Seminarausarbeitung:

HaskellDB

Betreuer: Sebastian Fischer

Literatur:

Daan Lejien, Erik Meijer, Domain Specific Embedded Compilers

Ken Bell

11. Januar 2006

Zusammenfassung

Mit HaskellDB(improved) wurde eine Bibliothek von Methoden zum Datenbankzugriff in Haskell geschaffen. Dabei ist HaskellDB nicht einfach eine Bibliothek, die in SQL formulierte Datenbankanfragen verarbeitet, sondern besteht aus einem Framework in welchem die Vorzüge von Haskell wie Typsicherheit bei der Formulierung von Anfragen an die Datenbank erhalten bleiben. Damit werden die klassischen Probleme in der Entwicklung von Datenbankgestützten Applikationen wie syntaktisch und semantisch inkorrekte SQL-Anfragen beseitigt. Die Anfragen werden nicht, wie normalerweise üblich, in SQL formuliert, sondern direkt in Haskellsyntax basierend auf relationaler Algebra, so daß sich der Entwickler während der Softwareentwicklung immer nur in einer Sprache befindet.

1 Einleitung

Datenbanken sind aus der heutigen Welt der Softwareentwicklung kaum noch wegzudenken. In den meisten modernen Hochsprachen ist man bei der Entwicklung von Datenbankapplikationen allerdings darauf angewiesen neben der eigentlichen Programmiersprache zur Entwicklung der Anwendung auch noch die Structured Query Language (SQL) zu beherrschen. Da die in SQL geschriebenen Anfragen an die Datenbank in den meisten Fällen als Strings im Quellcode vorliegen und der Compiler diese daher nicht auf Syntax- und Typkorrektheit überprüfen kann, liegt hier eine große Fehlerquelle in der Anwendungsentwicklung, denn diese Fehler treten im Regelfall erst zur Laufzeit und im schlimmsten Fall im produktiven Einsatz auf.
Um diesen Problemen aus dem Weg zu gehen, wurde bei der Integration von Datenbankverbindungen in Haskell ein anderer Weg beschritten. Hier wird ein Framework zugrunde gelegt, mit welchem die Typsicherheit und die Syntaxkorrektheit von Datenbankabfragen bereits durch den Compiler garantiert wird. Des Weiteren ist hervorzuheben, daß der Softwareentwickler sich nicht mit SQL beschäftigen muss, da Datenbankanfragen auf Basis von relationaler Algebra in Haskell formuliert werden.
In dieser Ausarbeitung werde ich anhand einer Beispieldatenbank kurz auf die Grundlagen von relationalen Datenbanken eingehen, aufzeigen wie SQL in Haskell eingebettet wird und die Besonderheiten des HaskellDB Frameworks erläutern.

2 Grundlagen

2.1 Relationale Datenbanken

In relationalen Datenbanken werden Daten in Form von zweidimensionalen Tabellen (oder auch Relationen) gespeichert. Eine Zeile (ein Tupel) einer solchen Tabelle entspricht einem Datensatz und eine Spalte der Tabelle ist ein Attribut.
Anfragen an eine relationale Datenbank erfolgen in den meisten Fällen durch SQL. Dabei folgen die Anfragen strikt den mathematischen Gesetzmäßigkeiten der Relationalen Algebra.
In der relationalen Algebra wird das Auswählen von Datensätzen (Tupel) aus einer Tabelle mit dem Selektionsoperator σ formuliert. Zur gezielten Auswahl von bestimmten Attributen einer Tupelmenge wird der Projektionsoperator π verwendet.
Betrachten wir also nun einen Auschschnitt einer Personentabelle mit dem Namen *Personen* Die Frage nach dem Geburtsdatum von John Doe lautet in relationaler Algebra

$$\pi_{Gebdatum}(\sigma_{Vorname=John,Name=Doe}Personen).$$

ID	*Name*	*Vorname*	*Gebdatum*	*Strasse*	*PLZ*	*Ort*
1	Muster-mann	Max	01.01.1975	Mustergasse 55	55555	Muster-hausen
2	Muster-frau	Sabine	01.01.1970	Hauptstrasse 7	24146	Kiel
3	Doe	John	12.03.1968	Torweg 12	23552	Lübeck

Das Ausführen dieser Anfrage liefert *12.03.1968*.
Da das Programmieren von Ausdrücken der relationalen Algebra nicht intuitiv ist und als nicht praktikabel angesehen wurde, ist die deklarative Anfragesprache SQL in den 1970er Jahren von IBM basierend auf einem Artikel von E. Codd entwickelt worden [Cod70].
Aufgrund der in SQL relativ einfach gehaltenen Syntax und dem hohen Funktionsumfang der Sprache hat sich SQL im Verlauf der Jahre zur Standardanfrageansprache entwickelt, die bis auf Syntaxunterschiede und proprietäre Erweiterungen auf allen populären Relationalen Datenbank Management Systemen (RDBMS) den gleichen Funktionsumfang bietet.
Die obige Abfrage sieht in SQL wie folgt aus:

```
SELECT Gebdatum
  FROM Personen
 WHERE Name = 'Doe'
   AND Vorname = 'John'
```

Diese Abfrage an die Datenbank wird aus gängigen Objektorientierten Programmiersprachen (z.B. Java, Delphi, C++) häufig durch das hardcodierte Manipulieren von Zeichenketten erzeugt. In Delphi wird die Anfrage beispielsweise wie folgt zusammengesetzt

```
var
  LStatement: String;
  ...
begin
  ...
  LStatement :=
    'SELECT Gebdatum'#13#10 +  // #13#10 für einen Zeilenumbruch
    '  FROM Personen'#13#10 +
    ' WHERE Name = ' + quotedStr('Doe') + #13#10 +
    '   AND Vorname = ' + quotedStr('John');
  ...
```

Anhand dieses einfachen Beispiels, sieht man sofort, welche Probleme auftreten können, wenn die Datenbankanfragen aus einer Programmiersprache durch das Zusammensetzen von Strings erzeugt werden, beziehungsweise fest im Quelltext codiert werden.

Der Compiler hat keine Möglichkeit, die syntaktische Korrektheit der Anfrage zu überprüfen; beispielsweise könnten SQL Schlüsselwörter falsch geschrieben worden sein, die Anfrage nicht vollständig sein oder die Tabelle nicht in der Datenbank vorhanden sein. Des Weiteren kann der Compiler auch keine Typüberprüfung vornehmen. Ist ein Attribut einer Tabelle (bspw. *ID*) vom Typ `Numeric` und wird eine Selektion anhand dieses Feldes gegen einen `Char`-Wert durchgeführt ($\sigma_{ID='Z'}Personen$), resultiert auch dies in einem Fehler auf Datenbankebene, der nicht von Compiler entdeckt werden kann.
Eine weitere nicht zu vernachlässigende Fehlerquelle ist eine Strukturänderung der Datenbank. Wird so ein Schritt durchgeführt muss das gesamte Programm von Hand überarbeitet und die betreffenden SQL-Anfragen müssen entsprechend angepasst werden. Dies ist eine große Fehlerquelle, da es im Entwicklungsprozess immer wieder auch zu undokumentierten Änderungen an der Datenbank kommen kann oder SQL-Ausdrücke bei der Überarbeitung nach einer Änderung übersehen beziehungsweise nicht korrekt überarbeitet werden.

2.2 Haskell

Haskel ist eine stark typisierte, rein funktionale Programmiersprache. Das heißt, daß ein Typsystem zu grunde liegt, mit welchem schon zur Compilierzeit Typüberprüfungen vorgenommen werden und so Typsicherheit garantiert wird.
Bei der Definition von Funktionen lassen sich die Typen der Ein- und Ausgabeparameter in der folgenden Notation, die sogenannte Signatur, angegeben:

```
name :: type
```

Funktionen selbst werden in der folgenden Notation angegeben

`name` $param_1$ $param_2$ `... = expression`.

Der angegebenen Notation werde ich auch in diesem Dokument folgen, so daß als konkretes Beispiel die Fibonacci Funktion in der folgenden Art und weise angegeben wird:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

Um oft wiederverwertbaren Code zu schreiben, lässt sich bei der Funktionsdeklaration auch die Typklasse der Parameter angeben. Beispielsweise könnte die `fib` Methode mit der Signatur `fib :: Num a => a -> a` angegeben werden und ließe sich dann für alle Parameter `a` auswerten, die aus

der Klasse `Num` stammen.
Hierbei sei angemerkt, daß die Argumente von Funktionen nach der *lazy evaluation* ausgewertet werden. Da es aber auch Funktionen gibt, die nicht Seiteneffektfrei sind und damit in einer bestimmten Reihenfolge abgearbeitet werden müssen, werden diese in einer Monade ausgeführt. Als Beispiel sei hier der Schreib-/Lesezugriff auf das Dateisystem erwähnt, welcher in der so genannten IO Monade durchgeführt wird.
Zur besseren Strukturierung lassen sich Berechnungen mit Seiteneffekten in der `do{}`-Notation zusammengefassen. Da Anfragen an Datenbanken nicht seiteneffektfrei sind und um die Lesbarkeit zu erhöhen, werde ich in dieser Ausarbeitung Quellcode häufig in der `do{}`-Notation angeben und zusätzlich die postfix Notation

```
(#) :: a -> (a -> b) -> b
object # method = method object
```

für Funktionsanwendung verwenden, um so einen Stil ähnlich dem in der Objektorientierten Programmierung gängigen Postfix Notation `object.methode` zu erreichen. Für weitere Informationen zu Haskell und zu der verwendeten Syntax sei hiermit auf [HaD] verwiesen.

Um die im vorigen Abschnitt erwähnten Fehlerquellen im Zusammenhang mit SQL auszuschließen und Haskell um die Möglichkeit zur Datenbankanbindung zu erweitern, wurde das HaskellDB Framework entwickelt. Wie obiges Beispiel konkret in Haskell implementiert wird und welche besonderen Sprachfeatures von Haskell bei der Implementierung von HaskellDB verwendet wurden, werde ich im folgenden erläutern.

3 HaskellDB

3.1 Grundlagen

Um Datenbankanfragen an einen Server zu schicken und das Ergebnis empfangen zu können, wird das folgende in Interface Definition Language (IDL) formulierte Interface verwendet

```
interface IServer
  { void SetRequest([in, string] char* expr);
    void GetResponse([out] char* result);
  }
```

Dieses Interface wird von einem externen Server (bspw. ein Datenbankmanagementsystem) durch COM zur Verfügung gestellt. Unter Zuhilfename des HaskellDirect (HDirect) Compilers, lässt sich dieses Interface für Haskellapplikationsentwickler zur Verfügung stellen.

Der HDirect Compiler kümmert sich um die lowlevel Funktionen, die zur Transformation der Datentypen - das sogenannte Marshalling - der beiden unterschiedlichen Systeme benötigt werden. Weitere Informationen zum COM Binding in Haskell finden sich in [Lej98].
In diesem Fall wird eine COM Funktion, die vom Datenbankserver zur Verfügung gestellt wird, importiert. Wie man leicht erkennt sind die beiden zur Verfügung gestellten Funktionen für eine Client-Server-Kommunikation vollkommen ausreichend, da man zum einen eine Möglichkeit hat, eine Anfrage an den Server abzusetzen und zum anderen das Ergebnis erhalten kann. Die Signatur der beiden bereitgestellten Funktionen lautet wie folgt

```
setRequest  :: String -> IServer s -> IO ()
getResponse :: IServer s -> IO String
```

Basierend auf den beiden Funktionen des Interfaces lässt sich nun eine Auswertungsfunktion schreiben, die eine SQL-Anfrage entgegennimmt und das Ergebnis zurückliefert.

```
runExpr :: String -> IO Int
runExpr = \expr ->
  do{ server <- createObject "Expr.Server"
    ; server # setRequest expr
    ; result <- server # getResponse
    ; return (read result)
    }
```

Diese Funktion stellt die Kommunikation mit einem Server dar, der eine Numerische Rückgabe liefert - etwa einen Fehlercode. Daher hat die Funktion `runExpr` den Ergebnistyp `IO Int`. Diese Funktion sei hier nur als Beispiel der prinzipiellen Funktionsweise einer Client-Server-Kommunikation angegeben.
Da die Methode `runExpr` jedoch einen SQL-Ausdruck in Form einer Zeichenkette als Eingabe erwartet und damit wieder die Eingangs erwähnten Probleme im Zusammenhang mit den SQL-Anfragen (syntaktische und semantische Inkorrektheit) auftreten können, wird diese Methode dem Haskellentwickler im HaskellDB Framework nicht zur Verfgung gestellt.
Ziel ist es also, das Erstellen von SQL-Anfragen vom Framework übernehmen zu lassen und damit sowohl syntaktische als auch semantische Korrektheit zu erhalten.
Wie eingangs erwähnt, basieren die Datenbankanfragen in SQL auf der Relationalen Algebra. Daher ist es ausreichend, eine Syntax für Ausdrücke in relationaler Algebra in Haskell einzubetten. Damit diese Einbettung vorgenommen werden kann, müssen zuerst einige Grundlagen, die für die Einbettung der Syntax notwendig sind, erläutert werden. Zu diesen Grundlagen gehören die Einführung von syntaktischer Korrektheit und Typsicherheit.

Wie beides erreicht wird, werde ich in den kommenden Abschnitten zeigen und anschließend aufzeigen, wie die SQL-Syntax in Haskell eingebettet wurde.
Wo immer es möglich ist, werde ich erwähnen, warum die zu dem Zeitpunkt eingeführten Typen und Funktionen noch nicht ausreichen, um das gewünschte Framework bieten zu können.

3.2 Syntaktische Korrektheit

Bei der Implementierung von HaskellDB wird für die syntaktische Korrektheit von Ausdrücken relationaler Algebra ein Datentyp eingeführt, mit dem sich SQL-Anfragen in einer abstrakten Syntax formulieren lassen. Diese abstrakten Ausdrücke werden zur Ausführung durch einen Codegenerator in einen syntaktisch korrekten SQL Ausdruck übersetzt.
Da der Einbettung von Ausdrücken relationaler Algebra in Haskell eine generelle Vorgehensweise zugrunde liegt und weil anhand dieser die Besonderheiten aufgezeigt werden können, betrachten wir zuerst einige grundlegende Eigenschaften, die bei der Einbettung einer Sprache in Haskell vorliegen.
Der Datentyp für die abstrakte Syntax ist wie folgt definiert

```
data PrimExpr
  = BinExpr   BinOp PrimExpr PrimExpr
  | UnExpr    UnOp PrimExpr
  | ConstExpr String
```

Die Typen `BinOp` und `UnOp` sind einfach Aufzählungen der Binären und Unären Operatoren der Relationalen Algebra.

```
data BinOp = OpEq | OpAnd | OpPlus | OpOr | ...
data UnOp  = OpNot | OpAsc | OpDesc | ...
```

Da das Formulieren von Ausdrücken in der abstrakten Syntax nicht sehr intuitiv ist, lassen sich diese Operatoren in Haskell durch fast die gleichen Operatoren schreiben, so daß die Ausdrücke dadurch intuitiver werden. Der Additionsopertator der relationalen Algebra `+` ist in Haskell wie folgt definiert

```
binop :: BinOp -> Expr a  -> Expr b -> Expr c
binop op (Expr primExpr1) (Expr primExpr2)
                = Expr (BinExpr op primExpr1 primExpr2)

numop :: Num a => BinOp -> Expr a -> Expr a -> Expr a
numop   = binop

add x y = numop OpPlus x y

(.+.) x y = add x y
```

so daß die Addition der Konstanten 1 und 4 wie folgt formuliert wird

```
-- Definition von constant
constant :: ShowConstant a => a -> Expr a
constant x      = Expr (ConstExpr (showConstant x))

-- Beispiel
constant 1 .+. constant 4
```

Mit dem Datentyp `PrimExpr` wird die syntaktische Korrektheit von Vergleichen garantiert, allerdings ist es noch immer leicht möglich falsch typisierte Vergleiche zu erzeugen. Beispielsweise ist der folgende Ausdruck durch die obige Definition der Datentypen und Operatoren syntaktisch korrekt, aber durch die falsche Typisierung wird eine RDBMS diesen Ausdruck als fehlerhaft zurückweisen

```
constant "Hello" .-. constant "World"
```

Mit der bis zu diesem Zeitpunkt vorgestellten Syntax lassen sich also bereits sämtliche Vergleiche, die innerhalb der `WHERE` Bedingung auftreten können formulieren. Die Relationale Algebra beschränkt sich aber, wie eingangs gezeigt, nicht auf Vergleiche, sondern bietet auch Selektionen und Projektionen. Damit man die Abstrakte Syntax für die Formulierung von Ausdrücken der Relationalen Algebra überhaupt einsetzen kann, muss `PrimExpr` also erweitert werden. Wie diese Erweiterung vorgenommen wird, und was dabei zu beachten ist, werde ich im folgenden erläutern.

3.3 Typsicherheit

Da der primitive Datentyp `PrimExpr` keine Typsicherheit bietet, wird dem Entwickler der Zugriff auf den primitiven Datentyp verwehrt und liegt nur zur internen Verwendung vor.
Die Typsicherheit der Anfragen wird mit *Phantom Typisierung* eingeführt. Dies geschieht indem der vorliegende Datentyp `PrimExpr` durch einen polymorphen Datentyp gekapselt wird. Der neue Datentyp ist wie folgt deklariert

```
data Expr a = Expr PrimExpr
```

Die Typvariable `a` ist in der Definition nur dafür da, einen Typ zu repräsentieren. Da die Variable auf der rechten Seite der Definition aber nicht vorkommt, ist `a` nie physikalisch im Hauptspeicher anwesend. Verwendet man nun diesen Datentyp für die Deklaration der Operationen, lässt sich die Typsicherheit garantieren. Die Definition der Addition bei Verwendung von Typklassen wird dabei wie folgt abgeändert

```
(.+.) :: Num a => Expr a -> Expr a -> Expr a
```

Verwendet man die entsprechend abgeänderten Methoden, dann kann der Compiler falsch typisierte Ausdrücke wie `constant "z" .+. constant "U"` zurückweisen, so daß solche Fehler nicht erst zur Laufzeit auftreten und vom Entwickler schon zur Entwicklungszeit korrigiert werden können.

3.4 Einbettung von Relationaler Algebra

Mit den Informationen aus den letzten beiden Abschnitten lässt sich also die Syntax zum Erzeugen von Ausdrücken der Relationalen Algebra in Haskell einbetten. Die konkrete Implementierung und einige Beispiele werde ich im Folgenden darstellen.
Die oben eingeführte abstrakte Syntax als Grundlage nehmend, sind entsprechende Modifikationen und Erweiterungen für den konkreten Einsatz zur Entwicklung einer Datenbankapplikation vorgenommen worden.
Als Basistypen wurden die folgenden Datentypen eingeführt

```
type TableName = String
type Attribute = String
type Scheme    = [Attribute]
type Assoc     = [(Attribute,PrimExpr)]
```

Zur Abfrage von Daten aus Tabellen werden entsprechend der relationalen Algebra Projektionen und Restriktionen auf Tabellen ausgeführt, es wurde also ein Datentyp eingeführt, mit dem es möglich ist, konkrete Anfragen zu erzeugen.

```
data PrimQuery
  = BaseTable TableName Scheme
  | Project   Assoc     PrimQuery
  | Restrict  PrimExpr  PrimQuery
  | Binary    RelOp     PrimQuery PrimQuery
  | Empty
```

Der in `PrimQuery` verwendete Datentyp `RelOp` definiert die Operationen der Relationalen Algebra. Die konkrete Implementierung lautet wie folgt:

```
data RelOp
  = Times
  | Union
  | Intersect
  | Divide
  | Difference
```

Der in der Definition von `PrimQuery` verwendete Datentyp `PrimExpr` ist analog zum oben eingeführten wie folgt definiert

```
data PrimExpr
  = BinExpr   BinOp PrimExpr PrimExpr
  | UnExpr    UnOp PrimExpr
  | ConstExpr String
  | AttrExpr  Attribute
```

Mit diesen Datentypen lässt sich die Eingangs gestellte Beispielabfrage

$$\pi_{Gebdatum}(\sigma_{Vorname=John,Name=Doe}Personen)$$

im Stil der Relationalen Algebra wie folgt konstruieren

```
Project  [("Gebdatum", AttrExpr "Gebdatum")]
  (Restrict (BinExpr OpEq (AttrExpr "Vorname")
                          (ConstExpr "John"))
    (Restrict (BinExpr OpEq (AttrExpr "Name")
                            (ConstExpr "Doe"))
      (BaseTable "Personen"
        ["ID", "Name", "Vorname",
"Gebdatum", "Strasse", "PLZ", "Ort"]
      )))
```

Wie man leicht sieht, ist das Formulieren von Anfragen in dieser Art und Weise sehr kompliziert, so daß auch für die abstrakte Syntax der relationalen Algebra die oben gezeigten entsprechenden Vereinfachungen eingeführt wurden.
Zusammen mit den Vereinfachungen ist es also möglich Ausdrücke Relationaler Algebra zu formulieren, die sich mit einem Codegenerator in syntaktisch korrekte SQL-Ausdrücke übersetzen lassen. Die weiter oben vorgestellte Funktion `runExpr` kann also so modifiziert werden, daß Sie einen Ausdruck vom Typ `PrimQuery` erwartet, diesen mit einem Codegenerator in die entsprechende Stringrepräsentation übersetzt und die so generierte Zeichenkette an den Server schickt.
Ein weiteres Problem, das sich im Einsatz ergibt, sind Vergleiche über Spalten mit gleichem Namen. Bisher existiert noch kein Mechanismus, um Abfragen der folgenden Form verarbeiten zu können

```
SELECT p1.Name, p1.Gebdatum
  FROM Personen p1, Personen p2
 WHERE p1.Name = p2.Name
   AND p1.Gebdatum <> p2.Gebdatum
```

Denn bisher werden die Felder in Ausdrücken der abstrakten Syntax über Ihren zugehörigen Feldnamen qualifiziert. Daher ist es nicht möglich eine Unterscheidung zwischen den Feldern aus `p1` und `p2` vorzunehmen. Da die Kartesischen Produkte allerdings ein sehr häufig verwendetes und nützliches Feature von SQL Statements sind, werden diese auch vom Haskell/DB

Framework unterstützt. Wie die Unterstüztung implementiert und die Lesbarkeit der Ausdrücke erhöht wurde, zeige ich kommenden Abschnitt.

3.5 Query-Monade

Um die am Schluss des letzten Abschnitts angegebene Anfrage an eine Datenbank formulieren zu können, wird die abstrakte Syntax so modifiziert daß die Attribute nicht nur durch den Spaltennamen sondern zusätzlich auch durch den Namen Ihrer zugehörigen Tabelle qualifiziert werden. Dieser Ansatz ist nicht neu, denn er wird bereits seit langer Zeit bei der Verarbeitung von SQL-Anfragen angewendet.
Da eine von Hand durchgeführte Umbenennung einzelner Attribute zur eindeutigen Qualifizierung sehr aufwändig und fehleranfällig wäre, wurde dieser Mechanismus in das Haskell/DB-Framework integriert. Die technischen Details der automatischen Umbenennung finden sich in [DL99].
Im Zuge dieser Entwicklung im Haskell/DB Framework wurde auch die `Query` Monade eingeführt. Die automatische Umbennenung wird in der `Query` Monade transparent durchgeführt, so daß sich der Entwickler nicht um die weiteren Details kümmern muss. Zusätzlich hat der Entwickler die Möglichkeit, Relationen an eine Variable zu binden, was wiederum die Lesbarkeit und damit einhergehend auch die Wartbarkeit erhöht.

```
do{ p <- table personen --binding
   ...
```

Damit die `do`-Notation überhaupt verwendet werden kann, ist hier die Definition der Query Monade und der Monaden Operationen `returnQ` und `bindQ` angegeben.

```
type QState = (Alias, PrimQuery)
data Query a = Query (QState -> (a, QState))

returnQ :: a -> Query a
bindQ   :: Query a -> (a -> Query b) -> Query b
```

Wie man hier sieht wird in dieser Monade ein unsichtbarer Zustand `QState` vorgehalten, der zum einen den bisher erstellten Ausdruck der relationalen Algebra enthält (`PrimQuery`) und zum anderen eine Quelle von bisher nicht verwendeten Namen für die automatische Umbenennung von Attributen in `Alias` mitführt.
Führt man noch zusätzlich den `!`-Operator ein, mit dem man gezielt auf ein Attribut einer Relation zugreifen kann, lassen sich Anfragen in einer sehr intuitiven Schreibweise formulieren. Die Definition von Rel ist weiter unten angegeben.

```
(!) :: Rel -> Attribute -> PrimExpr
```

Die im letzten Abschnitt angegebene SQL-Abfrage wird in einer Query Monade also wie folgt formuliert

```
do { p1 <- table personen
   ; p2 <- table personen
   ; restrict (p1!name     .==. p2!name)
   ; restrict (p1!gebdatum .<>. p2!gebdatum)
   ; project  (gebdatum = p1!gebdatum, name = p1!name)
   }
```

Die im Beispiel verwendeten Bezeichner `personen`, `vorname`, `name` und `gebdat` stammen aus dem mit DB/Direct erzeugten Schema. Intern generiert HaskellDB weiterhin Ausdrücke der Relationalen Algebra, die dann vom Codegenerator in die entsprechende Stringrepräsentation übersetzt werden.
Bisher lassen sich mit der Abstrakten Syntax gemäß der zugrunde gelegten relationalen Algebra syntaktisch korrekte Anfragen an die Datenbank formulieren. Allerdings ist es noch immer möglich, Vergleiche durchzuführen, die zwar syntaktisch korrekt, aber semantisch inkorrekt sind. Wie dieses Problem gelöst wird, zeige ich im nächsten Abschnitt.

3.6 Typsicherheit von Attributen

Da es bisher bisher immer noch möglich ist innerhalb von Querys Vergleiche durchzuführen die falsch typisiert wurden, lässt sicht die Typsicherheit ebenfalls durch Phantom Typisierung gewährleisten.
Anhand des in dieser Ausarbeitung häufig verwendeten Auswahloperators

```
(!) :: Rel -> Attribute -> PrimExpr
```

lässt sich die Typisierung leicht erklären.
Da ein Attribut von genau einem Typ ist, lässt sich die obige Definition so abändern, daß das Attribut durch seinen Typ parametrisiert wird und der Operator einen Ausdruck vom gleichen Typ zurückliefert.

```
data Attr a = Attr Attribute
(!) :: Rel -> Attr a -> Expr a
```

Allerdings ist der Entwickler auch so nicht dagegen abgesichert, Anfragen zu formulieren, in denen Felder vorkommen, die gar nicht in der Tabelle enthalten sind. Um dies zu verhindern, werden die typisierten Vergleiche zusätzlich durch ihr Datenbankschema parametrisiert.

```
data Rel r    = Rel   Alias Scheme
data Table r  = Table TableName Assoc
data Attr r a = Attr Attribute
```

Damit das Schema nicht für jede Datenbank von Hand erzeugt und gepflegt werden muss, wurde das Werkzeug *DB/Direct* entwickelt, welches die Schemainformationen aus der Datenbank auslesen kann und die entsprechenden Haskellklassen und Quellcode Dateien erzeugt. Da *DB/Direct* mit HaskellDB entwickelt wurde, unterstützt es die gleichen Datenbanken, wie HaskellDB.
Mit dem erzeugten Schema kann der Haskell Typ Checker jede Anfrage auf ihre Korrektheit überprüfen und im Fehlerfall zur Kompilierzeit aufzeigen, wo der Fehler liegt.

Damit sind nun die Funktionen geschaffen worden, mit deren Hilfe sich syntaktisch und semantisch korrekte Datenbankzugriffe in Haskell formulieren lassen. Insbesondere ist damit ein großes Problem in der Datenbankapplikationsentwicklung beseitigt worden, denn Fehler im Datenbankzugriff treten nicht mehr erst zur Laufzeit auf, sondern werden schon während der Compilierzeit aufgezeigt und können somit schnell behoben werden.

3.7 ADO

Viele Datenbankzugriffe liefern ein Ergebnis zurück, dieses muss natürlich von der Applikation empfangen und verwaltet werden können. Für die Kommunikation mit dem Datenbankserver werden in der vorliegenden Version von HaskellDB die ActiveX Data Objects (ADO) verwendet. Das oben erwähnte HDirect-Framework wird verwendet, um die von ADO bereitgestellten Funktionen und Interfaces in Haskell zu verwenden.
ADO liefert eine Abfrage in Form eines `RecordSet`-Objektes zurück. Dieses Objekt ist in IDL wie folgt definiert

```
dispinterface RecordSet {
  void Open
    ([in,optional] VARIANT Source
    ,[in,optional] VARIANT ActiveConnection
    ,[in,optional] CursorTypeEnum CursorType
    ,[in,optional] LockTypeEnum LockType
    ,[in,optional] long Options
    );

  Bool    EOF();
  void    MoveNext();
  Fields* GetFields();
}
```

Die Methode `Open` erwartet als ersten Parameter `Source` den Ursprung der Datensätze. Die Quelle kann ein Tabellennamen oder eine SQL-Anweisung sein, aus der die Datensätze stammen. Der zweite Parameter `ActiveConnection`

kann entweder eine bereits bestehende Verbindung oder ein String zum Aufbau einer neuen Verbindung sein. Die weiteren optionalen Parameter werden hier nicht verwendet, daher ist die Signatur der `Open` Funktion wie folgt definiert

```
open :: (VARIANT src, VARIANT actConn) =>
  src -> actConn -> IRecordSet r -> IO()
```

Die vom `RecordSet`-Objekt bereitgestellten Methoden `EOF(), MoveNext()` dienen zur unidirektionalen Navigation auf der Datenmenge und haben die folgenden Signaturen

```
moveNext  :: IRecordSet r -> IO ()
eof       :: IRecordSet r -> IO Bool
```

Mit der Methode `GetFields()` erhält der Entwickler eine Liste aller Felder des aktuellen Datensatzes. Diese Felder werden in Form einer Liste von `Fields` zurückgeliefert.

```
-- Signatur von GetFields()
getFields :: IRecordSet r -> IO (IFields ())

-- IDL Definition von Fields
dispinterface Fields {
  long   GetCount();
  Field* GetItem([in] VARIANT Index);
}
```

Auf den Wert und den Namen eines Feldes kann man über die folgenden Methoden des `Field`-Interfaces zugreifen.

```
getValue :: VARIANT a => IField f -> IO a
getName  :: IField f -> IO String
```

Die Bezeichner, die mit einem `I` beginnen, sind die vom HDL-Compiler in Haskell erzeugten Typen der bereitgestellten Interfaces. `IRecordSet` in Haskell entspricht also dem bereitgestellten `RecordSet` Interface.
Mit diesen Funktionen lassen sich Datenmengen, die von einem Datenbankzugriff durch die `open`-Funktion zurückgeliefert werden, verarbeiten. Da man in Haskell typischerweise über Listen iteriert, verarbeitet man nicht die `IRecordSet` Datenstruktur sondern eine Liste von `IFields`, die mit folgender Methode zurückgeliefert wird.

```
readFields :: (IO a -> IO a) -> IRecordSet r -> IO [IFields ()]
readFields perform records = perform $
  do{ atEOF <- records # eof
    ; if atEOF
```

```
      then do{ return [] }
      else do{ field <- records # getFields
             ; records # moveNext
     ; rest <- records # readfields perform
     ; return ([field] ++ rest)
     }
    }
```

Mit dem Parameter `perform` lässt sich steuern, ob die zurückgelieferte Liste sofort (eagerly) oder erst zum Zeitpunkt des tatsächlichen Zugriffs (lazily) erzeugt wird.
Mit diesen hier vorgestellten Funktionen ist die grundlegende Serverkommunikation abgedeckt, so daß es zusammen mit der weiter oben vorgestellten Einbettung von Relationaler Algebra in Haskell möglich ist, Datenbankapplikationen zu entwickeln.

3.8 Beispiel

Mit der eingebetteten Syntax der Relationalen Algebra und den eingeführten Funktionen zum Datenbankzugriff via ADO lassen sich also Datenbankapplikationen mit Haskell entwickeln.
Betrachten wir die Anfangs vorgestellte Relation *Personen* können wir darauf beispielsweise folgende Funktion definieren, die das Geburtsdatum zu einem gegebenen Personen ID zurückliefert.

```
type Person = Row(id :: Int, name :: String, ...)

-- Abfrage erzeugen
showDate :: Int -> Query Personen
showDate aid =
  do{ p <- table personen
    ; restrict (p!id .==. aid)
    ; project (gebdat = p!gebdatum)
    }

-- Abfrage ausführen. Verwende dabei die
-- entsprechend angepasste Variante von runQuery
doEvaluate :: Int -> IO ()
doEvaluate aid =
 do{ d <- runQuery (showDate aid)
   ; return (show d)
   }
```

4 Zusammenfassung

In der heutigen Zeit der Informationsverarbeitung basieren die meisten Applikationen auf Daten die aus einer Datenbank stammen. Daher ist mit dem HaskellDB-Framework eine sehr nützliche Erweiterung entwickelt worden, denn die Gemeinde der Haskellentwickler ist nun nicht mehr auf eventuell fehleranfälligen Eigenentwicklungen zur Datenspeicherung angewiesen. Gleichzeitig ist durch die Erweiterung und die Ablösung von der proprietären TRex-Erweiterung durch B. Bringert und A. Höckersten die Portierbarkeit erhöht worden, so daß sich Datenbankapplikationen leicht auf unterschiedlichen System ausführen lassen.
Abschließend betrachtet, kann man sagen, das sich mit geringem zusätzlichem Aufwand typsichere und sowohl syntaktisch als auch semantisch korrekte Anfragen an die Datenbank erzeugen lassen. Durch die nicht mehr benötigte Stringmanipulation zur Erzeugung von SQL-Anfragen ist eine große Fehlerquelle beseitigt worden und mit dem Werkzeug DB/Direct werden Anpassungen nach Strukturänderungen der Datenbank zu einer leichten Aufgabe, da der Typchecker von Haskell fehlerhaften Quellcode sofort kennzeichnet.

Literatur

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 1970.

[DL99] E. Meijer Daan Lejien. Translatin do-notation to sql, 1999.

[HaD] *Haskell Documentation*. `http://www.haskell.org`.

[Lej98] Daan Lejien. Haskell direct master thesis. `http://www.cs.uu.nl/~daan/download/papers/master-thesis.ps`, 1998.