) -> [a] -> Trans ()
mapTrans_ f = sequenceTrans_ . map f
```

Damit stehen für den generierten Code und dessen Erweiterungen des Benutzers alle Funktionen zu Verfügung, um mit der `Trans`-Monade umzugehen.

4.5. Vorlagen für die Codegenerierung

Abbildung 4.4.: ER-Diagramm

In diesem Abschnitt werden allgemein gehaltene Vorlagen für die Codegenerierung vorgestellt. Abkürzend für einen Entitätsnamen wird *en* oder *EN* verwendet, um zu unterscheiden, ob der Entitätsname mit einem klein oder groß geschriebenen Buchstaben beginnt. Alle anderen durch Elemente des ER-Diagramms zu ersetzenden Bezeichner sind selbsterklärend. Codeteile in eckigen Klammern werden nur in bestimmten Fällen generiert. Zur Veranschaulichung wird für jede Codevorlage eine Beispielfunktion zu dem ER-Diagramm in Abbildung 4.4 gezeigt. Den vollständigen generierten Code für dieses kleine Beispiel findet man in Anhang B. Die im generierten Code verwendeten allgemeinen Funktionen befinden sich im Modul `Intern`. Durch die Verwendung von allgemeinen Funktionen wird die Codegenerierung stark vereinfacht und die einzelnen generierten Funktionen wesentlich kürzer. Auch sind diese so einfach veränderbar, ohne etwas an der Codegenerierung selber ändern zu müssen.

4.5.1. Importe

Im generierten Code werden einige Module benötigt. Einerseits sind das die verwendeten Bibliotheken `Dynamic` (siehe Abschnitt 2.2), `Trans` (siehe Abschnitt 4.4) und `Intern` (allgemeine Funktionen, die im Folgenden vorgestellt werden) und andererseits die Module, in denen Datentypen definiert sind, die als Wertebereiche für Attribute angegeben sind (im Beispiel `MyModule`).

4.5.2. Datentypen

Für jede vom Benutzer angelegte Entität werden zwei Datentypen generiert, einen für die Entität selber und einen für den Schlüssel der Entität. Zur Vereinfachung werden für jede Entität künstliche Schlüssel vom Typ `Int` verwendet. Das hierfür verwendete Typsynonym

```
type Key = Int
```

ermöglicht die spätere Erweiterung auf beliebige vom Benutzer gewählte Schlüssel. Die Parameter des Konstruktors EN sind der Schlüsseltyp und die Typen der Attribute der Entität ($attrTyp_2$, ..., $attrTyp_n$), die auch vom Typ `Maybe` sein können, falls Nullwerte zugelassen werden, und die auch automatisch generierte Fremdschlüssel vom Typ `Key` sein können. Exportiert wird jeweils nur der Typ ohne Konstruktor um sicherzustellen, dass neue Einträge ausschließlich unterstützt durch Funktionen angelegt werden. Insbesondere können Schlüssel nicht vom Benutzer manipuliert werden, sondern werden automatisch erzeugt.

`data` EN `=` EN `Key` $attrTyp_2$ ... $attrTyp_n$

`data` EN`Key = `EN`Key Key`

Ist die Entität generiert, hat also keinen künstlichen Schlüssel, wird nur der Datentyp für die Entität erstellt.

Für die Entität *Veranstaltung* werden also die beiden Datentypen

```
data Veranstaltung = Veranstaltung Key Key Int String Int

data VeranstaltungKey = VeranstaltungKey Key
```

generiert. Hier sieht man einen Fremdschlüssel, der aus der Beziehung *Veranstalten* stammt.

4.5.3. Getter und Setter

Für jedes Attribut einer Entität gibt es eine Zugriffsfunktion, die dessen Wert liefert, der auch vom Typ `Maybe` sein kann, falls für das Attribut Nullwerte zugelassen sind. Der Wert des Schlüssels wird dabei in der Datenstruktur EN`Key` versteckt.
Veränderbar sind durch eine set-Funktion alle Attribute, die nicht Teil des Schlüssels sind. Schlüssel werden automatisch bei der Erstellung eines neuen Eintrags hinzugefügt und sollen dann nicht veränderbar sein. Dadurch werden Inkonsistenzen vermieden, die durch Verwendung des Schlüssels als Fremdschlüssel entstehen könnten.
Die Parameter `x1`, ..., `x`n stehen für die n Attribute einer Entität und $Attr_i$, $i = 1$, ..., *Anzahl Attribute ohne Schlüsselattribute* ist das i. Attribut vom Typ $attrTyp_i$. Eine Funktion, die dazu dient, den Wert eines Schlüssels aus der Datenstruktur zu holen, wird intern verwendet, soll aber nicht für den Benutzer zur Verfügung stehen und wird nicht exportiert.

$enAttr_i$ `::` EN `->` $attrTyp_i$
$enAttr_i$ `(`EN `_ ... x ... _) = x`

```
set$ENAttr_i$ :: $EN$ -> $attrTyp_i$ -> $EN$
set$ENAttr_i$ ($EN$ x1 ... _ ... x$n$) x = $EN$ x1 ... x ... x$n$

$en$Key :: $EN$ -> $EN$Key
$en$Key ($EN$ k _ ... _) = $EN$Key k

$en$KeyToKey :: $EN$Key -> Key   -- Private
$en$KeyToKey ($EN$Key k) = k
```

Die folgenden sind ein Teil der generierten Funktionen für Veranstaltungen.

```
veranstaltungKey :: Veranstaltung -> VeranstaltungKey
veranstaltungKey (Veranstaltung x _ _ _ _) = VeranstaltungKey x

veranstaltungTitel :: Veranstaltung -> String
veranstaltungTitel (Veranstaltung _ _ _ x _) = x

setVeranstaltungTitel :: Veranstaltung -> String -> Veranstaltung
setVeranstaltungTitel (Veranstaltung x1 x2 x3 _ x5) x =
  Veranstaltung x1 x2 x3 x x5
```

4.5.4. Dynamische Prädikate für Datensätze

Der Zugriff auf Daten wird mit einem dynamischen Prädikat realisiert, das exportiert wird und als Parameter einen Schlüssel und einen Datensatz erhält. Dadurch wird gewährleistet, dass der Benutzer nur auf Einträge mit existierenden Schlüsseln zugreifen kann.

```
$en$ :: $EN$Key -> $EN$ -> Dynamic
$en$ key obj
  | $en$KeyToKey key =:= $en$KeyToKey ($en$Key obj) = $en$Entry obj
```

Sollen die Daten in Dateien gespeichert werden, so wird die Funktion

```
$en$Entry :: $EN$ -> Dynamic
$en$Entry = persistent "file:$EN$DB"
```

generiert. ENDB ist der Name des Verzeichnisses, in dem die Daten für die Tabelle EN abgelegt werden.
Möchte man eine richtige Datenbank verwenden, so werden die folgenden Funktionen generiert.

en`Entry :: EN -> Dynamic`
en`Entry = persistent1 "db:`*file*`" `*en*`Spec`

en`Spec :: DBSpec EN`
en`Spec = cons`*AttributeAnzahl* *EN* `(int ...)`

In `(int ...)` werden die Funktionen für die einzelnen Wertebereiche der Spalten eingesetzt, die außer für `int`, `float`, `string`, `bool` und `date` vom Benutzer definiert werden müssen.

Da die Funktionen *en*`Entry` und *en*`Spec` nur intern verwendet werden, werden sie nicht exportiert.

Ein Beispiel ist der Code für das dynamische Prädikat für *Veranstaltung*:

```
veranstaltung :: VeranstaltungKey -> Veranstaltung -> Dynamic
veranstaltung key obj
  | (=:=) (veranstaltungKeyToKey key)
          (veranstaltungKeyToKey (veranstaltungKey obj)) =
              veranstaltungEntry obj

veranstaltungEntry :: Veranstaltung -> Dynamic
veranstaltungEntry = persistent "file:VeranstaltungDB"
```

bzw.

```
veranstaltungEntry :: Veranstaltung -> Dynamic
veranstaltungEntry = persistent1 "db:file" veranstaltungSpec

veranstaltungSpec :: DBSpec Veranstaltung
veranstaltungSpec = cons5 Veranstaltung (int int int string int)
```

4.5.5. Dynamische Prädikate für Rollen

Aus jedem Rollenbezeichner wird ein dynamisches Prädikat generiert. Für den Benutzer gibt es nur Unterschiede in der Anzahl der Parameter, sonst ist die Verwendung für alle Arten von Beziehungen gleich. Aus diesen Prädikaten lassen sich leicht neue Prädikate zusammensetzen, um Anfragen an die Datenbank zu bilden, wie am Beispiel `schueler` in Abschnitt 4.7 gezeigt wird. EN_1 ist der Name der ersten Entität in Leserichtung der Rolle, EN_2 der Name der zweiten Entität.

rollenbezeichner `::` EN_1`Key -> `EN_2`Key -> Dynamic`

Die Implementierung unterscheidet sich aber je nach Beziehungsart.

Für eine komplex-komplexe Beziehung (ohne eine bestimmte Kardinalität > 1), umgesetzt durch eine für den Benutzer nicht sichtbare zusätzliche Relation, sieht die Implementierung für ein dynamisches Prädikat für eine Rolle je nach Richtung mit oder ohne `flip` wie folgt aus:

rollenbezeichner `= [flip]` *beziehungsname*

Eine Tabelle, die eine Beziehung realisiert, wird implementiert durch den Datentypen und ein dynamisches Prädikat wie für normale Entitäten in den Abschnitten 4.5.2 und 4.5.4 beschrieben, allerdings werden diese Funktionen nicht exportiert. Als Name wird der Name der Beziehung verwendet.
Mit bestimmter Kardinalität > 1 wie im Beispiel in der Beziehung *Zugehoerigkeit* hat das Prädikat entsprechend mehr Parameter und sucht so mehrere verschiedene Einträge. Beispiele sind die folgenden generierten Funktionen:

```
nimmt_teil :: StudentKey -> VeranstaltungKey -> Dynamic
nimmt_teil = teilnahme

wird_besucht :: VeranstaltungKey -> StudentKey -> Dynamic
wird_besucht = flip teilnahme

besteht_aus :: GruppeKey -> StudentKey -> StudentKey -> StudentKey
            -> Dynamic
besteht_aus key key1 key2 key3 =
  zugehoerigkeit key1 key
    <> zugehoerigkeit key2 key
    <> zugehoerigkeit key3 key
    |> ((key1 /= key2) && (key1 /= key3) && (key2 /= key3))
```

Sonst ist die Beziehung durch einen Fremdschlüssel realisiert, der überprüft wird:

rollenbezeichner `key1 key2 =`
  en`2 key2 f2`
  `|&>` *en*`1KeyToKey key1 =:=` *en*2*EN*1_*Beziehungsname*`_Key f2`
  `where f2 free`

Die private Funktion *en*2*EN*1_*Beziehungsname*`_Key` holt den Fremdschlüssel, der aus der Relation $EN1$ stammt, aus der Relation $EN2$. Ein Beispiel ist die Umsetzung der Rolle „Dozent hält Veranstaltung“

```
haelt :: DozentKey -> VeranstaltungKey -> Dynamic
haelt key1 key2 =
  veranstaltung key2 f2
    |&> dozentKeyToKey key1 =:= veranstaltungDozent_Veranstalten_Key f2
```

```
  where f2 free
```

4.5.6. Datenbankzugriffs-Funktionen

Eine Entität kann mehrere Beziehungen zu anderen haben, also müssen alle Beziehungen einer Entität bei der Codegenerierung für diese berücksichtigt werden, d.h. bei Funktionen für das Erstellen (new), einer Änderung (update) und das Löschen (delete) eines Eintrags.

Get

Die allgemeine Funktion

```
getEntry :: k -> (k -> a -> Dynamic) -> Trans a
```

bekommt einen Schlüssel und ein dynamische Prädikat als Parameter und sucht damit den passenden Eintrag aus der Datenbank. Gibt es keinen Eintrag mit dem Schlüssel, wird ein `KeyNotExistsError` ausgegeben.
Für den Benutzer wird für jede Entität eine eigene Funktion generiert, die einen passenden Schlüssel als Parameter bekommt und damit den Eintrag aus der Datenbank holt. Mit dem Schlüssel und dem dynamischen Prädikat für die Entität wird die Funktion `getEntry` aufgerufen.

```
get EN :: ENKey -> Trans EN
get EN key = getEntry key en
```

Ein generiertes Beispiel für Einträge in der Tabelle *Veranstaltung*:

```
getVeranstaltung :: VeranstaltungKey -> Trans Veranstaltung
getVeranstaltung key = getEntry key veranstaltung
```

New

Allgemein wird ein neuer Eintrag in die Datenbank erstellt (`newEntry`), indem ein noch nicht vorhandener Schlüssel ermittelt wird (`newDBKey`), unter dem der Eintrag mit der Funktion gespeichert wird. Ist die Tabelle leer, so wird als Schlüssel 1 gewählt, sonst der größte vorhandene Schlüssel + 1.

```
newDBKey :: (a -> Key) -> (a -> Dynamic) -> IO Key
newDBKey keyf pred = do
  entries <- getDynamicSolutions pred
  return (if null entries then 1 else maxlist (map keyf entries) + 1)
```

```
newEntry :: (a -> Key) -> (a -> Key -> a) -> (a -> Dynamic) -> a
         -> Trans a
newEntry keyf keyset pred entry = trans $ do
  k <- newDBKey keyf pred
  let e = keyset entry k
  assert (pred e)
  returnTrans e
```

Als Argumente werden `newDBKey` eine Funktion, die aus einem Eintrag den Schlüssel als `Key` holt, und das Prädikat, über das die Einträge geholt werden. Da in `newEntry` `newDBKey` aufgerufen wird, bekommt diese Funktion ebenfalls eine Funktion, um den Schlüssel aus einem Eintrag zu holen. Außerdem wird eine Funktion, die den neuen Schlüssel in den übergebenen Eintrag einfügt, und wieder das dynamische Prädikat für den Zugriff auf die Daten benötigt. Der neue Eintrag wird als Aktion der Trans-Monade zurückgegeben.

Zusätzliche Tabellen haben keine künstlichen Schlüssel. Die Funktion zum Eintragen ist etwas einfacher:

```
newEntryR :: a -> b -> (a -> b -> Dynamic) -> Trans ()
newEntryR key1 key2 pred = do
  assert $ pred key1 key2
  returnTrans ()
```

Allerdings muss vor dem Eintragen sichergestellt werden, dass alle Vorgaben eingehalten werden. Deswegen wird für jede Entität eine Funktion generiert, in der vor dem Eintragen die jeweils benötigten Testfunktionen ($test_1$, ..., $test_k$) sequentiell ausgeführt werden. Ist eine Vorgabe nicht erfüllt, wird der Fehler dieser Testfunktion durchgeschleift und die Datenbank nicht verändert. Außerdem ist die new-Funktion die einzige Möglichkeit, neue Einträge in der Datenbank anzulegen, denn die Konstruktoren für Einträge werden nicht exportiert. Dadurch können nur Einträge in die Datenbank gelangen, die alle Vorgaben erfüllen.

Gibt es keine Beziehung mit einem Minimum, das für die Entität überprüft werden muss, so wird der Code nach der folgenden Vorlage generiert:

```
new EN :: attrTyp_1 -> ... -> attrTyp_n -> Trans EN
new EN attr_1_p ... attr_n_p =
  test_1 >>- ... >>- test_k
    >>- newEntry
            (enKeyToKey . enKey)
            setENKey
            enEntry
            (EN 0 attr_1_p ... attr_n_p)
```

Als Parameter werden die Attribute der Entität und die eingefügten Fremdschlüsseln ($attrTyp_1$, ..., $attrTyp_n$) übergeben. Wenn Nullwerte für ein Attribut zugelassen werden, ist dieser Typ mit `Maybe` versehen. Das gilt auch für Attribute, für die ein Standardwert angegeben ist. Dieser wird automatisch eingetragen, wenn `Nothing` übergeben wird. Als Ergebnis wird der erstellte Eintrag in einer Aktion der Trans-Monade zurückgegeben.

Die erweiterte Vorlage für die Codegenerierung ist für den allgemeinsten Fall gültig und kann durch Weglassen von Teilen für Entitäten mit jeder Art von Beziehung verwendet werden.

```
newEN :: attrTyp_1 -> ... -> attrTyp_n
        -> en_1Key -> ... -> en_1Key -> [en_1Key]
        -> ...
        -> en_mKey -> ... -> en_mKey -> [en_mKey]
        -> Trans EN
newEN attr_1_p ... attr_n_p
        k11 ... k1n_1 ks1
        ...
        km1 ... kmn_m ksm =
  test_1 >>- ... >>- test_k
    >>- newEntry
            (enKeyToKey . enKey)
            setENKey
            enEntry
            (EN 0 attr_1_p ... attr_n_p)
    >>~ \entry ->
        newEntryR k11 (enKey entry) beziehungsname
          >>- ...
          >>- newEntryR kmn_m (enKey entry) beziehungsname
```

Außer den Attributen der Entität und den eingefügten Fremdschlüsseln werden für jede der m mit der Entität verbundenen $(0,max_1):(min_2,max_2)$-Beziehungen weitere Parameter übergeben. Gilt $min_2 > 0$, so werden min_2 Schlüssel der anderen Seite als Parameter benötigt (`kij`, $i, j \in \mathbb{N}, i \leq m, j \leq min_2$). Gilt max_2 - $min_2 > 0$, so wird eine Liste von Parametern benötigt, deren Länge im Programm geprüft wird. In diesem Fall können Einträge in die Relation nur zusammen mit mindestens min_2 und höchstens max_2 Einträgen in die zusätzliche Relation, durch die die beiden in Beziehung stehenden Relationen verbunden werden, zugelassen werden.

Zu überprüfen ist:

- min_2 + Länge der zusätzlichen Liste $\leq max_2$

- keine doppelten zusätzlichen Parameter
- Eindeutigkeit der Einträge in Verbindungs-Relationen
- Existenz von Fremdschlüsseln
- Für jeden Fremdschlüssel, der als Parameter übergeben wird, muss (Vorkommen dieses Schlüssels in Relation) $< max_1$ gelten, falls es eine Beschränkung gibt.
- Eindeutigkeit von Attributen mit der Eigenschaft `Unique`

Die Einhaltung der Wertebereiche der Attribute wird durch das Typsystem von Curry sichergestellt. Es werden einerseits die Eigenschaften, die ein Benutzer den Attributen (`Unique`, Wertebereiche) und Beziehungen (Kardinalitäten) direkt geben kann, und andererseits die automatisch eingefügten Fremdschlüssel auf Existenz geprüft. Also wird alles geprüft, was bekannt ist.

Im Folgenden werden die Testfunktionen beschrieben. Beim Hinzufügen eines Eintrags mit einem eindeutigen Attribut muss festgestellt werden, ob es diesen Wert schon gibt. Der Funktion

```
unique :: a -> (b -> a) -> (b -> Dynamic) -> Trans ()
unique attr selector pred = do
  entries <- getDynamicSolutions (\info ->
               pred info |> attr == selector info)
  if null entries
     then returnTrans ()
     else return
            (TransError UniqueError
              ("entry for unique attribute "
                  ++show pred++"."++show attr++" already exists"))
```

wird der auf Eindeutigkeit zu prüfende Wert, der Selektor für das Attribut und das dynamische Prädikat der Relation übergeben. Gibt es den Wert schon, so wird ein `UniqueError` zurückgegeben. Für zusätzliche Relationen muss die Eindeutigkeit der beiden Fremdschlüssel zusammen geprüft werden:

```
unique2 :: a -> b -> (c -> a) -> (c -> b) -> (c -> Dynamic) -> Trans ()
```

Dafür werden beide Werte, die passenden Selektoren und das dynamische Prädikat übergeben. Im Fehlerfall wird auch ein `UniqueError` zurückgegeben. Für jeden Fremdschlüssel wird kontrolliert, ob dieser als Schlüssel existiert.

```
existsDBKey :: k -> (a -> k) -> (a -> Dynamic) -> Trans ()
```

Wenn nicht, so wird ein `KeyNotExistsError` zurückgegeben. Beschränkungen der Fremdschlüssel werden mit der Funktion

```
maxTest :: Int -> a -> (b -> a) -> (b -> Dynamic) -> Trans ()
```

geprüft, die ein Maximum, den Wert, den Selektor für das Attribut und das dynamische Prädikat bekommt und bei Überschreitung des Maximums einen `MaxError` zurückgibt. Die Funktion

```
duplicatePTest :: [a] -> Trans ()
```

prüft mit `List.nub`, ob es in einer Liste doppelte Elemente gibt, um die Parameter einer mit der Entität verbundenen $(0,max_1):(min_2,max_2)$-Beziehung auf doppelte Schlüssel zu überprüfen. Dann wird ein `DuplicateKeyError` ausgegeben. Für die Einhaltung eines Maximums einer $(0,max_1):(min_2,max_2)$-Beziehung ist die Funktion

```
maxPTest :: Int -> [a] -> Trans ()
```

zuständig. Ist die Liste der Parameter länger als das Maximum, so wird ein `MaxError` zurückgegeben.

Als Beispiele folgen die new-Funktionen für Veranstaltung und Gruppe. Bei einer Veranstaltung gibt es einen Standardwert, der eingetragen wird, wenn kein Wert angegeben wird. Überprüft werden die eindeutigen Attribute „Nr“ und „Titel“ und die Existenz des Fremdschlüssels. Eine Gruppe ist durch eine Beziehung, die ein Minimum von 3 hat, mit Student verbunden. Daher müssen drei verschiedene existierende Schlüssel für Studenten angegeben werden. Das Maximum ist auch 3, also gibt es keine Liste für weitere Schlüssel von Studenten. Der neue Eintrag wird erstellt und die Verbindungen zwischen den Studenten und der Gruppe werden in die Relation *Zugehoerigkeit* eingetragen.

```
newVeranstaltung :: DozentKey -> Int -> String -> Maybe Int
                 -> Trans Veranstaltung
newVeranstaltung dozent_Veranstalten_Key_p nr_p titel_p sWS_p =
  unique nr_p veranstaltungNr veranstaltungEntry
    >>- unique titel_p veranstaltungTitel veranstaltungEntry
    >>- existsDBKey dozent_Veranstalten_Key_p dozentKey dozentEntry
    >>- newEntry (veranstaltungKeyToKey . veranstaltungKey)
                 setVeranstaltungKey veranstaltungEntry
                 (Veranstaltung 0
                    (dozentKeyToKey dozent_Veranstalten_Key_p)
                    nr_p titel_p
                    (case sWS_p of (Just v) -> v
                                   Nothing -> 4))
```

```
newGruppe :: String -> StudentKey -> StudentKey -> StudentKey
          -> Trans Gruppe
newGruppe termin_p k11 k12 k13 =
  duplicatePTest (k11 : k12 : k13 : [])
    >>- existsDBKey k11 studentKey studentEntry
    >>- existsDBKey k12 studentKey studentEntry
    >>- existsDBKey k13 studentKey studentEntry
    >>- newEntry (gruppeKeyToKey . gruppeKey)
                 setGruppeKey gruppeEntry
                 (Gruppe 0 termin_p)
    >>~ \entry ->
           newEntryR k11 (gruppeKey entry) zugehoerigkeit
             >>- newEntryR k12 (gruppeKey entry) zugehoerigkeit
             >>- newEntryR k13 (gruppeKey entry) zugehoerigkeit
             >>- returnTrans entry
```

Update

Um einen existierenden Eintrag zu verändern, holt man ihn mit der get-Funktion aus der Datenbank, ändert mit setter-Funktionen die einzelnen Werte außer dem Schlüssel und ändert dann mit der update-Funktion den Eintrag in der Datenbank. Dafür wird der alte Eintrag entfernt und durch den veränderten Eintrag ersetzt.

```
updateEntry :: (a -> k) -> (k -> a -> Dynamic) -> a -> Trans ()
updateEntry keyf pred entry = trans $ do
  deleteDBEntry keyf pred entry
  assert (pred (keyf entry) entry)
  returnTrans ()

deleteEntry :: (a -> k) -> (k -> a -> Dynamic) -> a -> IO ()
deleteDBEntry keyf pred entry = seq pred $ seq (keyf entry) $ do
  entries <- getDynamicSolutions (\info -> pred (keyf entry) info)
  mapIO_ (\info -> retract (pred (keyf entry) info)) entries
```

Mit der allgemeinen Update-Funktion sieht die Code-Vorlage wie folgt aus:

```
update EN :: EN -> Trans ()
update EN en_p =
  test_1 >>- ... >>- test_n
    >>- updateEntry enKey en en_p
```

Wie vor dem Einfügen neuer Einträge in die Datenbank müssen die Vorgaben auch vor der Änderung bestehenden Einträge überprüft werden. Dazu werden vor dem Upda-

te für jede Relation die benötigten Testfunktionen $test_1$, ..., $test_n$ sequentiell aufgerufen. Mit der Funktion

```
uniqueUpdate :: b -> (b -> k) -> (b -> a) -> (k -> b -> Dynamic)
             -> Trans ()
```

wird geprüft, ob ein eindeutiges Attribut dies auch nach dem Update noch ist. Entweder gibt es diesen Wert noch nicht oder er wurde nicht verändert. Sonst wird ein `UniqueError` zurückgegeben. Für Fremdschlüssel, die durch ein Maximum beschränkt sind, muss geprüft werden, ob die Beschränkung nach der Änderung noch eingehalten wird. Sonst gibt es einen `MaxError`.

```
maxTestUpdate :: Int -> b -> (b -> k) -> (b -> a) -> (k -> b -> Dynamic)
              -> Trans ()
```

Außerdem muss wie in der new-Funktion die Existenz von Fremdschlüsseln geprüft werden.

Ein Beispiel für eine generierte update-Funktion ist:

```
updateVeranstaltung :: Veranstaltung -> Trans ()
updateVeranstaltung veranstaltung_p =
  uniqueUpdate veranstaltung_p veranstaltungKey
               veranstaltungNr veranstaltung
   >>- uniqueUpdate veranstaltung_p veranstaltungKey
                    veranstaltungTitel veranstaltung
   >>- existsDBKey
        (DozentKey (veranstaltungDozent_Veranstalten_Key veranstaltung_p))
        dozentKey dozentEntry
   >>- updateEntry veranstaltungKey veranstaltung veranstaltung_p
```

Delete

Es gibt verschiedene Konzepte für das Löschen von Einträgen. Am Leichtesten umzusetzen ist ein einfaches Löschen des Eintrags, ohne Abhängigkeiten zu berücksichtigen, aber das kann zu inkonsistenten Daten führen, wenn der Benutzer einen Fehler macht. Um Fehler zu vermeiden ist es besser, alle Einträge zu entfernen, die den zu löschenden Eintrag als Fremdschlüssel verwenden. Diese können auch wieder als Fremdschlüssel verwendet werden usw., so dass viele Abfragen notwendig sind und eventuell mehrere Einträge gelöscht werden. Dabei muss der Benutzer alle Löschungen überblicken und sich sicher sein, dass das auch so beabsichtigt ist.
Hier wird das Löschen nicht implementiert. Möchte man einen Eintrag löschen, geht das ähnlich wie in Abschnitt 4.5.8.

4.5.7. Korrektheits- und Konsistenzcheck

Das Ziel der automatischen Generierung von Funktionen, die Datenbankoperationen nur dann ausführen, wenn keine Integritätsbedingungen verletzt werden, sollte garantieren, dass die Datenbank immer in einem konsistenten Zustand ist. Trotzdem kann es nützlich sein, einen Konsistenzcheck durchzuführen, um unvorhergesehene Inkonsistenzen zu finden. Bei der Verwendung von Dateien zur Speicherung der Daten könnten beispielsweise die Dateien verändert oder gelöscht worden sein. Auch beim Löschen von Daten können leicht Inkonsistenzen entstehen, etwa wenn ein Eintrag entfernt wird, auf den ein Fremdschlüssel verweist.

Konsistenz der Daten bedeutet, dass alle Integritätsbedingungen aus Abschnitt 4.2.1 und die Vorgaben des Benutzers erfüllt sind. Durch die Verwendung der Programmiersprache Curry ist die Einhaltung der Wertebereiche der Attribute bereits gesichert. Mit der Funktion

```
checkData :: Trans ()
```

werden alle Tabellen sequentiell geprüft. Für jede Entität werden dafür die Funktionen

```
checkEN :: Trans ()
checkEN = do
  entries <- getDynamicSolutions enEntry
  mapTrans_ checkENEntry entries

checkENEntry :: EN -> Trans ()
checkENEntry en_p = test1 >>- ... >>- testn
```

generiert, wobei `checkEN` auch einzeln aufgerufen werden kann, wenn man nur die Konsistenz einer Tabelle testen möchte. Die Funktionen $test_1$, ..., $test_n$ stehen für die für die jeweilige Tabelle notwendigen Testfunktionen aus dem Modul `Intern`, die in den folgenden Abschnitten beschrieben werden.

Eindeutige Schlüssel

Da für jeden neuen Eintrag ein noch nicht vorhandener Wert als Schlüssel gewählt wird, sollten die Schlüssel eigentlich eindeutig sein. Zur Kontrolle wird für jede benutzerdefinierte Tabelle die Funktion

```
duplicateKeyTest :: (k -> a -> Dynamic) -> Trans ()
```

aufgerufen, die aus einer Tabelle über ein dynamisches Prädikat alle Schlüssel holt und diese Liste auf doppelte Elemente mit der Funktion `List.nub`, die doppelte Elemente aus einer Liste entfernt, überprüft. Wird ein Schlüssel mehrfach gefunden, so wird ein `DuplicateKeyError` ausgegeben.

Existenz von Fremdschlüsseln

Um die Existenz von Fremdschlüsseln zu prüfen, müssen alle ensprechenden Einträge aus den Tabellen, die Fremdschlüssel enthalten, mit der Funktion

```
existsDBKey :: k -> (a -> k) -> (a -> Dynamic) -> Trans ()
```

(siehe auch Abschnitt 4.5.6) geprüft werden. Zu jedem in der Datenbank enthaltenen Fremdschlüssel wird also nach dem passenden Eintrag gesucht. Wird ein solcher nicht gefunden, so wird ein `KeyNotExistsError` ausgegeben.

Eindeutigkeit von Attributen

Alle Werte für Attribute, die als `Unique` gekennzeichnet sind, müssen auf Eindeutigkeit geprüft werden. Dazu wird die Funktion

```
uniqueC :: b -> (b -> k) -> (b -> a) -> (k -> b -> Dynamic) -> Trans ()
```

verwendet, die als Parameter ein Objekt, also einen Eintrag, einen Selektor für das zu prüfende Attribut und ein dynamisches Prädikat erhält. Wenn es einen Wert mehr als einmal gibt, so wird ein `UniqueError` ausgegeben.

Eindeutige Einträge in Beziehungs-Relationen

Für in der Transformation generierte zusätzliche Relationen wird die Funktion

```
unique2C :: a -> b -> (c -> a) -> (c -> b) -> (c -> Dynamic) -> Trans ()
```

verwendet, um zu prüfen, ob es doppelte Einträge gibt. In diesem Fall wird ebenfalls ein `UniqueError` ausgegeben.

Kardinalitäten

Ist für eine an einer Beziehung beteiligten Entität Minimum oder Maximum angegeben, muss die Einhaltung dieser Vorgaben geprüft werden. Die Fälle (0,1) und (1,1) werden über die Fremdschlüssel durch das Typsystem von Curry gesichert, also sind Tests nur für größeres Minimum bzw. Maximum erforderlich. Die Funktionen

```
minTestC :: Int -> a -> (b -> a) -> (b -> Dynamic) -> Trans ()
```

```
maxTestC :: Int -> a -> (b -> a) -> (b -> Dynamic) -> Trans ()
```

bekommen jeweils das Minimum bzw. Maximum, einen Schlüssel der Seite der Beziehung, für die die Vorgabe gilt, die Selektor-Funktion, die aus einem Eintrag der anderen Seite den dazugehörigen Fremdschlüssel holt und das Prädikat für den Zugriff auf die Daten der anderen Seite der Beziehung. Damit werden alle Vorkommen des Schlüssels als Fremdschlüssel ermittelt und durch Minimum bzw. Maximum geprüft. Werden die Vorgaben nicht eingehalten, so wird ein `MinError` bzw. `MaxError` ausgegeben.

Beispiel

Für die Entitäten *Veranstaltung* und *Gruppe* und die generierte Entität zu der Beziehung *Teilnahme* aus dem Beispiel aus Abbildung 2.1 sieht der generierte Code für den Konsistenzcheck wie folgt aus:

```
checkAllData :: Trans ()
checkAllData =
  checkZugehoerigkeit
    >>- checkTeilnahme
    >>- checkStudent
    >>- checkVeranstaltung
    >>- checkDozent
    >>- checkGruppe

checkGruppe :: Trans ()
checkGruppe = do
  entries <- getDynamicSolutions gruppeEntry
  mapTrans_ checkGruppeEntry entries

checkVeranstaltung :: Trans ()
checkVeranstaltung = do
  entries <- getDynamicSolutions veranstaltungEntry
  mapTrans_ checkVeranstaltungEntry entries

checkTeilnahme :: Trans ()
checkTeilnahme = do
  entries <- getDynamicSolutions teilnahmeEntry
  mapTrans_ checkTeilnahmeEntry entries

checkGruppeEntry :: Gruppe -> Trans ()
checkGruppeEntry gruppe_p =
  duplicateKeyTest gruppe
    >>- maxTestC 3 (gruppeKeyToKey (gruppeKey gruppe_p))
                   zugehoerigkeitGruppe_Zugehoerigkeit_Key
                   zugehoerigkeitEntry
```

```
    >>- minTestC 3 (gruppeKeyToKey (gruppeKey gruppe_p))
                   zugehoerigkeitGruppe_Zugehoerigkeit_Key
                   zugehoerigkeitEntry

checkVeranstaltungEntry :: Veranstaltung -> Trans ()
checkVeranstaltungEntry veranstaltung_p =
  duplicateKeyTest veranstaltung
    >>- uniqueC veranstaltung_p veranstaltungNr veranstaltung
    >>- uniqueC veranstaltung_p veranstaltungTitel veranstaltung
    >>- existsDBKey
          (DozentKey
            (veranstaltungDozent_Veranstalten_Key veranstaltung_p))
          dozentKey dozentEntry

checkTeilnahmeEntry :: Teilnahme -> Trans ()
checkTeilnahmeEntry teilnahme_p =
  existsDBKey (teilnahmeStudent_Teilnahme_Key teilnahme_p)
              studentKey studentEntry
    >>- existsDBKey (teilnahmeVeranstaltung_Teilnahme_Key teilnahme_p)
                    veranstaltungKey veranstaltungEntry
    >>- unique2C
         (studentKeyToKey (teilnahmeStudent_Teilnahme_Key teilnahme_p))
         (veranstaltungKeyToKey
           (teilnahmeVeranstaltung_Teilnahme_Key teilnahme_p))
         (studentKeyToKey . teilnahmeStudent_Teilnahme_Key)
         (veranstaltungKeyToKey . teilnahmeVeranstaltung_Teilnahme_Key)
         teilnahmeEntry
```

4.5.8. Zyklische Beziehungen

Da hier alle aus dem ER-Diagramm bekannten Vorgaben abgesichert werden sollen, kann das bei zyklischen Beziehungen dazu führen, dass das Eintragen in die Relationen im Zyklus nicht möglich ist, da für jede Relation ein Fremdschlüssel einer anderen Relation benötigt wird, die wiederum einen Fremdschlüssel benötigt usw. Abbildung 4.5 soll zyklische ER-Diagramme andeuten. Die Lösung für dieses Problem liegt darin, das ER-Diagramm an einer Stelle zu modifizieren, indem statt der Kardinalität „(1,1)“ die schwächere Kardinalität „(0,1)“ gewählt wird, und selbst als Erweiterung des generierten Codes ein „Bulk Update“ zu programmieren, d.h. Eintragen in alle beteiligten Tabellen und Schließen des Zyklus durch Eintragen des letzten Fremdschlüssels in einer Transaktion.
Eine Verbesserung bzw. Aufgabe für die Weiterentwicklung dieser Arbeit wäre, Zyklen zu erkennen und dies dem Benutzer mitzuteilen oder sogar aufzulösen und das „Bulk Update“ zu generieren.

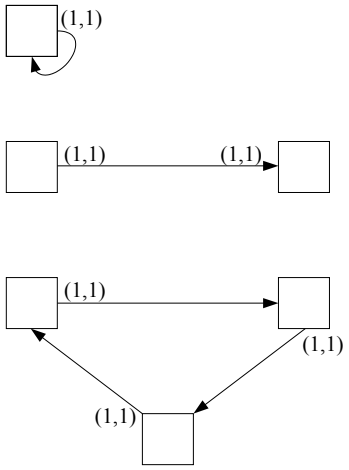

Abbildung 4.5.: Zyklen

Zyklen stellen aber nur ein Problem für den Benutzer des generierten Codes dar, die Generierung selbst läuft ohne Probleme auch mit Zyklen, nur Einträge sind dann eventuell nicht möglich. Beziehungen einer Entität mit sich selbst werden zwar nicht von Umbrello unterstützt, aber für die Generierung wäre dies auch kein Problem.

4.6. Codegenerierung

Aus dem transformierten ER-Modell soll nun ein Programm generiert werden. Die Bibliothek `AbstractCurry` implementiert die Darstellung eines Curry-Programms in Curry als abstrakten Syntaxbaum und unterstützt so die Codegenerierung. Zusammen mit der Bibliothek `AbstractCurryPrinter`, mit der sich die abstrakte Darstellung in lesbaren Code ausgeben lässt, können Programme erstellt werden, die in anderen Programmen importiert und verwendet werden können.

Die Codegenerierung befindet sich im Modul `CodeGeneration` und wird mit der Funktion

```
start :: Option -> String -> String -> IO ()
```

aus dem Modul `Main` gestartet, die als Parameter Optionen, den Pfad der mit Umbrello erstellten XML-Datei und den Pfad, unter dem die Curry-Datei mit dem generierten Code gespeichert werden soll, bekommt. Der Name der Datei ist der Name des ER-Diagramms. Die Optionen werden als Tupel übergeben, das als Typsynonym

```
type Option          = (Storage, ConsistencyTest)
data Storage         = Files DBPath | DB
```

```
type DBPath          = String
data ConsistencyTest = WithConsistencyTest | WithoutConsistencyTest
```

definiert wird. Dadurch sind die Optionen leicht erweiterbar. Momentan kann man wählen, ob die Daten in Dateien oder in einer richtigen Datenbank gespeichert werden und ob Konsistenztests generiert werden sollen.
Die Verwendung einer Datenbank muss eventuell noch angepasst werden.
Bei Speicherung in Dateien muss man einen Pfad angeben, unter dem die Datenbank angelegt wird.

Der Inhalt der XML-Datei wird eingelesen und in die Datenstruktur `ERD` konvertiert, die transformiert wird, um daraus den Code zu generieren und diesen in eine Datei zu schreiben.

4.6.1. AbstractCurry

Die Curry-Bibliothek `AbstractCurry`[1] unterstützt die Meta-Programmierung in Curry. Im Folgenden werden nur die in dieser Arbeit verwendeten Teile davon vorgestellt. Der Datentyp

```
data CurryProg =
  CurryProg String [String] [CTypeDecl] [CFuncDecl] [COpDecl]
```

repräsentiert ein Curry-Programm in Curry. Die Parameter sind der Modulname und Listen von importierten Modulen, Typ-, Funktions- und Operator-Deklarationen.

```
data CTypeDecl = CType    QName CVisibility [CTVarIName] [CConsDecl]
               | CTypeSyn QName CVisibility [CTVarIName] CTypeExpr

type QName = (String,String)
data CVisibility = Public | Private
type CTVarIName = (Int,String)

data CConsDecl = CCons QName Int CVisibility [CTypeExpr]

data CTypeExpr = CTVar CTVarIName
               | CFuncType CTypeExpr CTypeExpr
               | CTCons QName [CTypeExpr]
```

Typ-Deklarationen sind entweder Datentypen oder Typsynonyme.
Alle Namen sind qualifiziert (`QName`), um Namenskonflikte zu vermeiden, und bestehen aus dem Modulnamen und dem unqualifizierten Namen. Die Sichtbarkeit (`CVisibility`)

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

kann entweder `Public` oder `Private` sein und entscheidet so über den Export.
Eine Typvariable (`CTVarIName`) wird repräsentiert durch einen Index, der hier keine Bedeutung hat, da der generierte Code nur in einer Datei gespeichert werden soll und nicht als Datenstruktur weiterverarbeitet wird, und einen Namen, der im Programm verwendet wird.
Ein Datentyp und ein Typsynonym haben jeweils einen Namen, eine Sichtbarkeit und eine Liste von Typvariablen. Außerdem hat ein Datentyp noch eine Liste von Konstruktor-Deklarationen und ein Typsynonym einen Typausdruck.
Eine Konstruktor-Deklaration besteht aus dem Namen, der Stelligkeit und einer Liste der Argumenttypen des Konstruktors als `CTypeExpr`.

Ein Typausdruck ist entweder eine Typvariable, ein Funktionstyp oder ein Aufruf eines Typ-Konstruktors.

```
data CFuncDecl = CFunc QName Int CVisibility CTypeExpr CRules

data CRules = CRules CEvalAnnot [CRule]
            | CExternal String

data CEvalAnnot = CFlex | CRigid | CChoice

data CRule = CRule [CPattern] [(CExpr,CExpr)] [CLocalDecl]

data CPattern = CPVar CVarIName
              | CPLit CLiteral
              | CPComb QName [CPattern]

data CLiteral = CIntc   Int
              | CFloatc Float
              | CCharc  Char

data CLocalDecl = CLocalFunc CFuncDecl
                | CLocalPat  CPattern CExpr [CLocalDecl]
                | CLocalVar  CVarIName
type CVarIName = (Int,String)

data CExpr = CVar      CVarIName
           | CLit      CLiteral
           | CSymbol   QName
           | CApply    CExpr CExpr
           | CLambda   [CPattern] CExpr
           | CLetDecl  [CLocalDecl] CExpr
           | CDoExpr   [CStatement]
           | CListComp CExpr [CStatement]
```

```
           | CCase     CExpr [CBranchExpr]

data CStatement = CSExpr CExpr
                | CSPat CPattern CExpr
                | CSLet [CLocalDecl]

data CBranchExpr = CBranch CPattern CExpr
```

Funktions-Deklarationen bestehen aus dem Namen, der Stelligkeit, dem Typ und den Regeln der Funktion.
Eine Regel ist entweder eine Liste von Programmregeln mit einer Auswertungsweise oder sie ist extern definiert. Hier wird immer die Standard-Auswertungsweise `CFlex` verwendet.
Eine Programmregel besteht aus einer Liste von Pattern für die linken Regelseiten, einer Liste von Guards zusammen mit den zugehörigen rechten Regelseiten und einer Liste von lokalen Deklarationen. Ein Pattern ist entweder eine Pattern-Variable, ein Literal, d.h. eine Integer-, Float- oder Character-Konstante, oder eine Konstruktor-Anwendung. Eine lokale Deklaration kann eine lokale Funktionsdeklaration, Patterndeklaration oder Deklaration einer freie Variable sein.
Ein Curry-Ausdruck (`CExpr`) ist eine Variable, ein Literal, ein definiertes Symbol mit Modul und Name, eine Applikation, eine Lambda-Abstraktion, eine lokale let-Deklaration, ein do-Ausdruck, eine list comprehension oder ein case-Ausdruck.
Der Datentyp `CStatement`, der aus einem Ausdruck vom Typ `IO` oder `Bool`, einer Pattern-Definition oder einer lokalen let-Deklaration besteht, repräsentiert Statements in do-Ausdrücken und list comprehensions. Der für case-Ausdrücke verwendete Datentyp `CBranchExpr` steht für ein Pattern zusammen mit einem Ausdruck.
Operator-Deklarationen werden hier nicht verwendet.

Die Bibliothek `AbstractCurryPrinter` bietet die Funktion

```
showProg :: CurryProg -> String
```

um ein generiertes Programm in eine lesbare String-Darstellung umzuwandeln. Dieser String lässt sich mit der Prelude-Funktion

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

in eine Datei schreiben und somit wie ein handgeschriebenes Programm verwenden.

4.6.2. Vorgehensweise

Im Folgenden wird das Modul `CodeGeneration` beschrieben. Die Funktion

```
erd2code :: Option -> ERD -> CurryProg
```

nimmt ein transformiertes ER-Diagramm und macht daraus ein Curry-Programm in AbstractCurry-Darstellung unter Berücksichtigung der Optionen, die durch die meisten Funktionen durchgeschleift werden, um weitere Optionen später schnell hinzufügen zu können. Die verschiedenen Teile des Programms werden durch spezielle Funktionen erstellt. Das Programm importiert die Module `Dynamic`, `Intern`, `Trans` und alle Module mit benutzerdefinierten Datentypen für die Wertebereiche der Attribute, die mit der Funktion

```
getImports :: Entity -> [String]
```

aus allen Entitäten herausgesucht werden und aus denen durch die alphabetische Sortierung mit `FiniteMap.fmSortBy (<)` alle doppelten Vorkommen entfernt werden.

Hilfsfunktionen

Oft verwendete und geschachtelte Konstruktoren werden durch folgende Funktionen lesbarer, hier sind die wichtigsten für die Codegenerierung:
Zu besseren Lesbarkeit der Implementierung von Typsignaturen kann man den Infix-Operator `(~>)` verwenden, der die Typen verbindet.

```
infixr 9 ~>
(~>) :: CTypeExpr -> CTypeExpr -> CTypeExpr
t1 ~> t2 = CFuncType t1 t2
```

Da alle generierten Funktionen die Auswertungsart `CFlex` haben, werden Funktionsdeklarationen mit der Funktion

```
cfunc :: (String,String) -> Int -> CVisibility -> CTypeExpr -> [CRule]
      -> CFuncDecl
cfunc name arity v t rules =
  CFunc name arity v t (CRules CFlex rules)
```

implementiert. Die Funktionsanwendung auf eine Liste von Parametern wird durch

```
applyF :: (String, String) -> [CExpr] -> CExpr
applyF f es = foldl CApply (CSymbol f) es
```

vereinfacht.

Datentypen

Für die Generierung der Datenstrukturen werden die Entitäten in benutzerdefinierte und während der Transformation generierte aufgeteilt. Mit den Funktionen

```
entity2datatype :: Option -> String -> Entity -> CTypeDecl

entity2DatatypeKey :: Option -> String -> Entity -> CTypeDecl
```

werden für die benutzerdefinierten Entitäten die Datentyp-Deklarationen für Einträge und Schlüssel erstellt. Dabei werden zu den Wertebereichen der Attribute die passenden Curry-Typen als Parameter der Konstruktoren gesucht, die auch vom Typ `Maybe` sein können, wenn Nullwerte zugelassen werden. Generierte Entitäten haben keinen künstlichen Schlüssel und ihre Datentypen werden nicht exportiert. Deswegen werden sie durch eine eigene Funktion generiert:

```
generatedEntity2Datatype :: Option -> String -> Entity -> CTypeDecl
```

Selektoren und Modifikatoren

Mit der Funktion

```
entity2selmod :: Option -> String -> Entity -> [CFuncDecl]
```

werden für jede Entität die Selektoren und Modifikatoren generiert, indem die Funktionen

```
selector :: (String, String) -> Int -> Int -> (String,String)
         -> (String, String) -> Bool -> CVisibility -> CFuncDecl

modificator :: (String, String) -> Int -> Int -> (String,String)
            -> (String, String) -> Bool -> CVisibility -> CFuncDecl
```

entsprechend den Vorgaben aus Abschnitt 4.5.3 aufgerufen werden. Wieder werden in den Typsignaturen die Attribute, für die Nullwerte zugelassen werden, mit `Maybe` versehen.

Datenbankzugriffs-Funktionen und dynamische Prädikate für Datensätze

Für jede benutzerdefinierte Entität wird die Funktion

```
entity2DBcode :: Option -> String -> [Entity] -> [Relationship] -> Entity
              -> [CFuncDecl]
```

aufgerufen, in der weitere Funktionen aufgerufen werden. Das dynamische Prädikat, das für den Benutzer exportiert wird, generiert die Funktion

```
pred :: (String, String) -> Int -> CVisibility -> CFuncDecl
```

Das private dynamische Prädikat wird durch

```
predEntry :: (String, String) -> CVisibility -> String -> CFuncDecl
```

für Speicherung in Dateien bzw.

```
predEntry1 :: (String, String)-> CVisibility -> CFuncDecl
```

```
entitySpec :: (String, String) -> [Attribute] -> CVisibility -> CFuncDecl
```

für Speicherung in einer Datenbank generiert (siehe Abschnitt 4.5.4). Die Zugriffsfunktionen für Schlüssel unterscheiden sich von den Zugriffsfunktionen für andere Attribute, da diese in ihrer Datenstruktur versteckt werden. Die Funktionen

```
entityKey :: (String, String) -> [Attribute] -> CVisibility -> CFuncDecl
```

```
entityKeyToKey :: (String, String) -> CVisibility -> CFuncDecl
```

generieren eine Zugriffsfunktion für Schlüssel, die exportiert wird, und eine private intern verwendete Funktion, die den Wert des Schlüssels aus seiner Datenstruktur holt.

Als Beispiel für die Codegenerierung wird die Generierung einer get-Funktion gezeigt.

```
getEntity :: (String, String) -> CVisibility -> CFuncDecl
getEntity (s,eName) v =
  cfunc (s,"get" ++ eName) 1 v
        ((ctcons (s, eName ++ "Key"))
           ~> (CTCons (intern "Trans") [ctcons (s,eName)]))
        [CRule [var "key"]
               [noGuard
                 (applyF (intern "getEntry")
                         [cvar "key", CSymbol (s, firstToLower eName)])]
               []]
```

Die Parameter sind der Modulname (Name des ER-Diagramms), der Name der Entität und die Sichtbarkeit. Mit dem Aufruf der Funktion `cfunc` wird eine neue Funktion deklariert, der ein Name, die Stelligkeit, die Sichtbarkeit, die Typsignatur und eine Liste von Regeln übergeben wird. Das Ergebnis wurde in Abschnitt 4.5.6 vorgestellt.

Eine new-Funktion wird mit

```
newEntity :: (String, String) -> [Attribute]
          -> [Relationship] -> CVisibility
          -> [Entity] -> [Relationship] -> CFuncDecl
```

generiert. Die Parameter sind ein Paar aus Modulname und Entitätsname, die Attribute der Entität, die Beziehungen dieser Entität, die Sichtbarkeit, alle Entitäten und alle Beziehungen. Die jeweils notwendigen Parameter für Beziehungen werden generiert. Mit der Funktion `tests` mit dem Parameter `New` werden die Aufrufe der Testfunktionen generiert, die mit `foldr` und `(>>-)` zu einer Sequenz verknüpft werden. Darauf folgen der Eintrag in die Datenbank mit der allgemeinen new-Funktion, wobei Standardwerte berücksichtigt werden müssen falls vorhanden, und bei Bedarf auch die Einträge in Relationen, die Beziehungen realisieren. Dabei ist auf die richtige Reihenfolge der Parameter zu achten. Für eine genaue Beschreibung der generierten Funktion siehe Abschnitt 4.5.6.

Eine update-Funktion wird mit

```
updateEntity :: (String, String) -> [Entity] -> [Relationship]
              -> CVisibility -> CFuncDecl
```

generiert. Für die in der generierten Funktion aufgerufenen Testfunktionen wird die Funktion `test` mit dem Parameter `Update` verwendet.

Während der Transformation generierte Entitäten werden mit der Funktion

```
generatedEntity2DBcode :: Option -> String -> Entity -> [CFuncDecl]
```

verarbeitet. Für jede solche Entität werden private Zugriffsfunktionen für die Attribute generiert, die in generiertem Code verwendet werden. Außerdem werden eine new-Funktion und die dynamischen Prädikate für den Datenzugriff generiert.

Dynamische Prädikate für Beziehungen

Zu jedem vom Benutzer angegebenen Beziehungsnamen und Rollenbezeichner soll ein dynamisches Prädikat generiert werden. Dazu teilt die Funktion

```
rel2code :: Option -> String -> [Entity] -> Relationship -> [CFuncDecl]
```

einer Beziehung die passende Funktion zu. Aus einer durch Fremdschlüssel umgesetzte Beziehung werden mit der Funktion

```
roles :: String -> Relationship -> [CFuncDecl]
```

die in Abschnitt 4.5.5 beschriebenen dynamischen Prädikate generiert. Aus einer generierten Beziehung als Teil der Umsetzung einer n:m-Beziehung durch eine zusätzliche Entität werden mit der Funktion

```
rolesR :: String -> Relationship -> [Entity] -> [CFuncDecl]
```

die entsprechenden dynamischen Prädikate generiert.

Konsistenzcheck

Nur wenn die Option `WithConsistencyTest` gewählt ist, werden Funktionen zur Überprüfung der Konsistenz der Daten in der Datenbank generiert. Die Generierung wird durch die Funktionen

```
checkAll :: String -> [Entity] -> CFuncDecl

checkEntity :: String -> Entity -> [Entity] -> [Relationship]
            -> CFuncDecl

checkEntry :: String -> Entity -> [Entity] -> [Relationship]
           -> CFuncDecl
```

implementiert, wobei `checkAll` einmal aufgerufen wird, um die Funktion `checkAllData` zu generieren, die den kompletten Konsistenzcheck startet. Die anderen beiden Funktionen werden für jede Entität aufgerufen und generieren spezielle Testfunktionen für die einzelnen Entitäten, wobei `checkEntity` eine Funktion `check`*EN* generiert, die alle Einträge einer Tabelle holt und darauf die Funktion `check`*EN*`Entry` anwendet (siehe Abschnitt 4.5.7), die von `checkEntityEntry` generiert wird. Die eigentlichen Tests werden unterschiedlich für benutzerdefinierte und während der Transformation generierte Entitäten ermittelt. Für benutzerdefinierte Entitäten wird die Funktion `tests`, die auch zur Generierung der Tests in new- und update-Funktionen verwendet wird, hier mit dem Parameter `Consistency`. Überprüft werden muss die Eindeutigkeit der Schlüssel und sonstigen eindeutigen Attribute, die Existenz aller Fremdschlüssel als Schlüssel und die Einhaltung der Minima und Maxima sofern angegeben.
Die Tests für generierte Entitäten sind immer gleich und werden durch die Funktion

```
generatedEntityTests :: String -> Entity -> [CExpr]
```

generiert. Ein Eintrag besteht immer aus zwei Fremdschlüsseln, deren Existenz als Schlüssel überprüft werden muss. Ausserdem müssen beide Fremdschlüssel zusammen eindeutig sein.

4.7. Anwendungsbeispiel

In diesem Abschnitt soll der Umgang mit dem generierten Code am Beispiel zu Abbildung 2.1 gezeigt werden. Importiert werden müssen die Module `Dynamic` und `Trans`, alle Module mit eingenen Datenstrukturen, die als Wertebereiche der Attribute verwendet werden, und das generierte Modul.

Nach der Codegenerierung ist die neue Datenbank leer und wird mit einigen Testdaten gefüllt.

```
testData = do
  s1 <- newStudent 480811 "Mueller" "Marion" Nothing
  s2 <- newStudent 111111 "Name1" "Vorname1" Nothing
  s3 <- newStudent 222222 "Name2" "Vorname2" Nothing
  newStudent 333333 "Name3" "Vorname3" Nothing
  d1 <- newDozent 1 "Hanus" "Michael"
  v1 <- newVeranstaltung (dozentKey (result d1)) 1
                         "Logikprogrammierung" Nothing
  newTeilnahme (studentKey (result s1))
               (veranstaltungKey (result v1))
  newGruppe "Montag" (studentKey (result s1))
            (studentKey (result s2)) (studentKey (result s3))
```

Der Inhalt der Datenbank ist also:

Key	**MatrikelNr**	**Name**	**Vorname**	**Email**
1	480811	Mueller	Marion	Nothing
2	111111	Name1	Vorname1	Nothing
3	222222	Name2	Vorname2	Nothing
4	333333	Name3	Vorname3	Nothing

Tabelle 4.2.: Tabelle *Student*

Key	**Nr**	**Name**	**Vorname**
1	1	Hanus	Michael

Tabelle 4.3.: Tabelle *Dozent*

Key	**Dozent_Veranstalten_Key**	**Nr**	**Titel**	**SWS**
1	1	1	Logikprogrammierung	Nothing

Tabelle 4.4.: Tabelle *Veranstaltung*

Key	**Termin**
1	Montag

Tabelle 4.5.: Tabelle *Gruppe*

Student_Zugehoerigkeit_Key	**Gruppe_Zugehoerigkeit_Key**
1	1
2	1
3	1

Tabelle 4.6.: Tabelle *Zugehoerigkeit*

Student_Teilnahme_Key	Veranstaltung_Teilname_Key
1	1

Tabelle 4.7.: Tabelle *Teilnahme*

Ruft man `testData` erneut auf, so wird ein `UniqueError` ausgegeben, weil die Matrikelnummer eines Studenten eindeutig ist und deswegen schon das Anlegen des ersten Studenten fehlschlägt. Nun wird der `Student` mit dem Vornamen „Marion“ gesucht und ausgegeben.

```
t1 = do
  s <- getDynamicSolutions (\stud ->
         let key free
         in
         student key stud |> studentVorname stud == "Marion")
  print s
```

Das Ergebnis ist `[(Student 1 480811 "Mueller" "Marion" Nothing)]`. Für die optionale Email-Adresse ist noch nichts eingetragen, was man durch ein Update ändern kann. Der Nullwert wird durch eine Email-Adresse ersetzt.

```
update = do
  x <- getDynamicSolutions (\k ->
         let stud free
         in
         student k stud |> studentVorname stud == "Marion")
  m <- getStudent (head x)
  updateStudent
    (setStudentEmail (result m)
                     (Just (Email "mam@informatik.uni-kiel.de")))
```

Das Ergebnis einer erneuten Ausführung von `t1` ist wie erwartet

```
[(Student 1 480811 "Mueller" "Marion"
          (Just (Email "mam@informatik.uni-kiel.de")))].
```

Alle Studenten, die an einer Veranstaltung teilnehmen, werden als Paare der Schlüssel in einer Liste ausgegeben.

```
t2 = do
  s <- getDynamicSolutions (\(sk,vk) -> nimmt_teil sk vk)
  print s
```

Das Ergebnis von `t2` ist `[((StudentKey 1),(VeranstaltungKey 1))]`. Einfache Anfragen sind also schon durch die dynamischen Prädikate für Rollenbezeichner implementiert. Daraus lassen sich beliebige Anfragen kombinieren, wie beispielsweise das Prädikat

```
schueler :: Dozent -> Student -> Dynamic
schueler d s =
    dozent dk d
    <> student sk s
    <> nimmt_teil sk vk
    <> haelt dk vk
  where vk,dk,sk free
```

mit dem Schüler ermittelt werden können. Dozent und Schüler sind miteinander verbunden, indem ein Schüler an einer Veranstaltung teilnimmt, die von dem Dozenten gehalten wird. Mit der Testfunktion

```
t3 = do
  s <- getDynamicSolutions (\(doz,stu) -> schueler doz stu)
  print s
```

werden alle in den Daten enthaltenen Schüler-Kombinationen von Dozenten und Studenten gesucht. Mit `t3` wird eine Kombination gefunden:

```
[((Dozent 1 1 "Hanus" "Michael"),
  (Student 1 480811 "Mueller" "Marion"
           (Just (Email "mam@informatik.uni-kiel.de"))))].
```

Nun nimmt noch ein weiterer Student an der Veranstaltung teil. Dafür wird die Funktion `addTeilnahme` ausgeführt. Jede weitere Ausführung dieser Funktion liefert einen `UniqueError`, weil die Beziehung schon existiert.

```
addTeilnahme = do
  s <- getDynamicSolutions (\sk ->
         let stud free
         in
         student sk stud
           |> studentMatrikelNr stud == 111111)
  v <- getDynamicSolutions (\vk ->
         let ver free
         in
         veranstaltung vk ver
           |> veranstaltungTitel ver == "Logikprogrammierung")
  newTeilnahme (head s) (head v)
```

Dann liefert `t2`:

```
[((StudentKey 2),(VeranstaltungKey 1)),
 ((StudentKey 1),(VeranstaltungKey 1))]
```

und `t3`:

```
[((Dozent 1 1 "Hanus" "Michael"),
  (Student 1 480811 "Mueller" "Marion"
           (Just (Email "mam@informatik.uni-kiel.de")))),
 ((Dozent 1 1 "Hanus" "Michael"),
  (Student 2 111111 "Name1" "Vorname1" Nothing))]
```

5. Zusammenfassung

Die in dieser Arbeit implementierte Codegenerierung aus ER-Diagrammen für die Benutzung einer Datenbank mit funktional-logischer Programmierung in Curry ist eine wesentliche Arbeitserleichterung für zukünftige Projekte. Nach dem grafischen Entwurf der Datenbank ist es nicht mehr notwendig, die fehleranfällige Programmierung der Datenstrukturen für die Einträge in Tabellen mit ihren getter- und setter-Funktionen und der Datenbank-Operationen, in denen oft viele Überprüfungen sinnvoll sind, von Hand zu machen. Auch Beziehungen zwischen den Tabellen werden durch dynamische Prädikate sichtbar. Stattdessen kann man gleich mit den Erweiterungen beginnen.

Das Werkzeug zur Erstellung des ER-Diagramms als Basis der Codegenerierung ist leicht austauschbar, nur das Modul `XML2ERD` muss dafür neu implementiert werden. Während der Implementierung habe ich die Version von Umbrello gewechselt, was eine Veränderung des Speicherformates bedeutete. Die Anpassung war schnell möglich, da nur die Struktur und nicht die einzelnen XML-Elemente unterschiedlich war, sodass nur wenige Funktionen geändert werden mussten.

Die im ER-Diagramm vergebenen Namen werden für die generierten Datentypen und Funktionen übernommen. Dadurch ist der generierte Code leicht verständlich und es lassen sich lesbare Anfragen aus den Prädikaten kombinieren.

Die Trans-Monade ermöglicht einzelne Testfunktionen, die Fehlermeldungen zur Laufzeit ausgeben. Sollen Änderungen in der Datenbank gemacht werden, die nicht den Vorgaben des ER-Diagramms entsprechen, wird ein Fehler ausgegeben und nichts geändert. Auch ist die Trans-Monade so implementiert, dass Transaktionen nachträglich ergänzt werden können, ohne an der Codegenerierung etwas zu ändern.

Durch die Option, Funktionen für Konsistenztests zu generieren, erhält der Benutzer eine zusätzliche Möglichkeit, die Konsistenz der gespeicherten Daten jederzeit zu kontrollieren.

Außer der allgemein gehaltenen Datenstruktur `ERD` für ER-Diagramme wird das ER-Modell in dieser Implementierung auf binäre Beziehungen eingeschränkt. Sollen beliebige Beziehungen verwendet werden können, ist nur ein zusätzlicher Transformationsschritt der höherwertigen Beziehungen in binäre Beziehungen vor der in Abschnitt 4.3 beschriebenen Transformation notwendig.

5.1. Zukünftige Arbeiten

Mit CurryDoc [4] und speziellen Kommentaren kann man die Dokumentation eines Programms automatisch erzeugen. Hauptsächlich für die new-Funktionen mit den oft vielen Parametern würde eine Dokumentation die Benutzung erheblich unterstützen. Man könnte leicht die Reihenfolge der erforderlichen Parameter nachsehen. Denkbar wäre auch, Kommentare aus dem ER-Diagramm als Kommentare im Programm zu verwenden. Kommentare werden von `AbstractCurry` nicht unterstützt, dafür wäre also noch eine Erweiterung notwendig.

Noch mehr Unabhängigkeit vom verwendeten Werkzeug würde eine Trennung der Konvertierung von der Überprüfung der Vorgaben bringen. Das könnte man als zukünftige Aufgabe ändern, bringt aber nur Vorteile, wenn das Werkzeug tatsächlich ausgetauscht werden soll.

In dieser Implementierung werden künstliche Schlüssel eingefügt. Da der Wert eines Schlüssels in einer Datenstruktur versteckt wird, ist eine Erweiterung auf benutzerdefinierte Schlüssel möglich. Beispielsweise könnte man dies als zusätzlich Option anbieten.

Bei Codegenerierung mit der Option `DB` für die Verwendung einer Datenbank muss für jeden benutzerdefinierten Datentyp T in demselben Modul wie der Datentyp eine `DBSpec`-Funktion t defniert sein. Schöner wäre es, wenn bei der Eingabe eines benutzerdefinierten Datentyps im Werkzeug die entsprechende Funktion mit angegeben werden könnte.

A. Transformation

Das ER-Diagramm und das Ergebnis der Transformation in der Datenstruktur `ERD` für das Beispiel `Uni.xmi` kann man sich mit der Funktion `testTransformation` aus dem Modul `Main` ausgeben lassen und erhält:

```
(ERD "Uni"
  [(Entity "Student"
     [(Attribute "MatrikelNr" (IntDom Nothing) SimpleKey False),
      (Attribute "Name" (StringDom Nothing) NoKey False),
      (Attribute "Vorname" (StringDom Nothing) NoKey False),
      (Attribute "Email" (UserDefined "MyModule.Email" Nothing) NoKey True)]),
   (Entity "Veranstaltung"
     [(Attribute "Nr" (IntDom Nothing) SimpleKey False),
      (Attribute "Titel" (StringDom Nothing) Unique False),
      (Attribute "SWS" (IntDom (Just 4)) NoKey False)]),
   (Entity "Dozent"
     [(Attribute "Nr" (IntDom Nothing) SimpleKey False),
      (Attribute "Name" (StringDom Nothing) NoKey False),
      (Attribute "Vorname" (StringDom Nothing) NoKey False)]),
   (Entity "Gruppe"
     [(Attribute "Termin" (StringDom Nothing) NoKey False)])]
  [(Relationship "Veranstalten"
     [(REnd "Dozent" "wird_gehalten" (Exactly 1)),
      (REnd "Veranstaltung" "haelt" (Range 0 Nothing))]),
   (Relationship "Teilnahme"
     [(REnd "Student" "wird_besucht" (Range 0 Nothing)),
      (REnd "Veranstaltung" "nimmt_teil" (Range 0 Nothing))]),
   (Relationship "Zugehoerigkeit"
     [(REnd "Student" "besteht_aus" (Exactly 3)),
      (REnd "Gruppe" "ist_in" (Range 0 Nothing))])])

(ERD "Uni"
  [(Entity "Zugehoerigkeit"
     [(Attribute "Student_Zugehoerigkeit_Key" (KeyDom "Student")
                 NoKey False),
      (Attribute "Gruppe_Zugehoerigkeit_Key" (KeyDom "Gruppe")
                 NoKey False)]),
```

```
  (Entity "Teilnahme"
    [(Attribute "Student_Teilnahme_Key" (KeyDom "Student")
                NoKey False),
     (Attribute "Veranstaltung_Teilnahme_Key"
                (KeyDom "Veranstaltung") NoKey False)]),
  (Entity "Student"
    [(Attribute "Key" (IntDom Nothing) SimpleKey False),
     (Attribute "MatrikelNr" (IntDom Nothing) Unique False),
     (Attribute "Name" (StringDom Nothing) NoKey False),
     (Attribute "Vorname" (StringDom Nothing) NoKey False),
     (Attribute "Email" (UserDefined "MyModule.Email" Nothing)
                NoKey True)]),
  (Entity "Veranstaltung"
    [(Attribute "Key" (IntDom Nothing) SimpleKey False),
     (Attribute "Dozent_Veranstalten_Key" (KeyDom "Dozent")
                NoKey False),
     (Attribute "Nr" (IntDom Nothing) Unique False),
     (Attribute "Titel" (StringDom Nothing) Unique False),
     (Attribute "SWS" (IntDom (Just 4)) NoKey False)]),
  (Entity "Dozent"
    [(Attribute "Key" (IntDom Nothing) SimpleKey False),
     (Attribute "Nr" (IntDom Nothing) Unique False),
     (Attribute "Name" (StringDom Nothing) NoKey False),
     (Attribute "Vorname" (StringDom Nothing) NoKey False)]),
  (Entity "Gruppe"
    [(Attribute "Key" (IntDom Nothing) SimpleKey False),
     (Attribute "Termin" (StringDom Nothing) NoKey False)])]
 [(Relationship []
    [(REnd "Student" [] (Exactly 1)),
     (REnd "Zugehoerigkeit" "ist_in" (Range 0 Nothing))]),
  (Relationship []
    [(REnd "Gruppe" [] (Exactly 1)),
     (REnd "Zugehoerigkeit" "besteht_aus" (Exactly 3))]),
  (Relationship []
    [(REnd "Student" [] (Exactly 1)),
     (REnd "Teilnahme" "nimmt_teil" (Range 0 Nothing))]),
  (Relationship []
    [(REnd "Veranstaltung" [] (Exactly 1)),
     (REnd "Teilnahme" "wird_besucht" (Range 0 Nothing))]),
  (Relationship "Veranstalten"
    [(REnd "Dozent" "wird_gehalten" (Exactly 1)),
     (REnd "Veranstaltung" "haelt" (Range 0 Nothing))])])
```

B. Generierter Code

```
module Uni(Student, Veranstaltung, Dozent, Gruppe, StudentKey,
  VeranstaltungKey, DozentKey, GruppeKey, studentMatrikelNr,
  setStudentMatrikelNr, studentName, setStudentName, studentVorname,
  setStudentVorname, studentEmail, setStudentEmail,
  veranstaltungDozent_Veranstalten_Key,
  setVeranstaltungDozent_Veranstalten_Key, veranstaltungNr,
  setVeranstaltungNr, veranstaltungTitel, setVeranstaltungTitel,
  veranstaltungSWS, setVeranstaltungSWS, dozentNr, setDozentNr,
  dozentName, setDozentName, dozentVorname, setDozentVorname,
  gruppeTermin, setGruppeTermin, student, studentKey, newStudent,
  updateStudent, getStudent, veranstaltung, veranstaltungKey,
  newVeranstaltung, updateVeranstaltung, getVeranstaltung, dozent,
  dozentKey, newDozent, updateDozent, getDozent, gruppe, gruppeKey,
  newGruppe, updateGruppe, getGruppe, zugehoerigkeit, newZugehoerigkeit,
  teilnahme, newTeilnahme, ist_in, besteht_aus, nimmt_teil, wird_besucht,
  haelt, veranstalten, wird_gehalten, checkAllData, checkZugehoerigkeit,
  checkTeilnahme, checkStudent, checkVeranstaltung, checkDozent,
  checkGruppe) where

import Dynamic
import Intern
import Trans
import MyModule

data Student = Student Key Int String String (Maybe MyModule.Email)

data Veranstaltung = Veranstaltung Key Key Int String Int

data Dozent = Dozent Key Int String String

data Gruppe = Gruppe Key String

data StudentKey = StudentKey Key

data VeranstaltungKey = VeranstaltungKey Key
```

```
data DozentKey = DozentKey Key

data GruppeKey = GruppeKey Key

data Zugehoerigkeit = Zugehoerigkeit Key Key

data Teilnahme = Teilnahme Key Key

setZugehoerigkeitStudent_Zugehoerigkeit_Key :: Zugehoerigkeit -> Key
  -> Zugehoerigkeit
setZugehoerigkeitStudent_Zugehoerigkeit_Key (Zugehoerigkeit _ x2) x =
  Zugehoerigkeit x x2

setZugehoerigkeitGruppe_Zugehoerigkeit_Key :: Zugehoerigkeit -> Key
  -> Zugehoerigkeit
setZugehoerigkeitGruppe_Zugehoerigkeit_Key (Zugehoerigkeit x1 _) x =
  Zugehoerigkeit x1 x

setTeilnahmeStudent_Teilnahme_Key :: Teilnahme -> Key -> Teilnahme
setTeilnahmeStudent_Teilnahme_Key (Teilnahme _ x2) x = Teilnahme x x2

setTeilnahmeVeranstaltung_Teilnahme_Key :: Teilnahme -> Key -> Teilnahme
setTeilnahmeVeranstaltung_Teilnahme_Key (Teilnahme x1 _) x = Teilnahme x1 x

setStudentKey :: Student -> Key -> Student
setStudentKey (Student _ x2 x3 x4 x5) x = Student x x2 x3 x4 x5

studentMatrikelNr :: Student -> Int
studentMatrikelNr (Student _ x _ _ _) = x

setStudentMatrikelNr :: Student -> Int -> Student
setStudentMatrikelNr (Student x1 _ x3 x4 x5) x = Student x1 x x3 x4 x5

studentName :: Student -> String
studentName (Student _ _ x _ _) = x

setStudentName :: Student -> String -> Student
setStudentName (Student x1 x2 _ x4 x5) x = Student x1 x2 x x4 x5

studentVorname :: Student -> String
studentVorname (Student _ _ _ x _) = x

setStudentVorname :: Student -> String -> Student
setStudentVorname (Student x1 x2 x3 _ x5) x = Student x1 x2 x3 x x5
```

```
studentEmail :: Student -> Maybe MyModule.Email
studentEmail (Student _ _ _ _ x) = x

setStudentEmail :: Student -> Maybe MyModule.Email -> Student
setStudentEmail (Student x1 x2 x3 x4 _) x = Student x1 x2 x3 x4 x

setVeranstaltungKey :: Veranstaltung -> Key -> Veranstaltung
setVeranstaltungKey (Veranstaltung _ x2 x3 x4 x5) x =
  Veranstaltung x x2 x3 x4 x5

veranstaltungDozent_Veranstalten_Key :: Veranstaltung -> Key
veranstaltungDozent_Veranstalten_Key (Veranstaltung _ x _ _ _) = x

setVeranstaltungDozent_Veranstalten_Key :: Veranstaltung -> Key
  -> Veranstaltung
setVeranstaltungDozent_Veranstalten_Key (Veranstaltung x1 _ x3 x4 x5) x =
  Veranstaltung x1 x x3 x4 x5

veranstaltungNr :: Veranstaltung -> Int
veranstaltungNr (Veranstaltung _ _ x _ _) = x

setVeranstaltungNr :: Veranstaltung -> Int -> Veranstaltung
setVeranstaltungNr (Veranstaltung x1 x2 _ x4 x5) x =
  Veranstaltung x1 x2 x x4 x5

veranstaltungTitel :: Veranstaltung -> String
veranstaltungTitel (Veranstaltung _ _ _ x _) = x

setVeranstaltungTitel :: Veranstaltung -> String -> Veranstaltung
setVeranstaltungTitel (Veranstaltung x1 x2 x3 _ x5) x =
  Veranstaltung x1 x2 x3 x x5

veranstaltungSWS :: Veranstaltung -> Int
veranstaltungSWS (Veranstaltung _ _ _ _ x) = x

setVeranstaltungSWS :: Veranstaltung -> Int -> Veranstaltung
setVeranstaltungSWS (Veranstaltung x1 x2 x3 x4 _) x =
  Veranstaltung x1 x2 x3 x4 x

setDozentKey :: Dozent -> Key -> Dozent
setDozentKey (Dozent _ x2 x3 x4) x = Dozent x x2 x3 x4

dozentNr :: Dozent -> Int
```

```
dozentNr (Dozent _ x _ _) = x

setDozentNr :: Dozent -> Int -> Dozent
setDozentNr (Dozent x1 _ x3 x4) x = Dozent x1 x x3 x4

dozentName :: Dozent -> String
dozentName (Dozent _ _ x _) = x

setDozentName :: Dozent -> String -> Dozent
setDozentName (Dozent x1 x2 _ x4) x = Dozent x1 x2 x x4

dozentVorname :: Dozent -> String
dozentVorname (Dozent _ _ _ x) = x

setDozentVorname :: Dozent -> String -> Dozent
setDozentVorname (Dozent x1 x2 x3 _) x = Dozent x1 x2 x3 x

setGruppeKey :: Gruppe -> Key -> Gruppe
setGruppeKey (Gruppe _ x2) x = Gruppe x x2

gruppeTermin :: Gruppe -> String
gruppeTermin (Gruppe _ x) = x

setGruppeTermin :: Gruppe -> String -> Gruppe
setGruppeTermin (Gruppe x1 _) x = Gruppe x1 x

student :: StudentKey -> Student -> Dynamic
student key obj
  | (=:=) (studentKeyToKey key) (studentKeyToKey (studentKey obj))
  = studentEntry obj

studentEntry :: Student -> Dynamic
studentEntry = persistent "file:./StudentDB"

studentKey :: Student -> StudentKey
studentKey (Student x _ _ _ _) = StudentKey x

studentKeyToKey :: StudentKey -> Key
studentKeyToKey (StudentKey k) = k

newStudent :: Int -> String -> String -> Maybe MyModule.Email
           -> Trans Student
newStudent matrikelNr_p name_p vorname_p email_p =
  (>>-) (unique matrikelNr_p studentMatrikelNr studentEntry)
```

```
        (newEntry ((.) studentKeyToKey studentKey)
                  setStudentKey studentEntry
                  (Student 0 matrikelNr_p name_p vorname_p email_p))

updateStudent :: Student -> Trans ()
updateStudent student_p =
  (>>-) (uniqueUpdate student_p studentKey studentMatrikelNr student)
        (updateEntry studentKey student student_p)

getStudent :: StudentKey -> Trans Student
getStudent key = getEntry key student

veranstaltung :: VeranstaltungKey -> Veranstaltung -> Dynamic
veranstaltung key obj
  | veranstaltungKeyToKey key =:= veranstaltungKeyToKey (veranstaltungKey obj)
  = veranstaltungEntry obj

veranstaltungEntry :: Veranstaltung -> Dynamic
veranstaltungEntry = persistent "file:./VeranstaltungDB"

veranstaltungKey :: Veranstaltung -> VeranstaltungKey
veranstaltungKey (Veranstaltung x _ _ _ _) = VeranstaltungKey x

veranstaltungKeyToKey :: VeranstaltungKey -> Key
veranstaltungKeyToKey (VeranstaltungKey k) = k

newVeranstaltung :: DozentKey -> Int -> String -> Maybe Int
  -> Trans Veranstaltung
newVeranstaltung dozent_Veranstalten_Key_p nr_p titel_p sWS_p =
  unique nr_p veranstaltungNr veranstaltungEntry
    >>- unique titel_p veranstaltungTitel veranstaltungEntry
    >>- existsDBKey dozent_Veranstalten_Key_p dozentKey dozentEntry)
    >>- newEntry (veranstaltungKeyToKey . veranstaltungKey)
                 setVeranstaltungKey veranstaltungEntry
                 (Veranstaltung 0 (dozentKeyToKey dozent_Veranstalten_Key_p)
                                  nr_p titel_p
                                  (case sWS_p of (Just v) -> v
                                                 Nothing -> 4))

updateVeranstaltung :: Veranstaltung -> Trans ()
updateVeranstaltung veranstaltung_p =
  uniqueUpdate veranstaltung_p veranstaltungKey veranstaltungNr veranstaltung
    >>- uniqueUpdate veranstaltung_p veranstaltungKey veranstaltungTitel
                     veranstaltung
```

```
    >>- existsDBKey
          (DozentKey (veranstaltungDozent_Veranstalten_Key veranstaltung_p))
          dozentKey dozentEntry
    >>- updateEntry veranstaltungKey veranstaltung veranstaltung_p

getVeranstaltung :: VeranstaltungKey -> Trans Veranstaltung
getVeranstaltung key = getEntry key veranstaltung

dozent :: DozentKey -> Dozent -> Dynamic
dozent key obj
  | (=:=) (dozentKeyToKey key) (dozentKeyToKey (dozentKey obj))
  = dozentEntry obj

dozentEntry :: Dozent -> Dynamic
dozentEntry = persistent "file:./DozentDB"

dozentKey :: Dozent -> DozentKey
dozentKey (Dozent x _ _ _) = DozentKey x

dozentKeyToKey :: DozentKey -> Key
dozentKeyToKey (DozentKey k) = k

newDozent :: Int -> String -> String -> Trans Dozent
newDozent nr_p name_p vorname_p = unique nr_p dozentNr dozentEntry
  >>- newEntry (dozentKeyToKey . dozentKey) setDozentKey dozentEntry
               (Dozent 0 nr_p name_p vorname_p)

updateDozent :: Dozent -> Trans ()
updateDozent dozent_p = uniqueUpdate dozent_p dozentKey dozentNr dozent
  >>- updateEntry dozentKey dozent dozent_p

getDozent :: DozentKey -> Trans Dozent
getDozent key = getEntry key dozent

gruppe :: GruppeKey -> Gruppe -> Dynamic
gruppe key obj
  | (=:=) (gruppeKeyToKey key) (gruppeKeyToKey (gruppeKey obj))
  = gruppeEntry obj

gruppeEntry :: Gruppe -> Dynamic
gruppeEntry = persistent "file:./GruppeDB"

gruppeKey :: Gruppe -> GruppeKey
gruppeKey (Gruppe x _) = GruppeKey x
```

```
gruppeKeyToKey :: GruppeKey -> Key
gruppeKeyToKey (GruppeKey k) = k

newGruppe :: String -> StudentKey -> StudentKey -> StudentKey
          -> Trans Gruppe
newGruppe termin_p k11 k12 k13 =
  duplicatePTest (k11 : k12 : k13 : [])
    >>- existsDBKey k11 studentKey studentEntry
    >>- existsDBKey k12 studentKey studentEntry
    >>- existsDBKey k13 studentKey studentEntry
    >>- newEntry (gruppeKeyToKey . gruppeKey) setGruppeKey gruppeEntry
                 (Gruppe 0 termin_p)
    >>~ \entry ->
          newEntryR k11 (gruppeKey entry) zugehoerigkeit
            >>- newEntryR k12 (gruppeKey entry) zugehoerigkeit
            >>- newEntryR k13 (gruppeKey entry) zugehoerigkeit
            >>- returnTrans entry

updateGruppe :: Gruppe -> Trans ()
updateGruppe gruppe_p = updateEntry gruppeKey gruppe gruppe_p

getGruppe :: GruppeKey -> Trans Gruppe
getGruppe key = getEntry key gruppe

zugehoerigkeitStudent_Zugehoerigkeit_Key :: Zugehoerigkeit -> StudentKey
zugehoerigkeitStudent_Zugehoerigkeit_Key (Zugehoerigkeit x _) =
  StudentKey x

zugehoerigkeitGruppe_Zugehoerigkeit_Key :: Zugehoerigkeit -> GruppeKey
zugehoerigkeitGruppe_Zugehoerigkeit_Key (Zugehoerigkeit _ x) = GruppeKey x

zugehoerigkeit :: StudentKey -> GruppeKey -> Dynamic
zugehoerigkeit (StudentKey key1) (GruppeKey key2) =
  zugehoerigkeitEntry (Zugehoerigkeit key1 key2)

newZugehoerigkeit :: StudentKey -> GruppeKey -> Trans ()
newZugehoerigkeit key1 key2 = existsDBKey key1 studentKey studentEntry
  >>- existsDBKey key2 gruppeKey gruppeEntry
  >>- unique2
       (studentKeyToKey key1) (gruppeKeyToKey key2)
       (studentKeyToKey . zugehoerigkeitStudent_Zugehoerigkeit_Key)
       (gruppeKeyToKey . zugehoerigkeitGruppe_Zugehoerigkeit_Key)
       zugehoerigkeitEntry
```

```
  >>- newEntryR key1 key2 zugehoerigkeit

zugehoerigkeitEntry :: Zugehoerigkeit -> Dynamic
zugehoerigkeitEntry = persistent "file:./ZugehoerigkeitDB"

teilnahmeStudent_Teilnahme_Key :: Teilnahme -> StudentKey
teilnahmeStudent_Teilnahme_Key (Teilnahme x _) = StudentKey x

teilnahmeVeranstaltung_Teilnahme_Key :: Teilnahme -> VeranstaltungKey
teilnahmeVeranstaltung_Teilnahme_Key (Teilnahme _ x) = VeranstaltungKey x

teilnahme :: StudentKey -> VeranstaltungKey -> Dynamic
teilnahme (StudentKey key1) (VeranstaltungKey key2) =
  teilnahmeEntry (Teilnahme key1 key2)

newTeilnahme :: StudentKey -> VeranstaltungKey -> Trans ()
newTeilnahme key1 key2 = existsDBKey key1 studentKey studentEntry
  >>- existsDBKey key2 veranstaltungKey veranstaltungEntry
  >>- unique2
       (studentKeyToKey key1) (veranstaltungKeyToKey key2)
       (studentKeyToKey . teilnahmeStudent_Teilnahme_Key)
       (veranstaltungKeyToKey . teilnahmeVeranstaltung_Teilnahme_Key)
       teilnahmeEntry
  >>- newEntryR key1 key2 teilnahme

teilnahmeEntry :: Teilnahme -> Dynamic
teilnahmeEntry = persistent "file:./TeilnahmeDB"

ist_in :: StudentKey -> GruppeKey -> Dynamic
ist_in = zugehoerigkeit

besteht_aus :: GruppeKey -> StudentKey -> StudentKey -> StudentKey
            -> Dynamic
besteht_aus key key1 key2 key3 = zugehoerigkeit key1 key
  <> zugehoerigkeit key2 key
  <> zugehoerigkeit key3 key
  |> (key1 /= key2 && key1 /= key3 && key2 /= key3)

nimmt_teil :: StudentKey -> VeranstaltungKey -> Dynamic
nimmt_teil = teilnahme

wird_besucht :: VeranstaltungKey -> StudentKey -> Dynamic
wird_besucht = flip teilnahme
```

```
haelt :: DozentKey -> VeranstaltungKey -> Dynamic
haelt key1 key2 = veranstaltung key2 f2
  |&> dozentKeyToKey key1 =:= veranstaltungDozent_Veranstalten_Key f2
  where f2 free

veranstalten :: DozentKey -> VeranstaltungKey -> Dynamic
veranstalten = haelt

wird_gehalten :: VeranstaltungKey -> DozentKey -> Dynamic
wird_gehalten = flip haelt

checkAllData :: Trans ()
checkAllData = checkZugehoerigkeit
  >>- checkTeilnahme
  >>- checkStudent
  >>- checkVeranstaltung
  >>- checkDozent checkGruppe

checkZugehoerigkeit :: Trans ()
checkZugehoerigkeit = do
  entries <- getDynamicSolutions zugehoerigkeitEntry
  mapTrans_ checkZugehoerigkeitEntry entries

checkTeilnahme :: Trans ()
checkTeilnahme = do
  entries <- getDynamicSolutions teilnahmeEntry
  mapTrans_ checkTeilnahmeEntry entries

checkStudent :: Trans ()
checkStudent = do
  entries <- getDynamicSolutions studentEntry
  mapTrans_ checkStudentEntry entries

checkVeranstaltung :: Trans ()
checkVeranstaltung = do
  entries <- getDynamicSolutions veranstaltungEntry
  mapTrans_ checkVeranstaltungEntry entries

checkDozent :: Trans ()
checkDozent = do
  entries <- getDynamicSolutions dozentEntry
  mapTrans_ checkDozentEntry entries

checkGruppe :: Trans ()
```

```
checkGruppe = do
  entries <- getDynamicSolutions gruppeEntry
  mapTrans_ checkGruppeEntry entries

checkZugehoerigkeitEntry :: Zugehoerigkeit -> Trans ()
checkZugehoerigkeitEntry zugehoerigkeit_p =
  existsDBKey (zugehoerigkeitStudent_Zugehoerigkeit_Key zugehoerigkeit_p)
              studentKey studentEntry
    >>- existsDBKey
         (zugehoerigkeitGruppe_Zugehoerigkeit_Key zugehoerigkeit_p)
         gruppeKey gruppeEntry
    >>- unique2C
         (studentKeyToKey
           (zugehoerigkeitStudent_Zugehoerigkeit_Key zugehoerigkeit_p))
         (gruppeKeyToKey
           (zugehoerigkeitGruppe_Zugehoerigkeit_Key zugehoerigkeit_p))
         (studentKeyToKey . zugehoerigkeitStudent_Zugehoerigkeit_Key)
         (gruppeKeyToKey . zugehoerigkeitGruppe_Zugehoerigkeit_Key)
         zugehoerigkeitEntry

checkTeilnahmeEntry :: Teilnahme -> Trans ()
checkTeilnahmeEntry teilnahme_p =
  existsDBKey (teilnahmeStudent_Teilnahme_Key teilnahme_p)
              studentKey studentEntry
    >>- existsDBKey (teilnahmeVeranstaltung_Teilnahme_Key teilnahme_p)
                    veranstaltungKey veranstaltungEntry
    >>- unique2C
         (studentKeyToKey (teilnahmeStudent_Teilnahme_Key teilnahme_p))
         (veranstaltungKeyToKey
           (teilnahmeVeranstaltung_Teilnahme_Key teilnahme_p))
         (studentKeyToKey . teilnahmeStudent_Teilnahme_Key)
         (veranstaltungKeyToKey . teilnahmeVeranstaltung_Teilnahme_Key)
         teilnahmeEntry

checkStudentEntry :: Student -> Trans ()
checkStudentEntry student_p = duplicateKeyTest student
  >>- uniqueC student_p studentMatrikelNr student

checkVeranstaltungEntry :: Veranstaltung -> Trans ()
checkVeranstaltungEntry veranstaltung_p =
  duplicateKeyTest veranstaltung)
    >>- uniqueC veranstaltung_p veranstaltungNr veranstaltung
    >>- uniqueC veranstaltung_p veranstaltungTitel veranstaltung
    >>- existsDBKey
```

```
          (DozentKey
            (veranstaltungDozent_Veranstalten_Key veranstaltung_p))
          dozentKey dozentEntry

checkDozentEntry :: Dozent -> Trans ()
checkDozentEntry dozent_p = duplicateKeyTest dozent
  >>- uniqueC dozent_p dozentNr dozent

checkGruppeEntry :: Gruppe -> Trans ()
checkGruppeEntry gruppe_p = duplicateKeyTest gruppe
  >>- maxTestC 3 (gruppeKeyToKey (gruppeKey gruppe_p))
                 (gruppeKeyToKey . zugehoerigkeitGruppe_Zugehoerigkeit_Key)
                 zugehoerigkeitEntry
  >>- minTestC 3 (gruppeKeyToKey (gruppeKey gruppe_p))
                 (gruppeKeyToKey . zugehoerigkeitGruppe_Zugehoerigkeit_Key)
                 zugehoerigkeitEntry
```

C. Inhalt der CD

Datei	**Beschreibung**
DiplomarbeitMarionMueller.pdf	Diplomarbeit
Uni.xmi	mit Umbrello erstellte Datei, Beispiel zum Testen
Main.curry	enthält Aufruf der Codegenerierung aus einer XMI-Datei und eine Testfunktion für die Transformation
ERD.curry	Datenstruktur `ERD` für ER-Diagramme
XML2ERD.curry	Konvertierung des ER-Diagramms aus einer Umbrello-Ausgabe in die Curry-Datenstruktur `ERD`
Transformation.curry	Transformation der Datenstruktur `ERD`
Trans.curry	Trans-Monade
Intern.curry	allgemeine Funktionen, die im generierten Code aufgerufen werden
CodeGeneration.curry	Codegenerierung
Uni.curry	aus Uni.xmi generierter Code (siehe auch Anhang B)
MyModule.curry	enthält benutzerdefinierten Datentyp und wird im generierten Code importiert
Test.curry	Anwendungsbeispiel für den generierten Code

Literaturverzeichnis

[1] Peter Pin-Chan Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[2] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2), 2006. Available at `http://www.informatik.uni-kiel.de/~curry`.

[3] S. Fischer. Functional logic programming with databases. Diploma Thesis, Kiel University, 2005.
Available at `http://www.informatik.uni-kiel.de/~mh/lehre/diplom.html`.

[4] M. Hanus. Currydoc: A documentation tool for declarative programs. In *Proc. of the 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, pages 225–228, Grado (Italy), 2002. Research Report UDMI/18/2002/RR, Università degli Studi di Udine.

[5] M. Hanus, S. Antoy, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pakcs: The portland aachen kiel curry system, 2000.
Available at `http://www.informatik.uni-kiel.de/~pakcs`.

[6] Michael Hanus. Dynamic predicates in functional logic programs. *Journal of Functional and Logic Programming*, (5), 2004.

[7] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[8] Andreas Meier. *Relationale Datenbanken. Leitfaden für die Praxis.* Springer, 2004.

[9] Edwin Schicker. *Datenbanken und SQL.* Teubner, 2000.