Christian-Albrechts-Universität zu Kiel

Master Thesis

Typsichere Einbettung von Datenbankanfragen in Scala

Christoph Wulf

30. März 2012

Institut für Informatik
Lehrstuhl für Programmiersprachen und
Übersetzerkonstruktion

betreut durch:
Prof. Dr. Michael Hanus

Eidesstattliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt zu haben.

Kiel, 30. März 2012 _____

Christoph Wulf

Abstract

Der Datenbankzugriff ist in gängigen Programmiersprachen unkomfortabel und fehleranfällig. Das Kodieren von Anfragen als Zeichenketten und die beim Verarbeiten von Ergebnissen notwendige Typumwandlung führt zu Laufzeitfehlern, die oftmals durch eine statische Analyse identifiziert werden könnten.

In dieser Master Thesis wird ein Konzept zur Einbettung von Datenbankanfragen in die Sprache Scala verfeinert, erweitert und umgesetzt. Eingebettete Datenbankabfragen werden statisch analysiert und typsicher in das umgebende Scala-Programm integriert.

Dabei wird einerseits das starke Typsystem von Scala, aber auch die flexible Architektur des Scala-Übersetzers verwendet.

Die Arbeit geht insbesondere auf die Plug-in-Architektur des Scala-Übersetzers und die Typsicherheit der eingebetteten Anfragen ein.

Inhaltsverzeichnis

1. Einleitung

Datenbanken sind für die heutige Software-Entwicklung unerlässlich. Insbesondere bei der Entwicklung von Diensten im Internet sind Datenbanksysteme essentiell. Ein Großteil dieser Anwendungen speichert, lädt und aktualisiert geschäftsrelevante Daten, die nicht beispielsweise durch einen Stromausfall verloren gehen dürfen und daher nicht ausschließlich im Hauptspeicher vorgehalten werden können.

Um einen solchen Datenverlust zu vermeiden ist eine persistente Speicherung z.B. auf einer Festplatte notwendig. Die Speicherung in einem applikationsspezifischen Format in Dateien ist besonders in internetbasierten Systemen problematisch. Da dort mehrere Anfragen parallel bearbeitet werden können, ist die notwendige Synchronisation beim Lesen und Schreiben in Dateien entweder sehr komplex oder führt zu einem Verlust von Performanz.

Relationale Datenbanken ermöglichen die persistente Speicherung bei einem effizienten Lese- und Schreibzugriff, auch bei parallelen Anfragen. Die meisten relationalen Datenbanken bieten eine Schnittstelle über die Structured Query Language (SQL), deren Grundlagen in Kapitel 4 vorgestellt werden.

Trotz der gegebenen Notwendigkeit für relationale Datenbanken ist die Verwendung von SQL in den gängigen Programmiersprachen zur Anwendungsentwicklung sehr unkomfortabel. Datenbankanfragen werden in der Regel[1] als Zeichenketten eingebettet, die an den Datenbankserver abgesetzt und dort ausgewertet werden. Die Ergebnisse müssen per Typumwandlung vom Programmierer in der anfragenden Anwendung konvertiert werden.

Die Anfragen werden im Datenbankmanagementsystem übersetzt. Programmierfehler, die in einer übersetzten und statisch typisierten Sprache schon zur Übersetzungszeit auftreten würden, wie Syntax- oder Typfehler treten erst zur

[1] z.B. in Perl, PHP, JDBC, ADO.NET

Laufzeit auf. Fehler dieser Art in der Anfrage treten zur **Übersetzungszeit der Anfrage** auf, die wiederum die **Laufzeit des Programms** ist.

Das folgende Beispiel zeigt den ersten Lösungsentwurf für eine Übungsaufgabe zur Berechnung der kürzesten Verbindung von Lübeck nach Alicante.

```
1  WITH RECURSIVE routes (iteration, route, departure, depTime, arrival,
       arrTime, total) AS (
2    SELECT 1, departure || ' -' || no || '-> ' || arrival, departure,
         depTime, arrival, arrTime, price + 0
3    FROM traveloptions
4    UNION
5    SELECT iteration + 1, route || ' -' || no || '-> ' || t.arrival, r.
         departure, r.depTime, t.arrival, t.arrTime, total + price
6    FROM traveloptions t
7    JOIN routes r ON (t.departure=r.arrival AND t.depTime > r.arrTime)
8    WHERE iteration < 4
9  )
10 SELECT route, depTime, arrTime, arrTime - depTime AS time, total
11 FROM routes
12 WHERE departure = 'LBC' AND arrival = 'ALC' ORDER total, time
```

Beispiel 1.1: Rekursive Anfrage

In dieser rekursiven SQL-Anfrage wird die transitive Hülle der Verbindungen mit maximal vier Iterationen berechnet, aus denen die transitiven Verbindungen mit Anfangspunkt Lübeck und Endpunkt Alicante selektiert werden. Auch wenn das Datenbanksystem versucht, die Anfrage so zu optimieren, dass möglichst nur relevante Teile der transitiven Hülle berechnet werden, bietet dieser Lösungsentwurf noch reichlich Optimierungspotential für den Programmierer.

Beispiel 1.2 zeigt die Umsetzung der Anfrage in Java über die *Java Database Connectivity*[2] *(JDBC).*

[2] http://www.oracle.com/technetwork/java/javase/jdbc/index.html

```
1  StringBuilder sb = new StringBuilder();
2  sb.append("WITH RECURSIVE routes (iteration, route, departure, depTime, arrival, arrTime,
       total) AS (");
3  sb.append("  SELECT 1, departure || ' -' || no || '-> ' || arrival, departure, depTime,
       arrival, arrTime, price + 0");
4  sb.append("  FROM traveloptions");
5  sb.append(" UNION");
6  sb.append("  SELECT iteration + 1, route || ' -' || no || '-> ' || t.arrival, r.departure,
        r.depTime, t.arrival, t.arrTime, total + price");
7  sb.append("  FROM traveloptions t");
8  sb.append("  JOIN routes r ON (t.departure=r.arrival AND t.depTime > r.arrTime)");
9  sb.append("  WHERE iteration < 4");
10 sb.append("SELECT route, depTime, arrTime, arrTime - depTime, total");
11 sb.append("FROM routes");
12 sb.append("WHERE departure = ? AND arrival = ? ORDER BY total, time");
13
14 try {
15   PreparedStatement stm = connection.prepareStatement(sb.toString());
16   stm.setString(0, "LBC");
17   stm.setString(1, "ALC");
18   ResultSet rs = stm.executeQuery();
19   while (rs.next()) {
20     System.out.println("Route: " + rs.getString(0));
21     System.out.println("Total Costs: " + rs.getDouble(1));
22   }
23 } catch (Exception e) {
24   System.out.println("Runtime Exception: " + e);
25 }
```

Beispiel 1.2: Rekursive Anfrage in JDBC

Die eigentlich Anfrage wird als Zeichenkette integriert, was eine unübersichtliche Konkatenation erfordert. Die Zeichenkette ist mit Platzhaltern für Werte versehen, die in Zeile 16f. übergeben werden. Die Nummerierung der Spalten beginnt bei der *JDBC*-Programmierung mit der **1** und nicht wie üblich mit der **0**, was dazu führt, dass der zweite Platzhalter nicht gebunden wird. Die fehlende schließende Klammer, die in Zeile 10 erwartet wird, führt zu einem Syntaxfehler. Die Spalte `time`, nach der in Zeile 12 sortiert wird, existiert nicht. Das Ergebnis ist nicht typisiert, d.h. die projizierten Spalten müssen in 20f. per Typumwandlung extrahiert werden. Die falsche Nummerierung führt an dieser Stelle zu einem Typfehler, da die in Zeile 21 extrahierte erste Spalte vom Typ Zeichenkette ist, die nicht als Fließkommazahl extrahiert werden kann.

All diese Fehler, die erst zur Laufzeit des Programms auftreten, könnten bei der Übersetzung der Anfrage zur Übersetzungszeit des Programms verhindert werden.

Die Unübersichtlichkeit und Fehleranfälligkeit der Datenbankprogrammierung

mit *JDBC* führt dazu, dass viele Programmierer komplexe Datenbankabfragen in einzelne überschaubare Anfragen aufteilen. Im oben stehenden Beispiel würde die Rekursion aus der Datenbankanfrage in die Programmiersprache der Anwendung gezogen, was dem Datenbanksystem allerdings weitaus weniger Spielraum für Optimierungen gibt. Weit verbreitet ist auch der Verzicht auf die direkte Verwendung von *JDBC* durch den Einsatz von Persistenzbibliotheken, die auf *JDBC* aufbauen.

1.1. Objektrelationale Unverträglichkeit

Gerade im Bereich der Sprachen für die *Java Virtual Machine* (JVM) ist der Einsatz von Bibliotheken zur objektrelationalen Abbildung (O/RM[3]) weit verbreitet. Diese bilden Datenbanktabellen auf korrespondierende Klassen, Zeilen einer Tabelle auf passende Objekte ab. Die Abbildung des relationalen Modells auf das objektorientierte Modell ist nicht immer optimal, was als objektrelationale Unverträglichkeit[4] [7] bezeichnet wird. Nicht alle Anfragen lassen sich aus objektorientierter Sichtweise (gut) in SQL ausdrücken, während sich nicht alle gängigen Strukturen relationaler Datenbanken optimal auf Objekte abbilden lassen. Beispielsweise stellt die *Lift Persistence* Bibliothek [1], geschrieben in der JVM-Sprache Scala, zwar typsichere Anfragen zur Verfügung, kann aber keinen relationalen Verbund ausdrücken, der für Datenbankanfragen essentiell ist.

1.2. Zielsetzungen

Die Programmiersprache Scala, deren Grundlagen in Kapitel 2 vorgestellt werden, gilt als objektfunktionale Sprache. Als objektorientierte JVM-Sprache mit Komponenten aus der funktionalen Programmierung enthält sie Tupel als referentiell transparente Objekte. Tupel kommen der Darstellung von Relationen sehr nahe und sind damit optimal zur Repräsentation der Ergebnisse von Abfragen.

[3] englisch object-relational mapper
[4] englisch object-relational impedance mismatch

Diese Arbeit greift die Idee einer Einbettung von SQL-Anfragen in Scala auf, die unveröffentlicht auf der Scala-Konferenz[5] 2010 in Lausanne vorgestellt wurde [10]. Ziel ist eine möglichst umfassende Einbettung in Scala bei direkter SQL-Notation auf der Basis von statisch typisierten Ergebnissen. Nichtfunktional wird eine möglichst knappe Implementierung angestrebt, die bestehende Fähigkeiten des Scala-Compilers weitestgehend nutzt, um schnell auf Änderungen im Scala-Übersetzers reagieren zu können.

Im Artikel von 2010 wird vorgeschlagen, SQL-Anfragen in eine typsichere Repräsentation in Scala zu transformieren, die auf ebenfalls in Scala-Repräsentationen übersetzen Datenbankschemata arbeiten. Dieser Ansatz zeigt zwar das komplexe Typsystem der Sprache Scala auf, überlässt die Typisierung von Anfragen an den Scala-Übersetzer und löst die Notwendigkeit, bei jeder Übersetzung das Datenbankschema zu übergeben, hat aber diverse Nachteile. Der Ansatz erfordert sehr komplexen Zielcode. Abstrakte Syntaxbäume für Anfragen müssen separat sowohl auf der Übersetzer- als auch auf der Bibliothekebene implementiert werden. Der komplexe Zielcode führt bei fehlerhafter Anwendung zu sehr kryptischen Fehlermeldungen im transformierten Code, den der benutzende Programmierer nicht zu Gesicht bekommt und daher auch nicht korrigieren kann. Weiterhin können manche einfache Optimierungen nicht im Typsystem von Scala abgebildet werden. Es wäre auf der Übersetzerebene kein Problem, zu prüfen, ob alle Felder eines eindeutigen Schlüssels einer Tabelle an Werte gebunden sind. Das Ergebnis einer solchen Abfrage könnte dann als eine oder keine Zeile anstelle von beliebig vielen Zeilen typisiert werden. Dies ist allerdings im Scala-Typsystem nicht möglich.

Daher wird in dieser Arbeit die Typisierung der Ergebnisse von Anfragen vollständig in die Erweiterung des Übersetzers verlagert. Die Übergabe des Datenbankschemas als Übersetzerargument, die dann wieder notwendig ist, sollte beim Einsatz von gängigen Entwicklungsumgebungen oder Werkzeugen kein Problem sein.

Im Artikel wird vorgeschlagen, ähnlich der Unterstützung von XML in Scala, die Syntaxanalyse direkt um die SQL-Notation zu erweitern. Dies hat den Nachteil, das direkt am Quellcode des Scala-Übersetzers gearbeitet werden muss und daher jede Auslieferung einer neuen Scala-Version eine Anpassung erfordert. Stattdessen ist eine Umsetzung mit der Plug-in-Architektur des Scala-Übersetzers, die in Kapitel 3 vorgestellt wird, erstrebenswert. Dadurch

[5] http://days2010.scala-lang.org

dass XML-Notation in Scala-Programmen direkt erlaubt ist, lassen sich SQL-Anfragen in XML-Prozessoranweisungen einbetten:

```
val stm = <?sql SELECT * FROM actor WHERE actor_id = { scalaScopeId } ?>
```

Beispiel 1.3: eingebettete SQL-Anfrage

Dennoch soll das Typsystem von Scala intensiv genutzt werden. So soll das Extrahieren von typsicheren Ergebnissen von Abfragen möglichst mit den Mitteln der Sprache Scala realisiert werden, um die Erzeugung von Zielcode auf das Wesentliche und Notwendige zu reduzieren. Auch die Typprüfung von an Abfragen übergebener Werte (wie der Wert `scalaScopeId` in **Beispiel 1.3**) soll an den eigentlichen Scala-Übersetzer delegiert werden. Dadurch bleibt die Erweiterung frei von der Typprüfung von Scala-Ausdrücken und kann in einer einzigen Phase direkt nach der Syntaxanalyse umgesetzt werden.

1.3. Quellcode

Da die vorgestellte Übersetzererweiterung, die dazugehörige Bibliothek aber auch Beispielprogramme in Scala geschrieben sind, enthält diese Arbeit vier verschiedene Arten von Quellcode derselben Sprache. Diese werden durch unterschiedliche Hintergrundfarben und Rahmen unterschieden.

Der vollständige Code kann über die Internetseite[6] des Projekts inspiziert oder direkt über das Git Repository[7] heruntergeladen werden.

[6] Projektseite: `https://redmine.clinical-registry.com/projects/scalasql`
[7] Git Repository: `https://redmine.clinical-registry.com/projects/scalasql/repository`

	Beispielprogramme	Erweiterung
Übersetzungszeit	Originalprogramm	Übersetzer
Laufzeit	Zielprogramm	Bibliothek

Arten von Quellcode in dieser Arbeit

Die Beispielprogramme sind mit einem grauen Hintergrund hinterlegt, die zur Laufzeit relevanten Codeausschnitte komplett umrahmt.

```
val query = <?sql SELECT * FROM Users ?>
```

Quellprogramm

```
val query = new BagStatement {
  ...
}
```

Zielprogramm

```
class PluginComponent {
  ...
}
```

Übersetzer

```
class BagStatement {
  ...
}
```

Bibliothek

Dabei ist zu beachten, dass die angegebenen Zielprogramme in dieser Form nicht erstellt werden und nur der Veranschaulichung der Übersetzung dienen. Stattdessen werden nur Repräsentationen solcher Scala-Programme während der Übersetzung erzeugt, die durch den Scala-Übersetzer in die eigentliche Zielsprache überführt werden. Diese Repräsentationen in den Datenstrukturen des Scala-Übersetzers sind allerdings mit den angegebenen Zielprogrammen äquivalent.

2. Scala

Die Programmiersprache Scala wird von Martin Odersky und seiner Arbeitsgruppe am EPFL in Lausanne entwickelt [4]. Sie lässt sich sowohl für die *Java Virtual Machine* als auch in die *Common Intermediate Language*[1] übersetzen, wobei der Schwerpunkt allerdings in der Java VM liegt. Scala ist dabei interoperabel mit der jeweiligen Sprache der virtuellen Maschine: für Java bzw. .NET geschriebene Bibliotheken können in Scala direkt verwendet werden.

2.1. Funktionen höherer Ordnung

Scala wird als *objektfunktionale* Sprache verstanden. In ihr werden die objektorientierten Konzepte um Elemente funktionaler Sprachen erweitert. In funktionalen Programmiersprachen können Funktionen als *first-class citizens*, d.h. wie herkömmliche Datentypen als Übergabeparameter und Rückgabetypen von Funktionen werden. In Scala als einer auf dem objektorientierten Paradigma aufbauenden Sprache sind Funktionen daher Objekte.

Dazu sind in der Scala-Bibliothek Klassen für Funktionen nach der Anzahl ihrer Argumente definiert.

```
1 abstract class Function1[-T, +R] {
2   def apply(v: T): R
3
4 abstract class Function2[-T1,-T2, +R] {
5   def apply(v1: T1, v2: T2): R
6 }
7 ...
```

Klassen für Funktionen

[1] http://msdn.microsoft.com/de-de/library/bb979570.aspx

Konkrete Funktionen lassen sich als Implementierungen einer Funktionsklasse definieren, was häufig als anonyme Unterklasse geschieht.

```
1 val succ = new Function1[Int, Int] {
2   def apply(x: Int) = x + 1
3 }
4
5 succ.apply(0)
```

Beispiel 2.1: Funktionsinstanz

Die Funktion wird dabei über die Methode `apply` auf ihre Argumente (in **Beispiel 2.1** ihr einzelnes Argument) angewendet. Die Definition und Applikation von Funktionen wird durch syntaktischen Zucker stark vereinfacht.

```
1 val succ = (x: Int) => x + 1
2
3 succ(0)
```

Beispiel 2.2: Anonyme Funktion

Über einen Lambda-Ausdruck in der ersten Zeile von **Beispiel 2.2** wird eine Funktion erstellt, die zu der in den Zeilen 1 - 3 definierten Funktion aus **Beispiel 2.1** äquivalent ist. Um die Anwendung solchen Funktionen dem Aufruf von Methoden weiter anzugleichen, ist auch der Aufruf von `apply` in der Regel optional.

In gängigen funktionalen Programmiersprachen sind Funktionen höherer Ordnung wie `map`, `foldRight` oder `filter` über Listen definiert. In Scala werden diese Funktionen nicht über Listen sondern als Methoden in der Klasse `List` definiert. Das erlaubt die Definition analoger Funktionen mit denselben Namen auch für andere Datenbehälter wie `Set`, `Array` oder auch `Option`.

2.2. Traits

Über `traits`, die eine Erweiterung von `interfaces` in Java darstellen, ist in Scala eine Form von Mehrfachvererbung möglich. In `traits` können im Gegensatz zu `interfaces` auch Instanzvariablen deklariert und für Methodensignaturen optional Implementierungen angegeben werden.

```
1 trait T {
2   def aSignature(x: Int): Double
3   var y: Int = 0
4   def anImplementation(x: Int) = x * y
5 }
```

Beispiel 2.3: Definition eines `traits`

Wird ein `trait` wie `T` in **Beispiel 2.3** über das Schlüsselwort `with` in eine Klasse eingebunden, die wie `C` in **Beispiel 2.4** nicht als `abstract` deklariert wurde, müssen die Methoden des `traits` ohne Implementierung von der Klasse bereitgestellt werden. Bereits im `trait` implementierte Methoden können direkt verwendet werden.

```
1 class C extends S with T {
2   def aSignature(x: Int) = {
3     y = 2 * x
4     anImplementation(x)
5   }
6 }
```

Beispiel 2.4: Einbinden eines `traits`

Falls alle Methoden eines `traits` bereits vom diesem implementiert werden, kann dieser direkt beim Aufruf des Konstruktors einer Klasse eingebunden werden:

```
new C with T
```

Dabei wird vom Übersetzer eine anonyme Unterklasse erstellt, die von `C` erbt und `T` einbindet. Das ist allerdings im Fall von **Beispiel 2.3** nicht möglich, da in `T` die Methode `aSignature` nicht implementiert ist.

Eine Klasse kann von mehreren `traits`, aber nur von einer Oberklasse erben. Daher ist es auch nicht möglich, einen Konstruktor für einen `trait` anzugeben.

Ein bekanntes Problem der Mehrfachvererbung ist das Erben von mehreren `traits`, die jeweils eine Methode derselben Signatur implementieren.

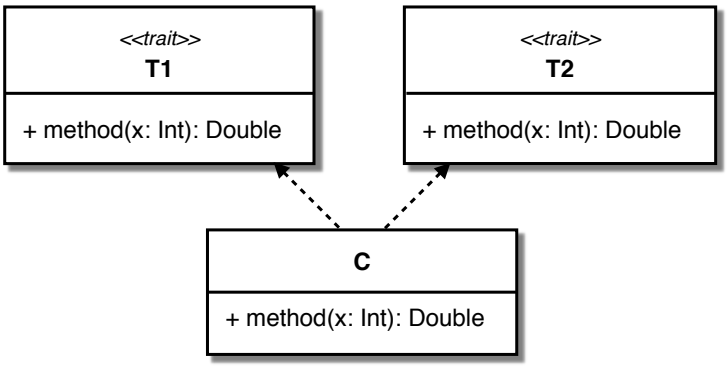


Abbildung 2.1.: Mehrdeutigkeit bei Mehrfachvererbung

Im Fall einer Hierarchie wie in **Abbildung 2.1** muss in der Klasse, die beide `traits` einbindet, die betroffene Methode überschrieben werden.

```
1 class C extends T1 with T2 {
2   override def method(x: Int) = {
3     super[T1].method(x) + super[T2].method(x)
4   }
5 }
```

Beispiel 2.5: Auflösen der Mehrdeutigkeit

Durch eine qualifizierte Variante von `super` kann dann bei Bedarf die Methode eines bestimmten `traits` aufgerufen werden. Durch diesen Ansatz ist der Programmierer dazu gezwungen, Mehrdeutigkeiten aufzulösen, kann aber dennoch einfach eine der Varianten auswählen.

In der Klassenhierarchie von **Abbildung 2.2** erbt die Klasse `C` von den `traits` `T1` und `T2`. Der `trait T2` erbt dabei ebenfalls von `T1` und überschreibt dessen Methode `method2`.

In dieser Konstellation erbt `C` primär von `T1`. Dadurch dass auch `T2` eingebunden wird, gilt in `C` allerdings die überschriebene Implementierung von `method2` aus `T2`. Es lassen sich also über das Einbinden von `traits` auch Methoden anderer eingebundener `traits` überschreiben. Diese Technik wird in Kapitel 7 verwendet, um die Methoden von Parser-Komponenten über einen `trait` mit anbieterspezifischen Parser-Definitionen zu überschreiben.

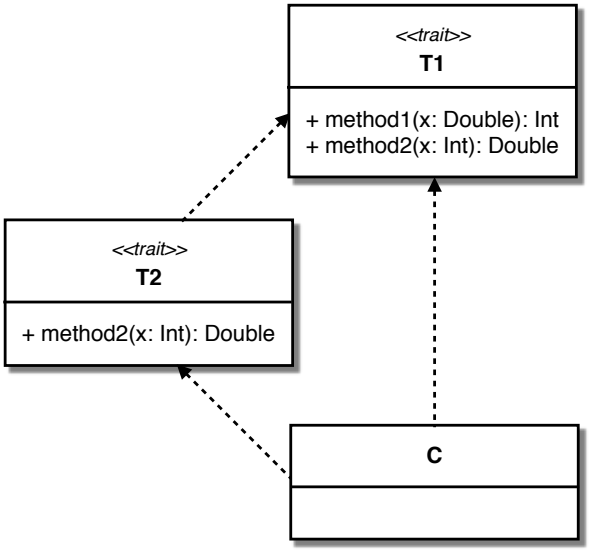


Abbildung 2.2.: Überschreiben von Methoden durch `traits`

2.3. Musterabgleich

Ein Ziel bei der Entwicklung von Scala ist die vereinfachte Umsetzung nebenläufiger Programme, insbesondere auf Rechnern mit mehreren Prozessoren. Um Nebenläufigkeitsanomalien zu vermeiden, eignen sich besonders gut referenziell transparente, nach ihrer Konstruktion unveränderbare Daten. Dazu sind in Scala `case`-Klassen definiert.

```
1 case class Role(number: Int, name: String, actor: String)
2
3 val two = Role(2, "Curtis", "Ian McKellen")
4 val six  = Role(6, "Michael", "Jim Caviezel")
```

Beispiel 2.6: `case`-Klasse

Die Konstruktorargumente können nach dem Aufruf des Konstruktors für eine Instanz der `case`-Klasse nicht mehr verändert werden. Für jedes Argument wird eine Methode für den lesenden Zugriff generiert (z.B. über `two.firstNames`). Auf das Schlüsselwort `new` kann bei dem Aufruf des Konstruktors einer `case`-Klasse verzichtet werden.

Eine spezielle Form solcher Klassen sind Tupel, die in Scala 2.9 bis zur Stelligkeit 22 definiert sind.

```
case class Tuple2[+T1, +T2](_1: T1, _2: T2)
case class Tuple3[+T1, +T2, +T3](_1: T1, _2: T2, _3: T3)
...
case class Tuple22[+T1, +T2, ... +T22](_1: T1, _2: T2, ..., _22: T22)
```

Klassen für Tupel

Für die Erzeugung von Tupelinstanzen ist eine übliche Notation als syntaktischer Zucker definiert. So lässt sich über `(2, "Kapitel 2: Scala")` eine Instanz des Typs `Tuple2[Int, String]` (oder auch kurz `(Int, String)`) erzeugen.

Scala enthält ein Konstrukt für den Musterabgleich [3], über die ein Ausdruck mit Werten, Typen und im Falle von `case`-Klassen auch mit den Werten oder Typen der einzelnen Komponenten abgeglichen werden kann.

```
1 any match {
2   case 1 => println("Got number one")
3   case date: java.util.Date => println("Got the date " + date)
4   case x: Int if x > 10 => println("Got a number > 10")
5   case ("load user", id: Int) => println("Load request for user " + id)
6   case r@Role(2, _, _) => println("Two's real name is " + r.name)
7   case _ => println("Got something else")
8 }
```

Beispiel 2.7: Musterabgleich

Wird wie in den Zeilen 3 und 4 von **Beispiel 2.7** der Ausdruck `any` mit einem Typ abgeglichen, so handelt es sich bei `date` bzw. `x` zwar um denselben Wert. Dieser kann allerdings typsicher als `java.util.Date` bzw. `Int` auf der rechten Seite der Regel verwendet werden.

In den Zeilen 5 und 6 findet der Abgleich auch auf die Komponenten von `case`-Klassen statt. Ein Abgleich auf Komponenten ließe sich auch in einander verschachteln. Über den Unterstrich wird jeder Wert und damit auch jeder Typ akzeptiert.

2.4. Typsystem

Wie Java und C# unterstützt Scala parametrisierte Klassen und Methoden.

```
1  abstract class Option[+A] {
2   def get: A
3   ...
4  }
5  case class Some[+A](x: A) extends Option[A] {
6    def isEmpty = false
7    def get = x
8  }
9  case object None extends Option[Nothing] {
10   def isEmpty = true
11   def get = throw new NoSuchElementException("None.get")
12 }
```

Beispiel 2.8: Parametrisierte Klasse `Option`

Typvariablen werden allerdings in eckigen Klammern angegeben, da spitze Klammern für die XML-Notation reserviert wird. Der Typ `Nothing`, der in **Beispiel 2.8** für die Definition von `None` verwendet wird, steht stellvertretend für jeden Typ. Damit ist `None` absolut generisch und benötigt prinzipiell nur eine Instanz. Dazu kennt Scala neben Klassen auch eindeutige Objekte, die wie Klassen implementiert werden, allerdings über das Schlüsselwort `object` eingeleitet werden. Der Zugriff auf ein solches Objekt erfolgt statisch über den Namen.

In Scala ist wie in C# die Typangabe bei einer Variable, der ein initialer Wert zugewiesen wird, und bei Methodendefinitionen mit Implementierung optional. Der Typ der Variable oder Funktion wird von dem Initialwert bzw. dem Rückgabewert abgeleitet. Die Typinferenz von Typvariablen, die in Java nur für parametrisierte Methoden unterstützt wird, ist in Scala auch auf Konstruktoren erweitert. So ist es beispielsweise möglich, statt `Some[Int](4)` oder `Some[String]("4")` direkt `Some(4)` bzw. `Some("4")` zu schreiben.

Manifeste

Für die Metaprogrammierung zur Laufzeit und um das aus der Software-Technik bekannte Konzept der losen Kopplung zu realisieren, wird in der Java-Programmierung oft Reflexion eingesetzt. Über die *Java Reflection API* lässt sich ein Objekt zur Laufzeit analysieren. Dazu stellt jedes Objekt eine Methode `getClass` bereit, welche die Repräsentation der Objektklasse liefert.

Die Repräsentation vom Typ `java.lang.Class` stellt wiederum Methoden wie `getMethods` bereit, die es ermöglichen, die Methoden einer Klasse zu inspizieren oder sogar für Objekte der Klasse auszuführen.

Die Reflexion beschränkt sich allerdings auf die Klasseninformationen und enthält keine vollständigen Typen. Die Typen, an die Typvariablen zur Übersetzungszeit gebunden sind, werden in der Übersetzung eliminiert und an den notwendigen Stellen durch Typumwandlungen ersetzt. Daher ist über ein Objekt, das zur Übersetzungszeit den Typ `List[Int]` hat, zur Laufzeit nur die Klasse `List` bekannt.

Dieses Problem wird in Scala teilweise durch Typmanifeste gelöst. Manifeste sind ein weiterführendes Konzept des Typsystems von Scala, sind allerdings für die in Kapitel 5 vorgestellte Bibliothek relevant. Dort werden mit Hilfe von Manifesten zur Laufzeit generische Projektionen zum Extrahieren von Werten aus den Ergebnissen von Abfragen erzeugt.

Ein Manifest vom Typ `scala.reflect.Manifest[T]` enthält neben der Klasse des Typs `T` auch die Manifeste der in `T` gebundenen Typvariablen. Wegen der Eliminierung der Typvariablen in der Übersetzung, kann ein Manifest nicht zur Laufzeit beispielsweise über `getManifest` bezogen werden. Stattdessen muss das Manifest zu einem bestimmten Objekt explizit zur Übersetzungszeit angefordert werden. Das wird in der Regel durch ein implizites Argument erreicht.

```
1 class ServiceRegistry {
2   private val registeredServices = new HashMap[Manifest[_], AnyRef]
3   def register[T](impl: T)(implicit service: Manifest[T]) {
4     println("Registering " + impl + " for service " + service)
5     registeredServices.put(manifest, impl)
6   }
```

Beispiel 2.9: vom Übersetzer erzeugtes Manifest

Dazu wird eine parametrisierte Methode mit einer weiteren Parameterliste versehen, die einen Parameter vom Typ des Manifests zur Typvariable enthält. Der Parameter wird als `implicit` markiert, so dass das Argument beim Aufruf der Methode optional ist und vom Übersetzer eingesetzt wird. So lässt sich in der in **Beispiel 2.9** implementierten Registrierung ein Dienst über den folgenden Aufruf anmelden:

```
serviceRegistry.register[GenericService[Int]](new IntServiceImpl)
```

Der Typ für Variable `T` ist optional, würde aber vom Übersetzer an den Typ `IntServiceImpl` gebunden werden. Das ist in diesem Fall der Registrierung von Diensten, die Schnittstellen auf eine Implementierung abbilden soll, gerade nicht gewünscht.

Ein Spezialfall sind Methoden, die ausschließlich eine Parameterliste mit impliziten Manifesten enthalten.

```
7   def lookup[T](implicit manifest: Manifest[T]): T = {
8     registeredServices(manifest).asInstanceOf[T]
9   }
10 }
```

Beispiel 2.10: Nur vom Übersetzer erzeugtes Argument

Eine solche Methode wird dann nur mit einer Typangabe für die Typvariablen der Methode aufgerufen:

```
val impl = serviceRegistry.lookup[GenericService[Int]]
```

Implizite Konversionen

Das Schlüsselwort `implicit` wird auch verwendet, um Konversionen zu implementieren. Seit *Java 5* ist es möglich an Stellen, an denen primitive Ganzzahlen vom Typ `int` erwartet werden, auch Objekte vom Typ `java.lang.Integer` zu verwenden und umgekehrt. Der Java-Übersetzer fügt dabei an solchen Stellen Funktionsaufrufe ein, die den Wert bzw. das Objekt konvertieren.

Dieses als *Autoboxing* bezeichnete Technik wird im Typsystem von Scala durch implizite Konversionen formalisiert und dem Programmierer zur Verfügung gestellt. Wird an einer Stelle ein Ausdruck vom `T'` erwartet, ein Ausdruck vom Typ `T` verwendet, fügt der Übersetzer einen Aufruf von `f` ein, falls `f` eine sichtbare als `implicit` markierte Funktion vom Typ `T => T'` ist.

Solche Konversionen lassen sich auch verwenden, um Objekte nach dem *Decorator Pattern* [5] zu ummanteln und damit die jeweilige Klasse quasi um Methoden zu erweitern. In Kapitel 6 wird dieses Verfahren verwendet, um Parser-Kombinatoren um einen Operator zu erweitern.

3. Erweiterung des Scala-Übersetzers

Der Scala-Übersetzer kompiliert einerseits in das Format der *Java Virtual Machine* aber auch in die *Common Intermediate Language* für .NET. Der Übersetzer ist in einzelne Phasen unterteilt.

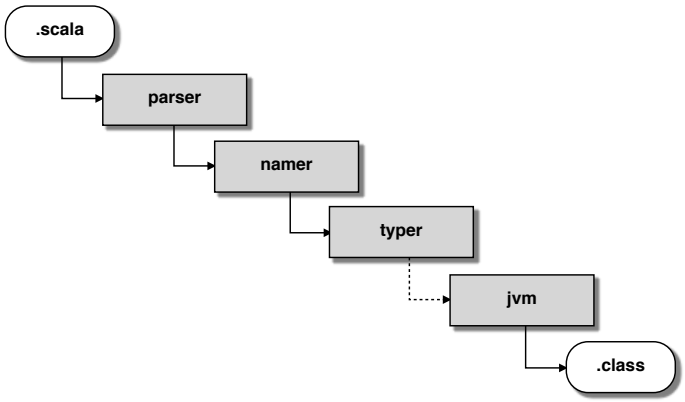


Abbildung 3.1.: Phasen des Scala-Übersetzers

Über ein Plug-in ist es möglich, eine oder mehrere Phasen zwischen den herkömmlichen Phasen des Scala-Übersetzers einzufügen.

3.1. Plug-in-Architektur

Ein kleineres Plug-in lässt sich kompakt in einer Quelldatei definieren. Um eine Erweiterung zu erstellen, ist zunächst eine Unterklasse von `nsc.plugins.Plugin` notwendig.

```
1  package extension
2
3  import scala.tools.nsc
4  import nsc.Global
5  import nsc.Phase
6  import nsc.plugins.Plugin
7  import nsc.plugins.PluginComponent
8
9  class Extension(val global: Global) extends Plugin {
10   import global._
11
12   val name = "extension"
13   val description = "extends the scala language"
14   val components = List[PluginComponent](Component)
```

Beispiel 3.1: Unterklasse von `Plugin`

Der Konstruktor der Klasse nimmt eine Repräsentation des Übersetzungsprozesses vom Typ `Global` entgegen. Die Klasse selber definiert nur einen Namen, eine Beschreibung und eine Liste von Plug-in-Komponenten. Jede Komponente repräsentiert eine zusätzliche Phase für die Übersetzung und wird als eingebettetes *Singleton*-Objekt [5] definiert.

```
15   private object Component extends PluginComponent {
16     val global: Extension.this.global.type = Extension.this.global
17     val phaseName = Extension.this.name
18     val runsAfter = "parser"
19     def newPhase(_prev: Phase) = new Extension(_prev)
```

Beispiel 3.2: Objekt vom Typ `PluginComponent`

In jeder Komponente wird die Übersetzungsrepräsentation des umgebenden Plug-ins übernommen und ein bei mehreren Komponenten ggf. vom Plug-in abweichender Name gesetzt. Hier wird der Name des Plug-ins übernommen, da nur eine Komponente definiert wird. In der Komponente wird über den Wert `runsAfter` die herkömmliche Phase des Übersetzers spezifiziert, nach der die Phase der Komponente eingefügt wird. Dabei wird nicht gewährleistet, dass die Phase der Komponente unmittelbar nach der angegebenen Phase des Scala-Übersetzers eingefügt wird, was insbesondere bei mehreren aktiven Plug-ins nicht unbedingt der Fall ist.

Über die Methode `newPhase` wird für eine übergebene vorherige Phase die eigentliche Phase, um die der Scala-Übersetzer mit dieser Komponente erweitert wird, erstellt.

```
20     class ExtensionPhase(prev: Phase) extends StdPhase(prev) {
21        def apply(unit: CompilationUnit) {
22           unit.body // analyze or manipulate the AST
23        }
24     }
25   }
26 }
```

Beispiel 3.3: Unterklasse von `Phase`

Die Phase wird als Unterklasse von `StdPhase` in die Komponente eingebettet. Die Methode `apply` enthält die eigentlichen Aktionen der Phase. Dort kann die Kompilierungseinheit des Übersetzungsprozesses analysiert werden und über `unit.body` auf den abstrakten Syntaxbaum zugegriffen werden. Eine spezielle Form sind transformierende Phasen, die gezielt Teile des abstrakten Syntaxbaums manipulieren. Eine solche Phase wird im folgenden Abschnitt erläutert.

Damit die Erweiterung vom Übersetzer berücksichtigt werden kann, muss sie als Java Archiv (JAR) mit einem *Plug-in-Deskriptor* exportiert werden. Der Deskriptor ist eine XML-Datei `scala-plugin.xml`, die den Namen des Plug-ins und den qualifizierten Namen der von `Plugin` abgeleiteten Klasse enthält.

```
1 <plugin>
2   <name>extension</name>
3   <classname>extension.Extension</classname>
4 </plugin>
```

Beispiel 3.4: Plug-in-Deskriptor

Um die Erweiterung in der Übersetzung zu aktivieren, muss der Name des Java Archivs als Argument übergeben werden:

```
~$ scalac -Xplugin:extension.jar SourceFile.scala
```

3.2. Beispielerweiterung: Assoziationen

Der Quellcode des in dieser Arbeit vorgestellten Plug-ins zur Transformation von eingebetteten SQL-Anfragen ist in weiten Teilen zu komplex um vollständig aufgeführt zu werden. Um aufzuzeigen, wie eine Transformation des Syntaxbaums für den Scala-Übersetzer realisiert werden kann, wird in diesem Abschnitt ein kleineres Plug-in vorgestellt, das Scala um Assoziationstypen erweitert.

Anforderungen

Über die *Unified Modeling Language*[1] lassen sich in Klassendiagrammen Assoziationen zwischen Klassen modellieren. Dabei gilt die Annahme, dass die Assoziationen in einem UML-Diagramm wie in **Abbildung 3.2** über Referenzen auf beiden Seiten implementiert werden. So lässt die bidirektionale *1 : n*-Beziehung zwischen `Role` und `Right` auf der *n : 1*-Seite `Right` durch eine einfache Variable vom Typ `Role` realisieren, während die *1 : n*-Seite durch eine mit `Right` parametrisierte Listen- oder Mengendatenstruktur in der Klasse `Role` realisiert wird.

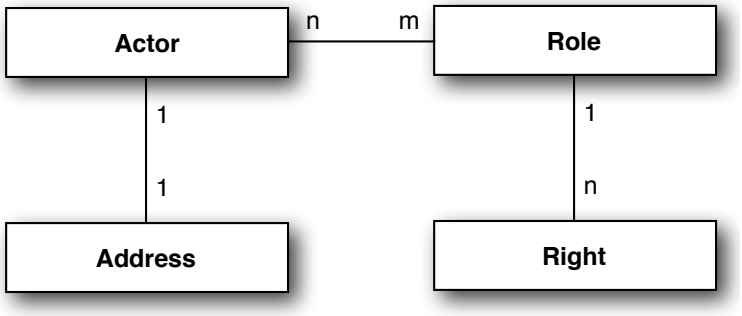


Abbildung 3.2.: Assoziationen in einem UML-Diagramm

Allerdings wird durch eine bidirektionale Assoziation eine Restriktion gefordert, die mit Referenzen allein nicht sicher gestellt ist. Ist ein Objekt `right` vom Typ `Right` in der Liste/Menge der Rechte in `role: Role` enthalten, muss es selbst auch `role` referenzieren.

Obwohl sie häufig benötigt werden, stellen objektorientierte Sprachen in der Regel keine direkte Unterstützung für Assoziationen bereit. Über ein Übersetzer-

[1] http://www.omg.org/spec/UML/

Plug-in soll Scala nun um bidirektionale Assoziationen der folgenden Form erweitert werden.

```
1 class Role {
2   val rights: OneToMany[Right -> role] // 1 : n
3   val users: ManyToMany[User -> roles] // n : m
4 }
5 class Right {
6   var role: ManyToOne[Role -> rights] // n : 1
7 }
```

Beispiel 3.5: Typen für Assoziationen

Dabei wird für eine *1 : 1*- oder *n : 1*-Seite eine Variable vom Typ `OneToOne` bzw. `ManyToOne` verwendet, die mit der gegenüberliegenden Klasse parametrisiert wird. Zusätzlich wird nach dem Pfeil `->` der Name des Felds in der gegenüberliegenden Klasse angegeben, das die Referenz zurück auf das Objekt darstellt bzw. enthält. Wird also die Variable `role` in `right: Right` gesetzt, muss `right` in `role.rights` hinzugefügt werden. Für eine *1:n*- oder *m:n*-Seite wird analog ein Wert vom Typ `OneToMany` bzw. `ManyToMany` verwendet.

```
1 val role1 = new Role
2 val right = new Right
3 role1.rights += right
4
5 val role2 = new Role
6 right.role = Some(role2)
```

Beispiel 3.6: Verwendung von Assoziationen

Ein Feld vom Typ `OneToOne[T -> ...]` oder `ManyToOne[T -> ...]` ist für den benutzenden Programmierer wie eine Eigenschaft vom Typ `Option[T]` zu verwenden. Felder mit den Assoziationstypen `OneToMany[T -> ...]` und `ManyToMany[T -> ...]` sind nach außen als Menge von `T` verwendbar.

Beim Hinzufügen von `right` in die Menge der Rechte in `role1` in Zeile 3 im **Beispiel 3.6** wird automatisch über `right.role = Some(role1)` die gegenüberliegende Referenz angepasst. Wird dann in Zeile 8 die Rolle von `right` gewechselt, wird das Recht automatisch aus der Menge in `role1` entfernt und in die Menge in `role2` eingefügt. Genau genommen handelt es sich bei der Umsetzung mit `Option` eigentlich um *0,1 : 0,1*- bzw. *n : 0,1*-Assoziationen, da ein Wert gefordert ist und `None` daher einen ungültigen Zustand beschreibt.

Allerdings lässt sich diese Restriktion im initialen Zustand und beim Wechsel von Referenzen schlecht sicherstellen.

Umsetzung

Damit beim Ändern einer Seite einer Assoziation automatisch die gegenüberliegende Seite angepasst wird, muss dieses Verhalten bei der Übersetzung eingefügt werden.

```
1  class Right {
2    private var role_$association: Option[Role]
3    def role = role_$association
4    def role_=(newValue: Option[Role]) {
5      if (newValue != role_$association) { // prevent an infinite loop
6        val oldValue = role_$association
7        role_$association = newValue
8        // remove association from the old opposite
9        oldValue.foreach(oldOpposite => oldOpposite.rights -= this)
10       // add association from to the new opposite
11       newValue.foreach(newOpposite => newOpposite.rights += this)
12     }
13   }
14 }
```

Beispiel 3.7: Übersetzung von `ManyToOne`

Beispiel 3.7 zeigt die Übersetzung einer *n : 1*-Seite. Die Methode `role_=` mit dem Suffix `_=` erlaubt dabei Schreibzugriff auf `role` als Eigenschaft. So kann die Referenz über `right.role = Some(role)` gesetzt werden.

Eine *1 : 1*-Seite wird analog übersetzt, nur dass die Referenzen der gegenüberliegenden Objekte auf `None` bzw. `Some(this)` gesetzt werden. Bei *1 : n*- und *m : n*-Seiten müssen anonyme Unterklassen von `scala.collection.mutable.Set` kompiliert werden, deren Methoden `+=` und `-=` überschrieben werden, um die gegenüberliegende Seite analog zur Implementierung von *1 : 1*- und *n : 1*-Seiten zu anzupassen.

Die Übersetzung wird als Erweiterung für den Scala-Übersetzer implementiert, die eine zusätzliche Phase bereitstellt. Die Phase transformiert direkt nach der Syntaxanalyse den abstrakten Syntaxbaum, um Assoziationen zu erkennen und zu übersetzen.

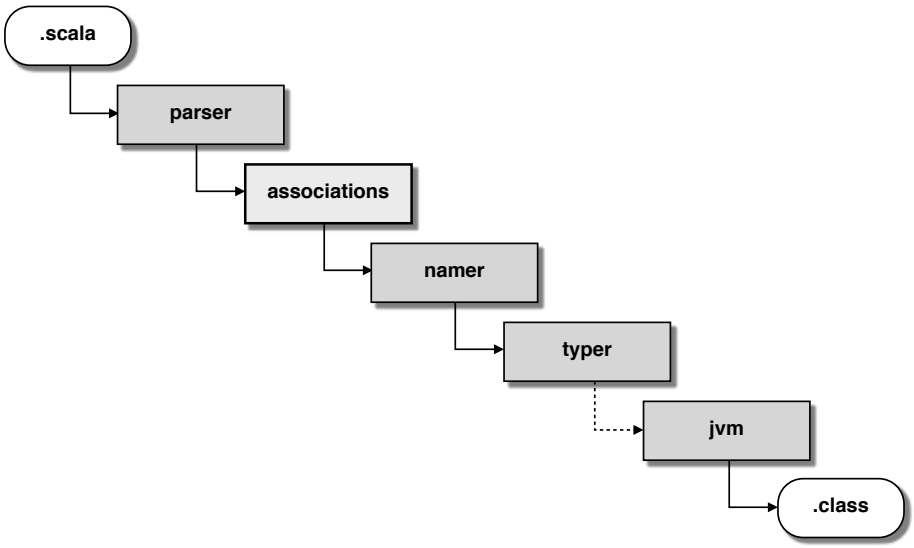


Abbildung 3.3.: Integration des Plug-ins

Dazu wird zunächst ein Plug-in mit einer Plug-in-Komponente benötigt. Da eine Phase bereitgestellt werden soll, die den abstrakten Syntaxbaum eines Programms transformiert, wird die Komponente um den `trait Transform` erweitert.

```
1  class AssociationsPlugin(val global: Global) extends Plugin {
2   import global._
3
4   val name = "Associations Plug-in"
5   val description = "transforms associations into regular Scala definitions"
6   val components = List[PluginComponent](Component)
7
8   private object Component extends PluginComponent with Transform {
9    val global: AssociationsPlugin.this.global.type = AssociationsPlugin.this.global
10   val runsAfter = List[String]("parser")
11   override val runsBefore = List[String]("namer")
12   val phaseName = AssociationsPlugin.this.name
```

Beispiel 3.8: Plug-in und -Komponente

Durch das Erben von `Transform` wird die Methode `newPhase` in der Plug-in-Komponente schon bereit gestellt. Stattdessen muss `newTransformer` implementiert werden. `Transformer` sind spezielle Phasen, die das Modifizieren des abstrakten Syntaxbaums erleichtern. Die Basisklasse stellt für die Elemente des Syntaxbaums Methoden bereit, die per Standardimplementierung den Baum nur kopieren. Durch das gezielte Überschreiben der Methoden können Teile des abstrakten Syntaxbaums transformiert werden.

```
13  def newTransformer(unit: global.CompilationUnit) = new AssocTransformer(unit)
14  class AssocTransformer(unit: global.CompilationUnit) extends global.Transformer {
15
16   import global._
17
18   override def transformStat(stat: Tree): List[Tree] = {
19    stat match {
20     case ClassDef(_, name, _, _) => withClassName(Some(name), List(transform(stat)))
21     case _: ModuleDef => withClassName(None, List(transform(stat)))
```

Beispiel 3.9: Transformieren von Anweisungen

Die Transformation beschränkt sich in diesem Plug-in auf Anweisungen, die durch das Überschreiben von `transformStat` modifiziert werden können. Assoziationen dürfen nur innerhalb von Klassen definiert werden. Die Definition in einem `object` macht keinen Sinn, da für sie kein herkömmlicher Typ[2] existiert. Daher wird die Transformation von Klassen- und Objektdefinitionen (`ModuleDef`) abgefangen, um den aktuellen Klassennamen (oder `None`) zu merken. Die eigentliche Transformation wird an die Oberklasse `Transformer` übergeben, welche die Definition kopiert und dabei den Unterbaum rekursiv transformiert. Klassen- und Objektdefinitionen gelten als Anweisungen, da sie im Rumpf von Modulen angegeben werden, in denen auch herkömmliche Anweisungen stehen dürfen.

Auch Instanzvariablen können als Anweisungen identifiziert werden. In Scala ist der Rumpf einer Klasse der Standardkonstruktor, daher werden Instanzvariablen und Methoden im Konstruktor ihrer Klasse angegeben.

```
22     case valDef @ ValDef(modifiers, termName, tpt @ AppliedTypeTree(Ident(name), args),
           expr) if associationTypeNames.contains(name.toString) => {
```

Beispiel 3.10: Identifizieren von Assoziationen

Die Phase soll nur Instanzvariablen mit dem Typ einer Assoziation transformieren. Alle herkömmlichen Variablen sollen kopiert werden. Daher wird bei einer Instanzvariable zunächst per Musterabgleich auf einen Typ mit dem Namen *OneToOne*, *OneToMany*, *ManyToOne* oder *ManyToMany* geprüft. Dieser Typ muss parametrisiert sein.

Wird eine Instanzvariable anhand des Namens an der Wurzel ihres Typs als vom Programmierer beabsichtigte Assoziation erkannt, muss der vollständige Typ validiert werden.

[2] *Singletons* haben zwar einen speziellen Typ, sind aber immer einzig mögliche Instanz.

```
23      args match {
24       case List(AppliedTypeTree(Ident(operator),
25                 List(otherTpt: RefTree, mappedBy @ Ident(_))))
26                 if operator.toString == "$minus$greater" => {
27         if (!currentClassName.isDefined) {
28          unit.error(valDef.pos, "Associations may only be defined in classes or traits.")
29          List(EmptyTree)
30         } else {
```

Beispiel 3.11: Validierung des Typs

Der Typ einer Assoziation ist für die Syntaxanalyse des Scala-Übersetzers parametrisiert mit der Anwendung des Typoperators `->` (der von der Syntaxanalyse in `$minus$greater` überführt wurde). Der linke Operand `otherTpt` ist der Typ der gegenüberliegenden Klasse. Der zweite Operand ist eigentlich kein Typ, sondern der Name der Instanzvariable in der gegenüberliegenden Klasse und muss damit ein einfaches Symbol vom Typ `Ident` sein, das als `mappedBy` benannt wird. Die Assoziation muss Instanzvariable einer Klasse sein, was über den in **Beispiel 3.9** gesetzten Klassennamen überprüft wird.

Der vollständige Typ hätte prinzipiell auch im Musterabgleich in **Beispiel 3.10** überprüft werden können. Das hätte allerdings zur Folge, dass falsch angewendete Assoziationen nicht transformiert werden und zu unerwarteten Fehlermeldungen führen. Wird der Name eine Assoziation verwendet, wird angenommen, dass eine solche beabsichtigt wurde und bei falscher Verwendung ein Typfehler auftreten soll (siehe dazu **Beispiel 3.12**, Zeilen 47 - 49).

Nach erfolgreicher Validierung des Typs, muss die Instanzvariable überprüft werden.

```
31            val initVal = if (!modifiers.hasFlag(Flags.DEFERRED)) Some(expr) else None
32            name.toString match {
33             case "OneToOne" if validate_ToOne(valDef) =>
34                                compileOneToOne(valDef, otherTpt, mappedBy, initVal)
35             case "OneToMany" if validate_ToMany(valDef) =>
36                                 compileOneToMany(valDef, otherTpt, mappedBy)
37             case "ManyToOne" if validate_ToOne(valDef) =>
38                                 compileManyToOne(valDef, otherTpt, mappedBy, initVal)
39             case "ManyToMany" if validate_ToMany(valDef) =
40                                  compileManyToMany(valDef, otherTpt, mappedBy)
41             case _ => List(EmptyTree) // not compilable
42            }
43           }
44          }
45          case _ => {
46           // wrong type
47           unit.incompleteInputError(tpt.pos, "Association type error:  %s has to be " +
```

```
48              "of the form %s[OtherClass -> propertyMappingThisClass].".format(name, name))
49            List(EmptyTree)
50           }
51          }
52         }
53         case _ => List(super.transform(stat))
54        }
55       }
```

Beispiel 3.12: Transformieren der Assoziationen

Für eine *1 : 1*- oder *n : 1*-Seite muss eine veränderbare Instanzvariable (`var`) definiert sein, die mit einem Initialwert belegt werden kann. Für eine *1 : n*- oder *m : n*-Seite ist wiederum eine unveränderliche Instanzvariable (`val`) erforderlich. Dieser Wert wird mit der Instanz einer anonymen Unterklasse von `Set` belegt und kann daher keinen Initialwert erhalten.

Für fehlerhafte Instanzvariablen werden durch die Validierungsmethoden Fehler ausgelöst. Erfolgreich validierte Assoziationen werden übersetzt. Alle anderen Instanzvariablen und Anweisungen werden durch die Oberklasse `Transform` kopiert.

Im Folgenden wird exemplarisch die vereinfachte Übersetzung von *n : 1*-Seiten nach dem Schema aus **Beispiel 3.7** gezeigt. Bei der Umsetzung lassen sich die Gemeinsamkeiten von *n : 1*- und *1 : 1*-Seiten extrahieren. Wird die Übersetzung von *n : 1*-Seiten alleine betrachtet, ist sie in der hier angegebenen Form besser nachzuvollziehen.

```
56    private def compileManyToOne(varDef: ValDef, otherTpt: RefTree, mappedByType: Ident,
          initialValue: Option[Tree]) {
57      val valType = AppliedTypeTree(scalaSelect(typeName("Option")), List(otherTpt))
58      val varName = varDef.name.toString + "_$association"
59      val varMods = stdModifiers(Flags.PRIVATE | Flags.MUTABLE)
60      val variable = ValDef(varMods, varName, valType, scalaSelect("None": TermName))
```

Beispiel 3.13: Variable für Referenz

Zunächst wird die private Instanzvariable erzeugt, welche die Referenz auf das gegenüberliegende Objekt hält. Ihr Typ ist daher `Option` parametrisiert mit der gegenüberliegenden Klasse. Der Name der ursprünglichen Variable wird um `_$association` erweitert. Die Variable wird nicht mit dem optionalen Initialwert, sondern mit `None` belegt. Die Zuweisung des Initialwerts erfolgt später über die Methode für den Schreibzugriff, damit das gegenüberliegende Objekt auch im Ausgangszustand informiert wird.

```
61      val methodMods = modifiers(varDef.mods &~ Flags.MUTABLE, false)
62      val getter = DefDef(methodMods, varDef.name, List(), List(), valType, Ident(varName))
```

Beispiel 3.14: Methode für Lesezugriff

Für den Lesezugriff wird eine einfache Methode mit dem Namen der ursprünglichen Variable generiert. Der Typ wird von der in **Beispiel 3.13** erzeugten Variable übernommen, deren Wert auch zurückgeliefert wird.

Wesentlich komplexer ist die Methode für den Schreibzugriff, die den neuen Wert sichert und die bisherige sowie neue gegenüberliegende Seite informiert.

```
63      var paramName = "newValue"
64      val paramMods = modifiersParam(varDef.mods)
65      val param = ValDef(paramMods, paramName, valType, EmptyTree)
66
67      val block = Block(
68       ValDef(Modifiers(0, "", List(), Map()), "oldValue", valType, Ident(varName)),
69       Assign(Ident(varName), Ident(paramName)),
70       // oldValue.foreach(x => x.ys -= this)
71       crossReference("oldValue", crossRemove, thisC),
72       // newValue.foreach(x => x.ys += this)
73       crossReference("newValue", crossAdd, thisC))
74
75      val ifClause = If(Apply(Select(Ident(varName), "$bang$eq"),
76        List(Ident(paramName))), block, Literal(()))
77
78      val setterType = scalaSelect(typeName("Unit"))
79      val setter = DefDef(methodMods, varDef.name + "_$eq", List(), List(List(param)),
            setterType, ifClause)
```

Beispiel 3.15: Methode für Schreibzugriff

In den Zeilen 63 - 65 wird zunächst der Parameter `newValue` generiert. Der Block, in dem der neue Wert gesetzt und die gegenüberliegenden Seiten informiert werden, wird in den Zeilen 67 - 73 erzeugt. Die Anweisungen zum Informieren werden dabei durch die `cross`-Methoden generiert, die auch beim Übersetzen der anderen Assoziationen verwendet werden.

Der Block wird in Zeile 75 in eine Bedingung integriert, die prüft, ob eine Aktualisierung des Werts überhaupt notwendig ist, um Endlosschleifen zu vermeiden. Die Methode enthält die Bedingung und wird mit dem Namen der ursprünglichen Variable versehen, der um `_$eq` erweitert wird, damit die Methode den Schreibzugriff auf die Assoziation ermöglicht.

```
80      val initialAssign = initialValue.map(expr => Assign(Ident(varDef.name), expr))
81      List(variable, getter, setter, initialAssign.getOrElse(EmptyTree))
82    }
```

Beispiel 3.16: Initialer Wert

Als letzte Anweisung wird der Initialwert zugewiesen, wenn dieser vorhanden ist. Dabei wird der ursprüngliche Name der Variable verwendet, damit die in **Beispiel 3.15** erzeugte Methode verwendet wird, um die gegenüberliegende Seite zu informieren.

Die Anweisung für die ursprüngliche Definition der Instanzvariable wird ersetzt durch die neue private Variable, die Methoden für Lese- und Schreibzugriff sowie der optionalen Zuweisung des initialen Werts.

4. Structured Query Language

Anfragen an relationale Datenbanken lassen sich mit der relationalen Algebra modellieren [2].

Die Algebra kennt Projektion π und Selektion σ, um die Spalten und Zeilen einer Relation zu reduzieren. Über das Kreuzprodukt $\times$ lassen sich Relationen kombinieren. Insbesondere bei der Kombination gleicher Relationen ist die Umbenennung ρ wichtig. Als Abkürzung für Kreuzprodukt, Selektion und Projektion ist der in Datenbankabfragen sehr wichtige Verbund $\bowtie$ definiert.

Ferner kennt die relationale Algebra die üblichen Mengenoperationen über Relationen gleichen Typs. Eine Erweiterung der Algebra um die Berechnung transitiver Hüllen bildet die Basis für rekursive Abfragen.

Im folgenden Beispiel werden die Namen der Benutzerrechte eines Benutzers mit Login *cwu* berechnet (das Beispielschema wird im Abschnitt 4.3 über Datenbankschemata auf Seite 43 beschrieben):

$$\pi_{right\_name}(\sigma_{login='cwu'}(actors \bowtie actor\_rights \bowtie \rho_{[right\_name \leftarrow name]}(rights)))$$

Zunächst wird die Spalte `name` in der Tabelle `rights` in `right_name` umbenannt, um eine Namenskollision mit der Relation der Benutzer zu vermeiden. Dann wird der Verbund aus der Tabelle der Benutzer, der Rechte und der passenden Schnitttabelle gebildet. Dabei werden über den Verbundoperator $\bowtie$ nur diejenigen Zeilen aus dem Kreuzprodukt der drei Relationen selektiert, die in den Schlüsseln übereinstimmen. Aus der Ergebnisrelation wird die Spalte `right_name` projiziert.

4.1. SQL als standardisierte Schnittstelle

Die gängigen relationalen Datenbanksysteme bieten eine Schnittstelle über die *Structured Query Language* an. Entgegen des weitläufigen Begriffs der *SQL-Datenbanken* handelt es sich bei diesen Systemen um relationale Datenbanken auf Basis des relationalen Modells, für die SQL als standardisierte Schnittstelle dient.

SQL ist eine Programmiersprache, in der sich Anfragen an relationale Datenbanksysteme formulieren lassen. Daher existieren für die Ausdrücke der relationalen Algebra äquivalente Konstrukte in SQL.

4.2. Abfragen

Das Schlüsselwort `SELECT` leitet Abfragen in SQL ein, steht allerdings nicht für die Selektion, sondern die Projektion:

$$\texttt{SELECT login FROM actors} \stackrel{\wedge}{=} \pi_{login}(actors)$$

Selektiert wird über das Schlüsselwort `WHERE` und einen folgenden booleschen Ausdruck:

$$\texttt{SELECT * FROM actors WHERE login = 'cwu'} \stackrel{\wedge}{=} \sigma_{login='cwu'}(actors)$$

Die Umbenennung ist in SQL über das Schlüsselwort `AS` für Spaltennamen nur in der Projektion, nicht aber im Tabellenausdruck möglich. Allerdings ist die Umbenennung auch auf Relationsnamen definiert, wobei das Schlüsselwort `AS` optional ist. Spaltennamen können mit dem (ggf. umbenannten) Relationsnamen qualifiziert angegeben werden:

$$\texttt{SELECT rs.name AS right\_name FROM rights rs} \stackrel{\wedge}{=} \pi_{rs.right\_name}(\rho_{[right\_name \leftarrow name]}(\rho_{rs}(rights)))$$

Das Kreuzprodukt der relationalen Algebra wird über die Auflistung der Tabellen realisiert:

$$\texttt{SELECT * FROM actors, actor\_rights, rights} \\ \hat{=} actors \times actor\_rights \times rights$$

Für den Verbund stellt SQL mehrere Möglichkeiten bereit. Der natürliche Verbund, der über gleiche Werte in den Spalten desselben Namens definiert ist, lässt sich über `NATURAL JOIN` bilden:

$$\texttt{SELECT * FROM actors NATURAL JOIN actor\_rights} \\ \hat{=} actors \bowtie actor\_rights$$

Beim qualifizierten Verbund werden die Zeilen von zwei Tabellen unter einer vom Programmierer spezifizierten Bedingung verbunden. Dabei gibt es zwei Alternativen. Das Schlüsselwort `JOIN` wird entweder mit der Klausel `USING` mit expliziten Spaltennamen für den Verbund kombiniert:

$$\texttt{SELECT * FROM actors JOIN actor\_rights USING (actor\_id)} \\ \hat{=} actors \bowtie_{actors.actor\_id=actor\_rights.actor\_id} actor\_rights$$

Alternativ kann über die Klausel `ON` eine booleschen Bedingung für den Verbund angegeben werden:

$$\texttt{SELECT * FROM actors a JOIN actor\_rights ar ON (a.actor\_id =} \\ \texttt{ar.actor\_id)} \\ \hat{=} \rho_a(actors) \bowtie_{a.actor\_id=ar.actor\_id} \rho_{ar}(actor\_rights)$$

Auch für die in der relationalen Algebra enthaltenen Operatoren der Mengenlehre gibt es äquivalente Ausdrücke in SQL. Auf Abfragen des gleichen Typs sind die Operatoren `UNION`, `INTERSECT` und `EXCEPT` definiert. Über einen Operator `EXISTS` ist der Existenzquantor definiert. Der Allquantor ist nur über `EXISTS` und Negation realisierbar.

Rekursive Abfragen

Der Operator `UNION` für die Vereinigungsmenge von Relationen gleichen Typs ist die Basis für rekursive Abfragen.

Wie im **Beispiel 1.1** auf Seite 10 gezeigt wurde, werden rekursive Abfragen über die Schlüsselwörter `WITH RECURSIVE` eingeleitet. In der `WITH`-Klausel wird zunächst eine virtuelle Tabelle (im Beispiel: `routes`) aus zwei Abfragen gebildet. Die erste Abfrage bildet den initialen Inhalt der virtuellen Tabelle. Die Abfrage wird über `UNION` mit der zweiten Abfrage vereinigt, in der die virtuelle Tabelle verwendet werden kann.

Die Vereinigungsmenge über `UNION` schließt das Eliminieren von Duplikaten ein. Die virtuelle Tabelle wird iterativ mit dem jeweiligen Ergebnis der zweiten Abfrage vereinigt, bis die Tabelle in einer Iteration nicht erweitert wurde.

Die `WITH`-Klausel enthält eine dritte Abfrage, die das Endergebnis der Iterationen als Tabelle verwenden kann.

4.3. Datenbankschemata

Um Anfragen an Datenbanken zu analysieren und auszuwerten, ist es notwendig, das zugrundeliegende Schema zu definieren. Dazu enthält die *Structured Query Language* als Teilsprache die *Database Description Language* (DDL), über die sich Datenbanktabellen erzeugen lassen.

Tabellen werden über die Anweisung `CREATE TABLE` durch die Angabe eines (ggf. mit Datenbank- und Schemanamen qualifzierten) Tabellennamens und einer Sequenz von Spaltendefinitionen und Restriktionen definiert.

Spalten müssen mit einem Namen und einem Typ angegeben werden, sofern der Typ nicht durch einen Standardwert über die Klausel `DEFAULT` abgeleitet werden kann, was allerdings nicht von allen Anbietern unterstützt wird. Zusätzlich können Spaltenrestriktionen wie Ausschluss von Nullwerten, Eindeutigkeit oder Beziehung zu anderen Tabellen definiert werden. Zusätzlich lassen sich tabellenweite Restriktionen angeben um beispielsweise (Primär-)Schlüssel über mehrere Spalten der Tabelle zu definieren.

```
1  CREATE TABLE scalasql.public.actors (
2    actor_id INTEGER PRIMARY KEY,
3    login VARCHAR(128) NOT NULL UNIQUE,
4    name VARCHAR(128) NOT NULL UNIQUE,
5    image BLOB,
6    salary DECIMAL(15,2) NOT NULL CHECK (salary > 0),
7    parent VARCHAR DEFAULT CURRENT_USER NOT NULL
8  );
9  CREATE TABLE scalasql.public.rights (
10   right_id INTEGER PRIMARY KEY,
11   name CHARACTER VARYING(64) NOT NULL UNIQUE,
12   description CLOB
13 );
14 CREATE TABLE scalasql.public.actor_rights (
15   actor_id INTEGER NOT NULL REFERENCES actors,
16   right_id INTEGER NOT NULL REFERENCES rights,
17   PRIMARY KEY (actor_id, right_id)
18 );
```

Beispiel 4.1: Erzeugung von Tabellen

Auch wenn das Erstellen einer Tabelle die zentrale Anweisung der DDL ist, lassen sich auch andere Elemente von Datenbanken wie Sichten, Sequenzen und sog. *Trigger* erzeugen. Sichten sind allerdings trivial, da sie nur eine Abkürzung für Abfragen sind, die in der Regel für häufige, komplexe Abfragen erstellt werden. Die Definition von Sequenzen und Triggern weicht wiederum sehr stark von System zu System ab, ist jedoch auch nicht für die Typisierung von Abfragen relevant.

Ferner enthält die DDL Anweisungen, die mit `ALTER TABLE` beginnen, über die bestehende Tabellen um Spalten und Restriktionen erweitert oder reduziert werden können. Derartige Anweisungen werden absichtlich in dieser Arbeit nicht berücksichtigt, da für die statische Typisierung von Abfragen auch von einem statischen Schema ausgegangen werden muss. Die Migration laufender Datenbanken ist zwar oft notwendig und muss berücksichtigt werden, betrifft aber die Interpretation der Schemata in dieser Arbeit nicht. Die gängigen Datenbanksysteme stellen Werkzeuge zur Verfügung, die aus laufenden, ggf. über `ALTER TABLE` Anweisungen manipulierten Datenbanken `CREATE TABLE` Sequenzen extrahieren. Mit diesen Anweisungen lässt sich ein Schema in seiner aktuellen Struktur neu erzeugen. Ein so extrahiertes Schema, das nur durch `CREATE`

TABLE Anweisungen beschrieben wird, kann dann zur Überprüfung und Typisierung von Abfragen über das hier vorgestellte Plug-in dienen.

4.4. Anweisungen

Um Abfragen an Datenbanken stellen zu können, müssen diese natürlich Daten enthalten. Dazu sind Anweisungen zum Einfügen, Manipulieren und Löschen der Datensätze notwendig.

```
1 INSERT INTO rights (right_id, name) VALUES(1, 'Create accounts');
2
3 UPDATE actors SET salary = salary * 2 WHERE login = 'cwu';
4
5 DELETE FROM actor_rights WHERE actor_id = 1 AND right_id = 2;
```

Beispiel 4.2: Manipulation von Daten

Anweisungen zur Manipulation von Daten werden im Rahmen dieser Arbeit nicht implementiert. Daher liegt auch in dieser Thesis der Fokus stattdessen auf einer möglichst umfangreichen Abdeckung von Abfragen, dem komplexeren und interessanteren Bereich von Datenbanksystemen. Das Konzept lässt sich allerdings sehr schnell auch auf Anweisungen übertragen, weshalb in den folgenden Kapiteln auch teilweise Aspekte einer möglichen Umsetzung von Anweisungen beleuchtet werden. Aus der Perspektive der Typprüfung ist die Anweisung zum Löschen nur eine vereinfachte Form einer Abfrage ohne Projektion und komplexem Tabellenausdruck. Zentraler Aspekt beim Einfügen hingegen ist die typsichere Parameterübergabe, die ebenfalls schon für Abfragen realisiert werden muss. Die UPDATE-Anweisung ist wiederum aus Sicht der Typprüfung eine Kombination aus Einfügen (SET-Liste) und Abfragen (WHERE-Klausel).

5. Bibliothek

Die Implementierung besteht im Wesentlichen aus einem Plug-in für den Scala-Übersetzer, das Anfragen in der in **Beispiel 1.3** angegebenen Notation transformiert.

Eine Abfrage der folgenden Form muss in einen passenden Scala-Ausdruck überführt werden.

$$\texttt{SELECT}\ p_1, ...p_n\ \texttt{FROM}\ t\ \texttt{WHERE}\ s$$

Zunächst muss geprüft werden, ob der Selektionsausdruck s von einem booleschen Typ ist. Werden dann für die Ausdrücke $p_1, ..., p_n$ der Projektion die Datentypen $t_1, ...t_{,n}$ berechnet und lassen sich diese auf die Scala-Typen $T_1, ..., T_N$ abbilden, muss ein Scala-Ausdruck für eine Anfrage erzeugt werden, die Ergebnisse als Tupel vom Scala-Typ `(T1, ..., TN)` liefert.

JDBC stellt allerdings keine parametrisierten Klassen für Anfragen und deren Ergebnisse bereit. Daher ist zusätzlich zu dem Übersetzer-Plug-in eine Laufzeitbibliothek notwendig, die Oberklassen für typsichere Anfragen bereitstellt.

5.1. Typsichere Anfragen

Zunächst wird eine Basisklasse für alle Anfragen definiert. Die Klasse enthält eine Typvariable, die mit dem Ergebnistyp der Anfrage belegt wird. Dieser Typ hängt von der Art der Anfrage und bei Abfragen auch von deren Struktur ab. Die Schnittstelle der abstrakten Anfrage sieht eine Methode zur Ausführung für eine gegebene *JDBC*-Datenbankverbindung vor. Eine korrekt ausgeführte Anfrage liefert dabei ein Objekt ihres Ergebnistyps zurück.

```
1 abstract class Statement[T](implicit val manifest: Manifest[T]) {
2
3   /* Interface */
4
5   def execute(connection: Connection): T
6
7  ...
8 }
```

Beispiel 5.1: Parametrisierte Anfrage (Schnittstelle)

Die eigentliche Ausführung, d.h. die Übergabe der SQL-Anfrage als Zeichenkette an *JDBC* und die typsichere Extraktion des Ergebnisses, wird an speziellere Unterklassen bzw. vom Übersetzer-Plug-in erzeugte anonyme Unterklassen delegiert.

5.2. Parameter

Anfragen sollen typsicher sein, allerdings nicht nur im Ergebnis, sondern auch in der Übergabe von Parametern. Dazu ist es notwendig, eine Verknüpfung zwischen Platzhaltern in einer Anfrage und dynamischen Werten in der Scala-Umgebung herzustellen. Analog zum Einbetten von Werten in XML-Ausdrücke in Scala kann eine Notation mit geschweiften Klammern dazu dienen, dynamische Werte im Sichtbereich der Scala-Umgebung an eine eingebettete Anfrage zu übergeben.

```
1 val num: Int = 080080
2 val str = "%Prinzipien von Programmiersprachen%"
3 val stm = <?sql SELECT id FROM lectures WHERE number = { num } OR name LIKE { str } ?>
```

Beispiel 5.2: Anfrage mit Parametern

Dabei werden alle Ausdrücke in geschweiften Klammern nicht als SQL, sondern als Scala-Ausdrücke interpretiert.

In einer möglichen Implementierung der Übersetzung lassen sich die eingebetteten Scala-Ausdrücke durch Konkatenation von Zeichenketten übergeben:

```
1 val num: Int = 080080
2 val str = "%Prinzipien von Programmiersprachen%"
3 val stm = new Statement[Iterator[Long]] {
4  def execute(connection: Connection) = {
```

```
5   val sql = "SELECT id FROM lectures WHERE number = " + num + " OR name LIKE '" + str +"'"
6   val jdbcStm = connection.createStatement(sql)
7   ...
8  }
9 }
```

Beispiel 5.3: Hypothetische Übersetzung einer Anfrage

Diese Variante hat mehrere Nachteile. Ein sicherheitskritischer Aspekt ist die Konkatenation, die einen Angriff vom Typ *SQL-Injection* ermöglicht. Dabei übergibt ein Angreifer eine Zeichenkette wie `; DROP TABLE lectures; --`. Über diese Technik ist es dann möglich, beliebige Operationen auf der Datenbank auszuführen, die dem aktuellen Datenbankbenutzer erlaubt sind. Es sollte daher stattdessen die bereits in **Beispiel 1.2** vorgestellten *vorbereiteten Anfragen* von *JDBC* verwendet werden.

Ein weiteres Problem liegt in der Typprüfung der Parameter. Falls ein Wert aus der Umgebung an eine Anfrage übergeben wird, dessen Scala-Typ nicht mit dem erwarteten SQL-Typ kompatibel ist, sollte schon zur Übersetzungszeit ein Typfehler auftreten. Wenn diese Aufgabe vom Übersetzer-Plug-in übernommen werden soll, ist mehr als eine Phase notwendig. Da während der Transformation der Anfragen nach der Syntaxanalyse die Typen der Scala-Ausdrücke noch nicht bekannt sind, müssten diese in einer weiteren Phase nach der Typprüfung mit den in der Anfrage erwarteten Typen verglichen werden. Stattdessen sollte die Transformation die für die Parameter erwarteten Datentypen nur deklarieren, die dann in der Typprüfung des Scala-Übersetzer mit den Typen der Scala-Ausdrücke verglichen werden.

```
1  val num: Int = 080080
2  val str = "%Prinzipien von Programmiersprachen%"
3  val stm = new Statement[Iterator[Long]] {
4    val _1: Long = num
5    val _2: String = str
6    def execute(connection: Connection) = {
7      val sql = "SELECT id FROM lectures WHERE number = ? OR name LIKE ?"
8      val pstm = connection.prepareStatement(sql)
9      pstm.setObject(1, _1)
10     pstm.setObject(2, _2)
11     ...
12   }
13 }
```

Beispiel 5.4: Übersetzung einer Anfrage

Dabei wird für jeden Parameter eine unveränderliche Instanzvariable in der generierten Klasse erzeugt. Die Variable wird mit dem Datentyp versehen, der von der Anfrage erwartet wird und mit dem zu übergebenden Wert belegt.

Die Prüfung der Datentypen von Parametern kann nun von der herkömmlichen Typprüfung des Scala-Übersetzers übernommen werden. Das ermöglicht auch Konversion zwischen primitiven Datentypen. Ist beispielsweise die Spalte `lectures.number` vom SQL-Typ `BIGINT`, wird als Scala-Typ ein `Long` für `_1` erwartet, zu dem `num: Int` konvertierbar ist.

Eine als Zeichenkette enthaltene SQL-Anfrage und die Möglichkeit der typsicheren Parameterübergabe soll für alle Arten von Anfragen gelten. Daher lassen sich diese Gemeinsamkeiten in die Basisklasse auslagern, um die Erzeugung von Zielcode zu reduzieren.

```
1  abstract class Statement[T](implicit val manifest: Manifest[T]) {
2
3    /* Interface */
4
5    def execute(connection: Connection): T
6
7    /* Implementation */
8
9    def toSQL: String
10
11   val params: List[Any] = List()
12
13   def prepare(connection: Connection): PreparedStatement = {
14     val stm = connection.prepareStatement(toSQL)
15     var index = 1
16     params.foreach { param =>
17       stm.setObject(index, param)
18       index += 1
19     }
20     stm
21   }
22 }
```

Beispiel 5.5: Parametrisierte Anfrage (Implementierung)

In der Transformation über das Plug-in müssen lediglich für alle Parameter Instanzvariablen und die eigentliche SQL-Anfrage als Zeichenkette erzeugen. Die Variablen müssen in einer Liste aufgezählt werden, so dass über `prepare`

eine vorbereitetes *JDBC*-Anfrage mit bereits übergebenen Parametern erstellt werden kann. Diese Methode kann von der Methode `execute` verwendet werden, die je nach Art der Anfrage in Unterklassen implementiert ist und im folgenden Abschnitt für Abfragen vorgestellt wird.

5.3. Typsichere Abfragen

Für Abfragen, die in der Regel mehrere Zeilen liefern können, wird eine Unterklasse `BagStatement` definiert. Die Klasse ist mit einer Typvariable versehen, die dem auf Scala abgebildeten Typ einer Zeile im Ergebnis der Abfrage entspricht. In der Methode `execute` wird die Anfrage mit Hilfe der Oberklasse vorbereitet und als Abfrage ausgeführt.

```
1 abstract class BagStatement[T](implicit val valueManifest: Manifest[T])
    extends Statement[ResultSetIterator[T]](manifest) {
2
3   def execute(connection: Connection) = {
4     val resultSet = prepare(connection).executeQuery
5     new ResultSetIterator(resultSet, projection)
6   }
```

Beispiel 5.6: Parametrisierte Abfrage

Die typsichere Abfrage liefert ein Objekt vom Typ `ResultSetIterator`, dessen Typvariable `T` wie die Abfrage selbst an den Typ der Zeile gebunden wird. Das Objekt iteriert über das Ergebnis und liefert Zeilen des Typs `T`. Dazu wird eine Projektion benötigt. Als Projektion wird hier eine Funktion bezeichnet, die ein *JDBC*-Ergebnis vom Typ `java.sql.ResultSet` nimmt und die nächste Zeile als Objekt des generischen Typs aus dem Ergebnis extrahiert.

```
7   val projection = Queries.mkProjection(valueManifest)
8 }
```

Beispiel 5.7: Angabe der Projektion

Eine solche Projektion kann dynamisch zur Laufzeit aus dem Manifest des Typs einer Zeile generiert werden.

Dynamische Projektionen

Eine Projektion, die aus einem *JDBC*-Ergebnis eine Zeile vom Typ `T` extrahiert, ist eine Funktion vom Typ `ResultSet => T`. Diese wird erzeugt, indem jeder Typ im Manifest zu `T` behandelt wird.

```
1  object Queries {
2   def mkProjection[T](manifest: Manifest[T]): ResultSet => T = {
3    if (manifest.erasure.getName.startsWith("scala.Tuple")) {
4     val constr = manifest.erasure.getConstructors.first
5     val arity = constr.getParameterTypes.length
6     val extractors = (1 to arity).map { col =>
7      mkExtractor(col, manifest.typeArguments.apply(column - 1))
8     }
9     rs => constr.newInstance(extractors.map(extr => extr(rs)): _*)
10   } else {
11    mkExtractor(1, manifest)
12   }
13  }
```

Beispiel 5.8: Dynamische Projektion

Zunächst wird in **Beispiel 5.8** geprüft, ob es sich an der Wurzel des Manifests um ein Tupel handelt, was bei Ergebnissen mit mehreren Spalten der Fall ist. Enthält das Ergebnis nur eine Spalte, ist kein Tupel notwendig. Stattdessen wird der Typ der Zeile direkt an den Typ der Spalte gebunden.

Die Wurzel des Manifests wird über die Methode `erasure` berechnet und liefert ein Klassenobjekt der *Java Reflection API*. Wird in Zeile 3 die Klasse über ihren Namen in als Tupel erkannt, existiert auch nur ein Konstruktor (Zeile 4), der das Tupel mit Argument je nach der Stelligkeit `arity` erzeugt. Die Stelligkeit entspricht der Anzahl der Spalten, die im Ergebnis erwartet werden. Für jedes Argument wird in Zeile 7 eine Funktion erstellt, die aus der jeweiligen Spalte im Ergebnis ein Objekt extrahiert. Zu jedem Argument existiert wiederum ein Typargument im Manifest, das beim Extrahieren berücksichtigt wird. In Zeile 9 wird die eigentliche Funktion für die Projektion erstellt, die bei der Anwendung wiederum die Funktionen zum Extrahieren der Tupelargumente anwendet und das Tupel erzeugt.

Ist die Typvariable der Abfrage an einen skalaren Typ - den Typ einer Spalte - gebunden, entspricht die Projektion genau der Funktion zum Extrahieren der einzigen Spalte.

In der Methode, die eine Funktion zum Extrahieren einer Spalte erzeugt, wird in **Beispiel 5.9** zunächst geprüft, ob es sich um einen Typ `Option[C]` handelt.

```
14  def mkExtractor[C](col: Int, manifest: Manifest[_]): ResultSet => C = {
15   if (manifest.erasure == classOf[Option[_]]) { rs =>
16    val valueType = manifest.typeArguments.first
17    val some = Some(mkExtractor(column, valueType)(rs))
18    val result = if (rs.wasNull) None else some
19    result.asInstanceOf[C]
20   } else { rs =>
21    rs.getObject(col).asInstanceOf[C]
22   }
23  }
24 }
```

Beispiel 5.9: Extrahieren von Spalten

Ist dies nicht der Fall, kann in Zeile 21 eine Funktion angeben werden, welche die jeweilige Spalte als typunsicheres Objekt vom Typ `AnyRef` extrahiert (dabei verwendet *JDBC* die zu primitiven Datentypen passenden Klassen), das allerdings problemlos in den erwarteten Typ umgewandelt werden kann. Eine Fallunterscheidung, um die spezielleren Methoden von `ResultSet` wie `getInt` oder `getFloat` zu verwenden, ist an dieser Stelle überflüssig. Diese Methoden liefern zwar Werte primitiver Typen, die allerdings in Objekte umgewandelt werden müssten, damit der Tupelkonstruktor darauf angewendet werden könnte.

Wird ein Typ `Option[C]` erwartet, wird davon ausgegangen, dass die Spalte mit `NULL` belegt sein kann. Bei der Verwendung von *JDBC* muss eine Spalte zunächst extrahiert werden und anschließend über `wasNull` geprüft werden, ob es sich um einen Nullwert handelt, da bei primitiven Ganzzahlen nicht zwischen `0` als Zahl und als Nullwert unterschieden werden kann.

Daher wird rekursiv eine Funktion zum Extrahieren derselben Spalte (allerdings für das im Manifest von `Option[C]` enthaltene Manifest für den Typ `C`) erstellt und angewendet. Das Ergebnis wird nur dann zurückgeliefert, wenn kein Nullwert ausgelesen wurde.

Für die Abfrage in **Beispiel 5.2** wird direkt eine Funktion zum Extrahieren der ersten Spalte vom Typ `Long` zurückgeliefert.

Für die Abfrage im folgenden **Beispiel 5.10** wird der Konstruktor für Tupel der Stelligkeit 4 ermittelt.

```
1 val stm2: BagStatement[(Long, String, String, Option[String])] =
2   <?sql SELECT * FROM lectures ?>
```

Beispiel 5.10: `BagStatement` mit mehreren Spalten

Für die enthaltenen Typen `Long`, `String`, `String` und `Option[String]` wird jeweils eine Funktion erstellt, die je einen Wert aus den vier Spalten des Ergebnisses extrahiert. Beim Anwenden der Projektion werden die vier Funktionen angewendet und das Tupel der Stelligkeit 4 mit den Werten erzeugt.

Die hier vorgestellte Technik, um die Zeilen eines Ergebnisses in Objektbäume eines Typs zu überführen, ist zugegebenermaßen langsamer als kompilierte Methoden, die gezielter mit dem *JDBC*-Ergebnis umgehen würden. Die Erzeugung von Zielcode ist allerdings wesentlich aufwändiger als die Meta-Programmierung mit *Java Reflection*. Weiterhin wird die Mächtigkeit von Typmanifesten in Scala in Verbindung mit der *Java Reflection API* aufgezeigt.

Nullwerte

In **Beispiel 5.8** wurde auch der Umgang mit Nullwerten gezeigt, die ein komplexes Konstrukt in SQL darstellen. Nullwerte werden in SQL anders als in den meisten Programmiersprachen behandelt [8]. So führen Nullwerte in SQL selten zu Laufzeitfehlern: Wird ein Operator auf mindestens einen Nullwert angewendet, ist auch das Ergebnis `NULL`. Dasselbe gilt für die meisten Funktionen, falls Nullwerte als Argumente übergeben werden. Bei der Definition eigener Funktionen in SQL kann über eine Option angegeben werden, ob der Aufruf sofort zu `NULL` ausgewertet werden soll, wenn ein Nullwert übergeben wird. Das dient einerseits der Optimierung, führt allerdings auch zu weitaus weniger Laufzeitfehlern in Abfragen. Etwaige Fallunterscheidungen werden dadurch aus dem Datenbanksystem in die Applikation verlagert, welche die Datenbank verwendet.

Selbst von der Problematik mit primitiven Werten abgesehen ist die Verwendung von Nullreferenzen in einer teils funktionalen Sprache wie Scala nicht üblich. Stattdessen stellt Scala mit `Option` einen Datentyp bereit, der dem

Fall eines gegebenenfalls fehlenden Werts entspricht. Daher sollen Ergebnisspalten, die Nullwerte enthalten können und deren SQL-Datentyp einem Scala-Datentyp `T` entspricht, auf `Option[T]` abgebildet werden. Ein Nullwert wird dann als `None` repräsentiert.

Das Prüfen auf fehlende Werte per *pattern matching* ist allerdings lästig, wenn alle Werte vorhanden sind. Daher sollte das Typsystem der Anfragen, das im folgenden Kapitel 6 vorgestellt wird, möglichst nur diejenigen Spalten in einem Ergebnis auf `Option` abbilden, die auch Nullwerte enthalten können.

Einzeilige Ergebnisse

Es gibt unterschiedliche Bedingungen, unter denen ohne Berücksichtigung des Datenbankzustands das Ergebnis einer Abfrage genau eine Zeile enthält oder leer ist. Fachliche Anforderungen oder ein komplexes Konstrukt von Restriktionen, die dazu führen, dass das Ergebnis eine Abfrage genau eine oder keine Zeile enthält, können nicht oder nur sehr schwierig in der statische Analyse erkannt werden. Allerdings wäre eine Analyse möglich, ob alle Spalten eines eindeutigen Schlüssels einer Tabelle an Werte gebunden sind. Falls ein passender Eintrag vorhanden ist, wird eine Zeile zurückgeliefert, ansonsten ist das Ergebnis leer. In diesem Fall könnte statt einer Typisierung über `ResultSetIterator[T]` direkt eine Abbildung auf `Option[T]` verwendet werden. Dafür ist die Klasse `RowStatement[T]` vorgesehen, über die beim Ausführen höchstens eine Zeile aus dem Ergebnis ausgelesen wird. Diese Klasse ist bereits in der Bibliothek implementiert, wird allerdings von dem im Rahmen dieser Thesis entwickelten Plug-in noch nicht verwendet.

5.4. Anweisungen

Auch wenn im Zuge in dieser Arbeit Anweisungen nicht implementiert werden, lassen sich dennoch passende Klassen dafür konzipieren. In *JDBC* wird eine Anweisung per `executeUpdate` ausgeführt. Das Ergebnis der Methode ist eine ganze Zahl, die für die Anzahl der eingefügten, aktualisierten oder gelöschten Zeilen steht. Für aktualisierende (`UPDATE`) und löschende (`DELETE`) Anweisungen ist diese Abbildung optimal und sollte über eine Klasse `BulkStatement` unterstützt werden, die von `Statement[Int]` erbt.

Diese Abbildung kann auch bei einfügenden Anweisungen (`INSERT`) verwendet werden, in denen mehrere Zeilen eingefügt werden. Für Anweisungen, die nur eine Zeile einfügen, lässt sich eine bessere Abbildung implementieren. Dazu könnte eine Klasse `InsertStatement[K] extends Statement[K]` implementiert werden, wobei die Typvariable `K` mit dem Primärschlüssel der Tabelle belegt wird. Wird eine Zeile eingefügt, liefert die Anweisung typsicher den Primärschlüssel des neuen Eintrags zurück. Das ist insbesondere dann praktisch, wenn ein synthetischer Schlüssel wie eine fortlaufende Nummer oder eine *UUID*[1] von der Datenbank generiert wird.

Bei dem Ausführen von Anweisungen ist nach wie vor die Behandlung von Laufzeitfehlern wichtig. Fehlerbehandlung ist an dieser Stelle notwendig, da die meisten Restriktionen wie Primär- oder Fremdschlüssel nicht statisch überprüft werden können.

[1] *Universally Unique Identifiers* sind systemübergreifende Schlüssel, die sich auch im Parallelbetrieb auf mehreren Systemen erstellen lassen.

6. Einbettung

Neben der Bibliothek existiert ein Plug-in für den Scala-Übersetzer. Das Plug-in identifiziert in Scala-Programme eingebettete SQL-Anfragen und übersetzt diese in neue Ausdrücke unter Verwendung der in der Bibliothek enthaltenen Klassen.

Eine eingebettete Abfrage der folgenden Form muss analysiert und in einen passenden Ausdruck transformiert werden.

$$\texttt{SELECT}\ p_1, ...p_n\ \texttt{FROM}\ t\ \texttt{WHERE}\ s$$

Seien $v_1 : V_1, ..., v_m : V_M$ die eingebetteten Scala-Variablen v_i mit jeweils erwartetem Scala-Typ V_i in den Ausdrücken $s, p_1, ..., p_n$.

Lässt sich dann für den Selektionsausdruck s ein boolescher Typ berechnen und werden für $p_1, ..., p_n$ die SQL-Typen $t_1, ...t_{,n}$ berechnet, die auf die Scala-Typen $T_1, ..., T_N$ abgebildet werden, dann muss ein Zielcode in der Form von **Beispiel 6.1** erzeugt werden.

```
1 new scalasql.lib.BagStatement[TupleN[T1, ..., TN]] {
2     val _1: V1 = v1;
3     ...
4     val _m: VM = vm;
5     val params = scala.List(_1, ..., _m);
6     val toSQL: java.lang.String = "SELECT p1, ..., pn FROM t WHERE s"
7   };
8 }
```

Beispiel 6.1: Übersetzungsschema einer Abfrage

Das Plug-in enthält somit in gewisser Weise einen eigenen SQL-Übersetzer. Allerdings handelt es sich dabei vorwiegend um eine Analysephase für SQL, da kein Zielcode erzeugt wird, der eine eingebettete SQL-Anfrage auswertet. Stattdessen wird ein herkömmlicher Scala-Ausdruck erzeugt, der die Anfrage ausführt.

6.1. Plug-in

Aufgabe des Plug-ins ist es, SQL-Anfragen zu identifizieren und zu transformieren. Daher handelt es sich wie bei der in Abschnitt 3.2 vorgestellten Erweiterung um einen transformierendes Plug-in.

Eingebettete Abfragen der Form `<?sql SELECT ... ?>` werden von der Syntaxanalyse des Scala-Übersetzers in Ausdrücke für die Repräsentationen von XML in Scala umgewandelt. Eine Anfrage wird daher in einen Konstruktoraufruf der Klasse `scala.xml.ProcInstr` für XML-Prozessoranweisungen umgewandelt. Der Konstruktor wird auf zwei Argumente angewendet: den Namen der Prozessoranweisung und deren Inhalt.

Um eine Anfrage zu identifizieren wird daher per Musterabgleich auf Konstruktoraufrufe von `scala.xml.ProcInstr` geprüft. Eine XML-Prozessoranweisung wird als Anfrage erkannt, wenn deren Name `"sql"` entspricht.

```
1     override def transform(tree: Tree): Tree = {
2       tree match {
3         case Apply(
4           Select(New(procInstr), _),
5           List(
6             Literal(Constant("sql")),
7             Literal(Constant(sql: String))))
8             if procInstr.toString.endsWith("scala.xml.ProcInstr") => {
9           val result = analyzeSQL(tree, sql).flatMap { analysis =>
10            compileSQL(analysis)
11          }
12          result match {
13            case Compiled(newTree) => newTree
14            case Error(queryPos, msg) => {
15              val pos = mergePositions(tree.pos, queryPos)
16              unit.error(pos, msg)
17              super.transform(tree)
18            }
19          }
20        }
21        case _ => super.transform(tree)
22      }
23    }
```

Beispiel 6.2: Identifizieren von SQL-Anfragen

Der Inhalt der Prozessoranweisung wird dann als Anfrage erkannt, analysiert und übersetzt.

Im Fall einer erfolgreichen Übersetzung, liefert diese einen neuen Teilbaum des abstrakten Syntaxbaum, der den Konstruktoraufruf der Prozessoranweisung ersetzt. Im Fehlerfall ist es wichtig, dass nicht die Position innerhalb der Anfrage verwendet wird. Auch die Angabe der Position der Anfrage im Quellcode des Programms wäre in der Regel zu ungenau. Daher wird die Position eines Fehlers in der Anfrage auf die der Prozessoranweisung addiert, um möglichst genaue Fehlermeldungen der folgenden Form zu ermöglichen.

```
test/scalasql/Test.scala:92: error: Table actor is unknown.
  val query = <?sql SELECT * FROM actor ?>
                                    ^
one error found
```

Schema

Damit der verwendende Programmierer das Schema definieren kann, das den Anfragen zu Grunde liegt, wird ein Argument für das Plug-in definiert. Dieses wird wie das Argument zum Aktivieren des Plug-ins über die Kommandozeile an den Scala-Übersetzer übergeben.

```
1  class ScalaSQLPlugin(val global: Global) extends Plugin {
2   var schemaFileName = "schema.sql"
3
4   override def processOptions(options: List[String],
5     error: String => Unit) {
6     for (option <- options) {
7       if (option.startsWith("schema:")) {
8         schemaFileName = option.substring("schema:".length)
9       } else {
10        error("Option %s not understood.".format(option))
11      }
12    }
13  }
14  override val optionsHelp = Some(
15     "-P:scalasql:schema:file set the file name of the schema to use")
16  ...
17 }
```

Beispiel 6.3: Argument für das Schema

Das Argument ist optional und wird in Zeile 2 von **Beispiel 6.3** mit einem initialen Wert belegt. Als Standardeinstellung wird versucht, die Datei mit dem Namen `schema.sql` zu laden, die sich im Ordner befinden muss, von dem aus der Scala-Übersetzer gestartet wird. Wird das Argument übergeben, dann wird als Wert ein relativer oder absoluter Pfad zu einer Textdatei erwartet, die Anweisungen im Format `CREATE TABLE ...` enthält.

```
~$ scalac -Xplugin:plugin-scalasql.jar
  -P:scalasql:schema:/pathTo/someSchema.sql SourceFile.scala
```

6.2. Übersetzung

Die Übersetzung einer Anfrage erfolgt in mehreren Phasen:

I. Das der Anfrage zugrunde liegende Datenbankschema wird gesucht.

II. Das Schema wird analysiert und daraus ein Tabellenindex generiert.

III. Die Syntax der Abfrage wird analysiert.

IV. Die Abfrage durchläuft eine Typprüfung:

   a) Die Umgebung der Abfrage wird als lokaler Tabellenindex und Symboltabelle der Spalten aus dem Tabellenausdruck erstellt.

   b) Falls vorhanden, wird der Typ der Selektion geprüft.

   c) Die Projektion der Abfrage wird ermittelt.

V. Der Zielcode wird erzeugt.

Die Phasen, die das Datenbankschema betreffen, sind nur einmal in einem Durchlauf des Scala-Übersetzers notwendig. Bei der Übersetzung von mehreren Anfragen in einem Scala-Programm kann der Index der Tabellen, die in Anfragen verwendet werden können, wiederverwendet werden.

Um die Phasen des Übersetzungsprozesses und deren Unterphasen zu kombinieren, lässt sich eine parametrisierte Hilfsdatenstruktur `Compile[T]` implementieren, die der Datenstruktur `Option[T]` nachempfunden ist.

```
1  sealed abstract class Compile[+T] {
2    def map[E](f: T => E): Compile[E]
3    def flatMap[E](f: T => Compile[E]): Compile[E]
4  }
5  case class Compiled[+T](result: T) extends Compile[T] {
6    def map[E](f: T => E) = Compiled(f(result))
7    def flatMap[E](f: T => Compile[E]) = f(result)
8  }
9  case class Error(pos: Position, msg: String) extends Compile[Nothing] {
10   def map[E](f: Nothing => E) = this
11   def flatMap[E](f: Nothing => Compile[E]) = Error(pos, msg)
12 }
```

Beispiel 6.4: Hilfsdatenstruktur für Teilergebnisse

Anstelle eines generischen Werts `None` wird über die Klasse `Error` eine Fehlermeldung zu einer Position im Quellcode geliefert. Über die Methode `map` können Ergebnisse umgewandelt werden. Mit Hilfe der Methode `flatMap`, die auch in **Beispiel 6.2**, Zeile 9 verwendet wird, können mehrere Schritte in Reihe geschaltet werden. Dabei ist es unerheblich, ob ein Fehler in `analyzeSQL` oder in `compileSQL` auftritt.

Im Folgenden werden Aspekte der einzelnen Schritte in der Übersetzung einer eingebetteten Anfrage weiter beleuchtet.

6.3. Syntaxanalyse

Um Anfragen zu übersetzen, ist es einerseits notwendig ihre Syntax zu analysieren. Andererseits muss auch das der Anfrage zugrunde liegende Datenbankschema analysiert werden. Es werden daher Parser für die Teile der Grammatik von SQL benötigt, die Anfragen und Schemadefinitionen betreffen. Die komplette Grammatik von SQL ist weitaus umfangreicher als die der meisten Programmiersprachen, da insbesondere für *Stored Procedures*, die in Sprachen wie *C* oder *COBOL* implementiert werden können, auch die Grammatik dieser Sprachen enthalten sein müssen. Allerdings wird wohl von keinem Datenbanksystem die komplette Grammatik von SQL unterstützt. So wird auch in dieser Arbeit nur ein relevanter Ausschnitt der Grammatik des Standards betrachtet, der im Anhang B aufgeführt wird.

Die Syntaxanalyse für diesen Ausschnitt der Grammatik wird mit Parser-Kombinatoren realisiert, die von der Scala-Bibliothek bereitgestellt werden.

Parser-Kombinatoren

Eine Reihe von Operatoren und Funktionen ermöglichen es in Verbindung mit den Typsystem von Scala, Parser fast direkt nach den Ableitungsregeln einer Grammatik in Backus-Naur-Form zu implementieren. Mit Hilfe dieser Kombinatoren lassen sich Parser für *LL*-Grammatiken realisieren, was für den in dieser Thesis relevanten Teil der Grammatik von SQL ausreichend ist. Die Effizienz der mit *Backtracking* realisierten Parser kann sich zwar nicht mit der einer generierten Syntaxanalyse messen, ist allerdings auch ausreichend, da in der Regel keine umfangreichen Programme, sondern nur Abfragen eingelesen werden.

Für die Produktionen einer Grammatik werden Funktionen vom parametrisierten Typ `Parser[T]` implementiert.

```
1 // <table definition> ::= CREATE [ <table scope> ] TABLE <identifier> <table element list>
2 def tableDefinition = "CREATE" ~ opt(tableScope) ~ "TABLE" ~ identifier ~ tableElementList
```

Beispiel 6.5: Basiskombinatoren

Zeichenketten können direkt verwendet werden, um Schlüsselwörter einzulesen. Wird eine Zeichenkette durch die Verwendung des Operators ~ für sequentielle Komposition mit einem anderen Parser kombiniert, wird die Zeichenkette in einen Parser für das jeweilige Schlüsselwort umgewandelt. Die Funktion `opt` nimmt einen Parser vom Typ `Parser[T]` und generiert einen neuen Parser vom Typ `Parser[Option[T]]`, über den optionale Syntaxelemente eingelesen werden können.

Um die mögliche Wiederholung von Nichtterminalsymbolen zu realisieren, wird werden eine Reihe von Funktionen bereitgestellt, die einen Parser vom Typ `Parser[T]` in einen weiteren vom Typ `Parser[List[T]]` überführen.

```
3 // <table element list> ::= <left paren> <table element> [ { <comma> <table element> }...
      ] <right paren>
4 def tableElementList = "(" ~> rep1sep(tableElement, ",") <~ ")"
```

Beispiel 6.6: Wiederholung von Nichtterminalsymbolen

Die Funktion `repsep` nimmt dabei noch einen zusätzlichen Parser für ein Trennsymbol zwischen den wiederholten Nichtterminalsymbolen. Die in **Beispiel 6.6** verwendete Funktion `rep1sep` schlägt im Gegensatz zu `repsep` fehl, wenn kein Element eingelesen werden konnte. Die Operatoren `~>` und `<~` sind nützlich, um Terminalsymbole einzulesen, die nicht weiterverarbeitet werden müssen. Die Operatoren entsprechen der sequentiellen Komposition, bei der das Ergebnis des links- bzw. rechtsseitigen Parsers verworfen wird. Hier wird eine Klammerung um die Liste der Tabellenelemente gefordert, allerdings reicht diese Liste als Ergebnis des Parsers aus.

Um mehrere Alternativen für eine Produktion anzugeben, kann der Operator `|` auf mehrere Parser-Definitionen gleichen Typs angewandt werden.

```
5  // <table element> ::= <column definition> | <table constraint definition>
6  def tableElement = columnDefinition | tableConstraintDefinition
```

Beispiel 6.7: Alternativen einer Regel

Syntaxbaum

Neben der Prüfung auf korrekte Syntax werden die eingelesenen Tokens zur Weiterverarbeitung in Elemente eines abstrakten Syntaxbaums überführt. Dafür existiert für jede Produktion der Grammatik eine Klasse, die den `trait AST` einbindet.

```
1  trait AST extends Positional {
2    def toSQL: String
3  }
4  case class Table(optScope: Option[TableScope], name: SQLIdentifier, contents: List[
       TableElement]) extends AST {
5   def toSQL = "CREATE " + ...
6  }
```

Beispiel 6.8: Elemente des abstrakten Syntaxbaums

Dabei wird einerseits eine Methode `toSQL` gefordert, die den Teil des Baums wieder in SQL überführt. Der `trait` erbt von `Positional`, der wiederum eine Variable auf eine Position im Quellcode hält. Dadurch ist es möglich, die Elemente des abstrakten Syntaxbaums den dazugehörigen Elementen im Ableitungsbaum zuzuordnen. Im Fall von Fehlern, die wie beispielsweise Typfehler in Phasen nach der Syntaxanalyse auftreten, können die Fehlermeldungen mit einer oftmals hilfreichen Positionsangabe versehen werden.

Der Operator ^^ wird verwendet, um das Ergebnis eines Parsers vom Typ `Parser[T]` über eine Funktion `T => E` zu transformieren.

```
1 // <table definition> ::= CREATE [ <table scope> ] TABLE <identifier> <table element list>
2 def tableDefinition = positioned {
3   ("CREATE" ~ opt(tableScope) ~ "TABLE" ~ identifier ~ tableElementList) ^^ {
4     case _ ~ optScope ~ _ ~ name ~ contents => Table(optScope, name, contents)
5   }
6 }
```

Beispiel 6.9: Ergebnistransformation und Position

Dabei entsteht ein neuer Parser vom Typ `Parser[E]`. Die Funktion `positioned` setzt die Position im Quellcode in einem Ergebnis eines übergebenen Parsers, das wie `AST` vom `trait Positional` erbt.

Da die Transformation hier nur zum Überführen von eingelesenen Regeln in Elemente des abstrakten Syntaxbaums genutzt wird, die alle eine Position im Code erhalten sollen, lassen sich diese beiden Schritte zusammenfassen. Es lässt sich ein weiterer Operator `@@` implementieren, der das Ergebnis eines Parsers transformiert und dabei die Position setzt. Da der Operator ^^ in der Klasse `Parser` definiert ist, müsste auch `@@` dort definiert werden. Dazu wird das in Abschnitt 2.4 vorgestellte Verfahren mit impliziten Konversionen angewendet.

```
1 trait SQLParsers extends SQLTokenParsers {
2
3   class ExtendedParser[+T](wrapped: Parser[T]) extends Parser[T] {
4     def apply(in: Input) = wrapped(in)
5     def @@[U <: AST](f: T => U): Parser[U] = positioned(this ^^ f)
6   }
7   implicit def extendParser[T](p: Parser[T]): ExtendedParser[T] = new ExtendedParser(p)
8   ...
9 }
```

Beispiel 6.10: Erweiterung von `Parser`

Es wird eine Unterklasse von `Parser` implementiert, die als *Decorator* für einen Parser fungiert. In Zeile 5 wird dann der neue Operator `@@`, der Transformation und Positionsangabe kombiniert, für diese Klasse definiert. Über die Funktion `extendParser` wird die Konversion in allen Parser-Komponenten sichtbar, die von `SQLParsers` erben.

Beispiel 6.9 lässt sich damit wie folgt vereinfachen.

```
1 // <table definition> ::= CREATE [ <table scope> ] TABLE <identifier> <table element list>
2 def tableDefinition = ("CREATE" ~ opt(tableScope) ~ "TABLE" ~ identifier ~
      tableElementList) @@ {
3   case _ ~ optScope ~ _ ~ name ~ contents => Table(optScope, name, contents)
4 }
```

Beispiel 6.11: Ergebnistransformation mit Position

Parser-Komponenten

Parser für Tabellen- und Spaltendefinitionen sind nur für die Analyse der Schemata notwendig, für die allerdings Parser für Abfragen nicht benötigt werden. Die Parser werden daher in unterschiedlichen Komponenten definiert.

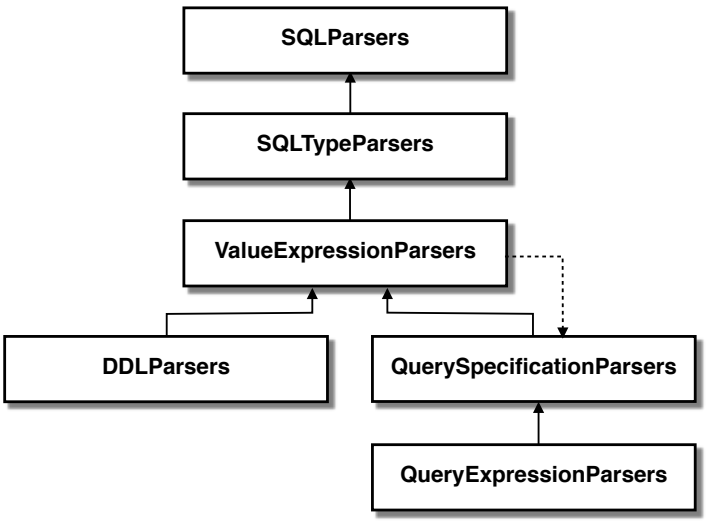


Abbildung 6.1.: Abhängigkeiten der Parser-Definitionen

Sowohl die Analyse der Schemata als auch die der Abfragen basieren auf den Parser-Definitionen für Ausdrücke über Werte. Durch Ausdrücke der Form `CAST(<value expression> AS <data type>)` basieren diese auf der Parser-Komponente für Typdeklarationen. Diese wiederum basiert auf der Basiskomponente `SQLParsers`, über die neben dem Operator `@@` auch Parser für Bezeichner und Literale bereitgestellt werden. Die Parser für Abfragen sind aufgeteilt in eine Komponente `QuerySpecificationParsers` für die eigentlichen Abfragen und eine weitere Komponente für Ausdrücke über Abfragen wie Mengenoperationen und (ggf. rekursive) `WITH`-Klauseln.

Jede Komponente wird als `trait` implementiert und enthält Parser-Funktionen der in **Beispiel 6.5 ff.** vorgestellten Form. Die Syntaxanalysen für Datenbankschemata und Abfragen lassen sich als Klassen definieren, die von den jeweils benötigten Parser-Komponenten erben.

Durch die booleschen Ausdrücke der Form `EXISTS (<query specification>)` und `<value expression> IN (<query specification>)`, die Unterabfragen enthalten können, ist prinzipiell eine gegenseitige Abhängigkeit zwischen den Komponenten für Ausdrücke und Abfragen vorhanden. Allerdings sind Ausdrücke mit Unterabfragen nur im Kontext von Abfragen selbst möglich, nicht aber als Standardwerte für Spalten in Tabellendefinitionen.

```
1  trait ValueExpressionParsers extends SQLTypeParsers {
2  ...
3    def querySpecification: Parser[QuerySpecification] = failure("Sub" +
4      "queries are not allowed at this point.")
5  }
6
7  trait QuerySpecificationParsers extends ValueExpressionParsers {
8    // <query specification> ::= SELECT [ <set quantifier> ] <select list>
         <table expression>
9    override def querySpecification = "SELECT" ~ opt(setQuantifier) ~
         selectList ~ tableExpression @@ {
10     case _ ~ quan ~ proj ~ expr => QuerySpecification(quan, proj, expr)
11   }
12 ...
13 }
```

Beispiel 6.12: Parser für Unterabfragen

Dazu wird in der Komponente für Ausdrücke über Werte zunächst ein Parser für (Unter-)Abfragen vorgesehen. Im Kontext eines Parsers für Abfragen, der dementsprechend vom `trait QuerySpecificationParsers` erbt, wird die Parser-Definition für Abfragen überschrieben. Die Methode wird in der Komponente `DDLParsers` für Tabellendefinitionen nicht neu definiert, so dass die Basisimplementierung zu einem Fehlschlag führt, falls ein `IN`/`EXISTS`-Ausdruck als Standardwert für eine Tabellenspalte deklariert wird.

6.4. Symboltabellen

In Ausdrücken können Spaltennamen auftreten, die mit Schema- und Tabellennamen qualifiziert sein können, aber nicht müssen. Im Tabellenausdruck einer Abfrage, aber auch in Projektionseinträgen der Form `actors.*` können wiederum Tabellennamen auftreten, die ebenso möglicherweise mit dem Schemanamen qualifiziert werden. Daher sind Symboltabellen für Tabellen (die Tabellen eines Schemas als globaler Tabellenindex und die Tabellen einer Anfrage als lokaler Tabellenindex) und für die je nach Tabellenausdruck in einer Abfrage verfügbaren Spalten notwendig.

Da Spalten- und Tabellennamen nicht qualifiziert werden müssen, kann das Nachschlagen eines Bezeichners wegen einer Mehrdeutigkeit scheitern. Die parametrisierte Schnittstelle einer Symboltabelle mit möglichen Mehrdeutigkeiten lässt sich wie folgt definieren.

```
1 trait SymbolTable[+T] {
2   def lookup(id: SQLIdentifier): Lookup[T]
3 }
4 sealed abstract class Lookup[+T]
5 case class LookedUp[+T](result: T) extends Lookup[T]
6 case class Ambiguous(in: List[String]) extends Lookup[Nothing]
7 case object Unknown extends Lookup[Nothing]
```

Beispiel 6.13: Schnittstelle für Symboltabellen

Im Fall einer Mehrdeutigkeit muss dann angegeben werden, in welchen Tabellen oder Schemata die Bezeichner von Spalten bzw. Tabellen nicht eindeutig sind. Da diese Informationen nur in Fehlermeldungen relevant ist, können sie als Zeichenketten übergeben werden.

Die Symboltabelle für die Spalten einer Tabelle lässt sich effizient mit einer Hashtabelle und einer Menge realisieren. Die Hashtabelle bildet die Spaltennamen auf die passenden Spaltendefinitionen ab. In der Menge werden alle Suffixe des Tabellennamens abgelegt. Diese stellen potentielle Präfixe für Spaltennamen in qualifizierten Bezeichnen dar. Für die Tabelle `scalasql.public.actors` werden dann die folgenden Präfixe abgelegt.

```
List("scalasql","public","actors")
List("public","actors")
List("actors")
Nil
```

Bei der Verwendung eines Bezeichners wie `public.actors.login` werden die Präfixe `List("public", "actors")` vom Spaltennamen `"login"` getrennt und separat in der Menge und der Hashtabelle nachgeschlagen.

Beim Nachschlagen in einer Symboltabelle für die Spalten einer Tabelle kann keine Mehrdeutigkeit auftreten, da die Tabelle vorher auf eindeutige Spaltennamen validiert wurde. Mehrdeutigkeiten können erst auftreten, wenn mehrere Symboltabellen zusammengeführt werden. Das Zusammenführen wird bei Tabellenausdrücken notwendig, in denen mehrere Tabellen durch Kreuzprodukt oder Verbund kombiniert werden.

```
1  def merge[T](symbolTables: List[SymbolTable[T]]) =
2    symbolTables match {
3     case table :: tables => {
4      val merged = merge(tables)
5      new SymbolTable[T] {
6       def lookup(id: SQLIdentifier) = {
7         (table.lookup(id), merged.lookup(id)) match {
8          case (Unknown, lookedUp) => lookedUp
9          case (lookedUp, Unknown) => lookedUp
10         case (LookedUp(_), LookedUp(_)) => Ambiguous(List(table.toString,merged.toString))
11         case (LookedUp(_), Ambiguous(in)) => Ambiguous(table.toString :: in)
12         case (Ambiguous(in), LookedUp(_)) => Ambiguous(merged.toString :: in)
13         case (Ambiguous(in1), Ambiguous(in2)) => Ambiguous(in1 ::: in2)
14        }
15       }
16      }
17     }
18    case _ => emptyTable
19  }
```

Beispiel 6.14: Zusammenführen von Symboltabellen

Beim Zusammenführen mehrerer Symboltabellen wird der Bezeichner in einer Symboltabelle und in den rekursiv zusammengeführten weiteren Symboltabellen nachgeschlagen. Das Nachschlagen ist nur dann erfolgreich, wenn es in einer der beiden Tabellen erfolgreich und der Bezeichner in der jeweils anderen Symboltabelle unbekannt ist. In allen anderen Fällen tritt eine Mehrdeutigkeit auf.

In manchen Fällen ist es notwendig, Symboltabellen in Reihe zu schalten. Dabei wird ein Bezeichner nur dann in der zweiten Tabelle nachgeschlagen, wenn dieser in der ersten Symboltabelle unbekannt war.

```
1 def prepend[T](st1: SymbolTable[T], st2: SymbolTable[T]) =
2   new SymbolTable[T] {
3     def lookup(id: SQLIdentifier) = st1.lookup(id) match {
4       case Unknown => st2.lookup(id)
5       case result => result
6     }
7   }
```

Beispiel 6.15: Präzedenz bei Symboltabellen

Dies ist im Wesentlichen bei Unterabfragen notwendig. In der folgenden Anfrage tritt die Tabelle `actors` im Sichtbereich der Unterabfrage zweimal auf.

```
SELECT * FROM actors a WHERE login IN(SELECT parent FROM actors b)
```

Qualifiziert mit `a` könnte jede Spalte der Tabelle auch in der Unterabfrage verwendet werden. Dennoch führt die Projektion von `parent`, die nicht mit `b` qualifiziert wurde, nicht zu einer Mehrdeutigkeit. Die Tabelle der Unterabfrage hat hier die höhere Priorität. Dazu muss die Symboltabelle der Unterabfrage vor die der äußeren Abfrage geschaltet werden.

Die Funktion `prepend` kann auch verwendet werden, um den natürlichen Verbund zu realisieren. Im Tabellenausdruck der folgenden Abfrage sind alle Spalten der Tabellen `actors` und `actor_rights` sichtbar. Allerdings werden doppelte Spalten eliminiert, da im natürlichen Verbund die Zeilen beider Tabellen über die Gleichheit in Spalten gleichen Namens kombiniert werden.

```
SELECT actor_id FROM actors NATURAL JOIN actor_rights
```

Das Nachschlagen von `actor_id` darf also nicht zu einer Mehrdeutigkeit führen. Da für den Verbund die Datentypen der Spalten gleichen Namens in beiden Tabellen gleich sein müssen, können diese in der Symboltabelle der zweiten Tabelle verworfen werden. Wird die Symboltabelle von `actors` mit der Funktion `prepend` vor die für `actor_rights` geschaltet, erfüllt dies genau die Spezifikation des natürlichen Verbunds.

Das Nachschlagen von Spaltendefinitionen und deren Datentypen zu in Ausdrücken auftretenden Spaltennamen bildet die Basis für die Typprüfung der Selektion und die Typinferenz der Projektion.

6.5. Typprüfung

Die Prüfung auf korrekte Datentypen ist an mehreren Stellen notwendig. Bereits in der Analyse des Tabellenausdrucks einer Abfrage kann eine Typprüfung notwendig werden.

```
actors a JOIN actor_rights ar ON a.actor_id = ar.actor_id
```

Dies ist beim qualifizierten Verbund der Fall, bei dem der Datentyp der Bedingung für den Verbund (hier `a.actor_id = ar.actor_id`) berechnet und auf `BOOLEAN` validiert werden muss. Analog muss die Bedingung einer `WHERE`-Klausel auf booleschen Typ geprüft werden.

Zentral ist die Berechnung der Datentypen für die Projektion. Diese werden bei der Erzeugung des Zielcodes auf die Datentypen von Scala abgebildet und formen den Ergebnistyp einer Abfrage in Scala.

Damit eine Typprüfung implementiert werden kann, müssen zunächst Datentypen definiert werden, die in Abfragen auftreten können.

Datentypen

Ein Problem bei der Abbildung der Datentypen von SQL auf Scala und umgekehrt ist die unterschiedliche Genauigkeit. Während die Wertebereiche der primitiven Typen und Zeichenketten in Scala klar definiert und für primitive Typen keine Nullwerte erlaubt sind, können (Maximal-)Längen von Zeichenketten und Präzision von manchen numerischen Typen in SQL genau spezifiziert werden. Nullwerte werden in SQL wiederum für jeden Typ erwartet.

Grundsätzlich bietet sich eine Abbildung der SQL-Typen auf die Datentypen von Scala gemäß *JDBC* an, die so von der Bibliothek unterstützt wird, wie in Kapitel 5 gezeigt wurde.

Die Genauigkeit der Datentypen wie Längen der Zeichenketten sind im Wesentlichen nur für das Datenbanksystem zur Optimierung der Persistenz relevant und kann bei Abfragen vernachlässigt werden. Daher wird die Genauigkeit bei Literalen und (sofern angegeben) in der Syntaxanalyse der Tabellen eines Schemas zwar erfasst, allerdings erfolgt keine statische Berechnung der Genauigkeit/Längen von Datentypen, die zum Beispiel bei Operatorapplikationen entstehen.

Beispielsweise könnte bei der Konkatenation einer Zeichenkette fester Länge vom Typ `CHAR(n)` und einer variablen Zeichenkette des Typs `VARCHAR(k)` zwar der Ergebnistyp genau[1] als `VARCHAR(n + k)` bestimmt werden, allerdings ergibt sich dadurch für die Übersetzung der Abfrage kein Mehrwert. Bei einer Multiplikation zweier Zahlen vom Typ `DECIMAL(p, s)` und `FLOAT` wäre die Berechnung des Ergebnistyps weitaus komplexer oder gar nicht genau definiert, bringt wiederum keinen Mehrwert und kann vernachlässigt werden.

Die Genauigkeit kann bei Abfragen in jedem Fall vernachlässigt werden. Bei einfügenden und aktualisierenden Anweisungen wäre eine Validierung der Wertebereiche prinzipiell wichtig. Allerdings ist nur bei der Verwendung von Literalen und Spalten die statische Berechnung der Genauigkeit möglich. Bei den dynamischen Werten durch Parameter aus der Scala-Umgebung wären entweder Laufzeitfehler in Kauf zu nehmen oder eine automatische Reduktion auf den Wertebereich zur Laufzeit vorstellbar.

Weitaus wichtiger ist der Umgang mit Nullwerten. Wie bereits in Abschnitt 5.3 festgestellt wurde, ist die Darstellung von `NULL` (SQL) durch den Zeiger `null` (Scala, JVM) für primitive Typen gar nicht möglich, aber auch für Objekttypen wie Zeichenketten aus funktionaler Perspektive nicht wünschenswert. Daher sollten die Datentypen von Ausdrücken, die `NULL` als Wert annehmen können, mit `Option[T]` abgebildet werden. Daher muss in einem Datentyp erfasst sein, ob `NULL` in seinem Wertebereich liegt.

Dies kann in einer Basisklasse für Datentypen erfasst werden.

```
1 case class ScalaType(name: String, args: List[ScalaType])
2
3 abstract class DataType extends AST {
4   def scalaType: ScalaType
5   val nullable: Boolean
6   def asNullable(n: Boolean): DataType
```

Beispiel 6.16: Basisklasse für Datentypen

Zunächst wird für jeden Datentyp die Repräsentation des dazugehörigen Datentyps in Scala gefordert. Dazu dient der rekursive Typ `ScalaType`, über den ggf. parametrisierte Scala-Typen dargestellt werden können.

[1] Genau genommen wäre das Ergebnis eine Zeichenkette mit n bis $n + k$ Zeichen, was sich allerdings auch mit SQL-Typen nicht ausdrücken lässt.

Der Typ `Option[Array[Byte]]`, der für Ausdrücke des SQL-Typs `BLOB` inklusive Nullwerte verwendet wird, lässt sich wie folgt darstellen.

```
ScalaType("scala.Option", List(
    ScalaType("Array", List(
        ScalaType("Byte", Nil)))))
```

Für einen Datentyp wird zusätzlich erfasst, ob Nullwerte möglich sind, damit eine prägnante Abbildung mit bzw. ohne `Option` möglich ist. Da in der Typberechnung häufig ein Typ einer Operator- oder Funktionsapplikation für das Ergebnis in den dazugehörigen Datentyp inklusive oder exklusive Nullwerten überführt werden muss, wird ferner die Methode `asNullable` gefordert.

Damit beispielsweise bei mehreren passenden Operatoren für zwei Datentypen der speziellste Operator verwendet werden kann, muss eine partielle Ordnungsrelation über den unterstützten Typen definiert werden.

```
7   def isSuperTypeOf(t: DataType): Boolean
8   def isSubTypeOf(t: DataType): Boolean = t.isSuperTypeOf(this)
9   def isEqualTo(t: DataType) = isSuperTypeOf(t) && isSubTypeOf(t)
10  def isOrthogonalTo(t: DataType) = !isSuperTypeOf(t) && !isSubTypeOf(t)
```

Beispiel 6.17: Partielle Ordnung über Datentypen

Dazu muss die Methode `isSuperTypeOf` implementiert werden. Alle weiteren Beziehungen zu anderen Datentypen lassen sich davon ableiten.

Für die Implementierung der erforderlichen Methoden in den Klassen konkreter Datentypen werden Hilfsmethoden bereitgestellt.

```
11  def ratherNullable(that: DataType) = nullable || !that.nullable
12  def createScalaType(name: String, args: String*) =
13    wrapNullable(ScalaType(name, args.toList.map(ScalaType(_, List()))))
14  def wrapNullable(t: ScalaType) =
15    if (nullable) ScalaType("scala.Option", List(t)) else t
16 }
```

Beispiel 6.18: Vergleichsmethoden für Datentypen

Ein Datentyp A kann in der Hierarchie der Typen nur dann über einem anderen Datentyp B stehen, wenn A Nullwerte zulässt oder durch B keine Nullwerte zugelassen werden. Dazu kann in Implementierungen der Methode `isSuperTypeOf` die Methode `ratherNullable` verwendet werden.

Einfache Scala-Typen mit maximal einer Hierarchiestufe in der Parametrisierung lassen sich mit Hilfe von `createScalaType` konstruieren. Der Typ `Array[Byte]` wird über den Aufruf `createScalaType("Array", "Byte")` erzeugt. Im Fall von Datentypen, die Nullwerte zulassen, wird der Scala-Typ durch `wrapNullable` mit `Option[...]` dekoriert.

Im Rahmen dieser Arbeit werden die booleschen, numerischen, textuellen und binären Datentypen von SQL unterstützt. **Abbildung 6.2** zeigt die Hierarchie der unterstützten Datentypen ohne Berücksichtigung von Genauigkeit und Nullwerten sowie die jeweilige Abbildung auf Typen von Scala.

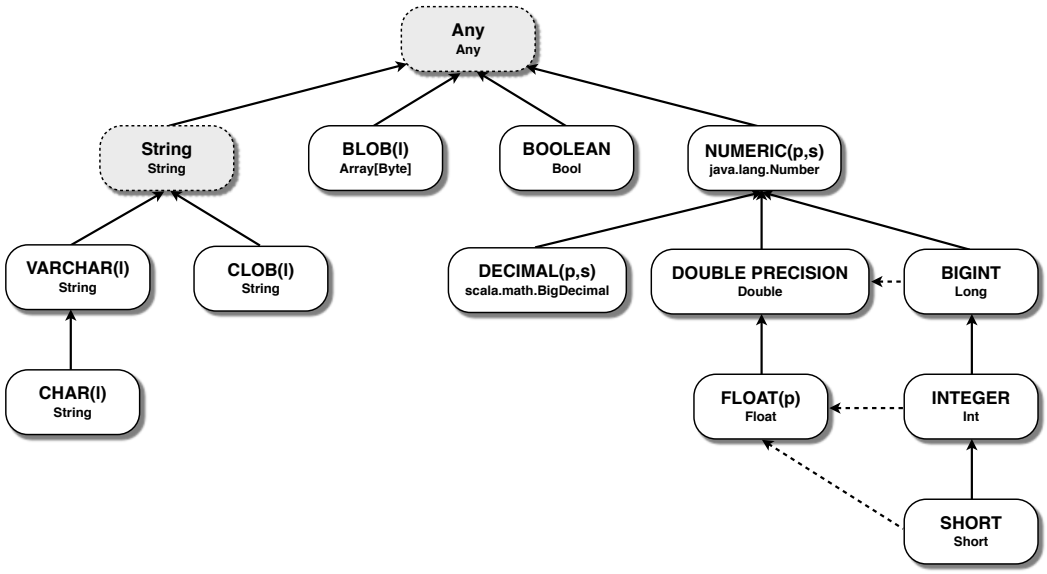


Abbildung 6.2.: Hierarchie der Datentypen

Die Typen `String` und `Any` existieren so in SQL nicht. Ein abstrakter Datentyp `String` bietet sich aber für die Definition gemeinsamer Eigenschaften von textuellen Typen an. `Any` wird verwendet, um für das Argument eines `CAST`-Ausdrucks einen erwarteten Typ anzugeben.

Bei allen SQL-Typen, die mit einer Genauigkeit versehen werden können, ist die Angabe optional. Wird die Genauigkeit nicht angegeben, muss vom maximalen Wertebereich ausgegangen werden. Die Genauigkeit muss wie mögliche Nullwerte in der Ordnungsrelation berücksichtigt werden. Während `VARCHAR` prinzipiell ein allgemeinerer Typ als `CHAR` ist, sind `VARCHAR(1)` und `CHAR(2)` allerdings orthogonal zueinander.

```
1  object DfltVariableStringType extends VariableStringType(None,true,None)
2  class VariableStringType(
3    _alias: Option[String],
4    _nullable: Boolean,
5    _length: Option[Int]) extends StringType(_alias, _nullable, _length) {
6
7    override def toSQL =
8      alias.getOrElse("VARCHAR") + length.mapOrEmpty("(" + _ + ")")
9
10   override def asNullable(n: Boolean) =
11     new VariableStringType(alias, n, length)
12
13   override def isSuperTypeOf(that: DataType) = that match {
14     case that: FixedStringType => ratherNullLonger(that)
15     case that: VariableStringType => ratherNullLonger(that)
16     case _ => nullable && that == NullType
17   }
18 }
```

Beispiel 6.19: Zeichenkette variabler Länge

Die Methode `ratherNullLonger` prüft zusätzlich zu `ratherNull`, ob die Länge des Typs größer oder gleich der Länge des übergebenen Typs ist. Das Objekt `NullType` ist ein spezieller Datentyp, der Untertyp für jeden Datentyp ist, der Nullwerte annehmen kann, und wird nur `NULL` als Typ zugeordnet. Für jede Klasse, die einen Datentyp repräsentiert, wird ein Objekt definiert (hier `DfltVariableStringType`), das die allgemeinste Variante mit dem maximalen Wertebereich und Nullwerten darstellt. Diese Objekte können verwendet werden, um Signaturen von Operatoren und Funktionen zu definieren.

Die SQL-Datentypen lassen sich bezüglich ihrer Syntax in drei Gruppen einteilen. Es existieren Datentypen ohne definierbare Genauigkeit (wie `INTEGER`), mit optionaler Länge (wie `VARCHAR`) und zusätzlich optionalen Nachkommastellen (wie `DECIMAL`). Es ist nicht notwendig für jeden Datentyp eine eigene Parser-Funktion zu definieren. Stattdessen kann für jede der drei Arten eine generische Parser-Funktion aus einer Liste von Typdeklarationen implementiert werden. Die unterstützten Datentypen werden dazu zentral in der Klasse `DataTypes` aufgelistet. In Kapitel 7 werden in einer Unterklasse von `DataTypes` durch Überschreiben der Methoden die Datentypen eines bestimmten Datenbankanbieters hinzugefügt.

```
1  class DataTypes {
2   def noLength = List(
3    (List("BOOLEAN"), new BooleanType(_,false)),
4    (List("SMALLINT"), new ShortType(_,false)),
5    (List("INTEGER","INT"), new IntegerType(_,false)),
6    (List("BIGINT"), new LongType(_,false)),
7    (List("DOUBLE PRECISION"), new DoubleType(_,false)))
8
9   def optionalLength = List(
10   (List("FLOAT"), new FloatType(_,false,_)),
11   (List("CHARACTER LARGE OBJECT","CHAR LARGE OBJECT","CLOB"), new TextType(_,false,_)),
12   (List("CHAR VARYING","CHARACTER VARYING","VARCHAR"), new VariableStringType(_,false,_)),
13   (List("CHARACTER", "CHAR"), new FixedStringType(_,false,_)),
14   (List("BINARY LARGE OBJECT","BLOB"), new BinaryType(_,false,_)))
15
16  def optionalLengths = List(
17   (List("NUMERIC","DECIMAL","DEC"), new DecimalType(_,false,_,_)))
18 }
```

Beispiel 6.20: Deklaration der Datentypen

Für jede Art wird eine Liste von Typdeklarationen angegeben. Jede Typdeklaration enthält die möglichen Namen des Datentyps als Liste von Zeichenketten und einen Typkonstruktor. Dieser erstellt den Datentyp für einen von der Parser-Methode eingelesenen Namen und je nach Art optional eingelesener Genauigkeit.

Operatoren

In Scala werden Operatoren als Methoden auf dem linksseitigen Typ definiert. Der Operator + ist beispielsweise als überladene Methode der Klasse `Int` definiert. Ähnlich lassen sich Operatoren in den Unterklassen von `DataType` definieren. Dabei ist zu beachten, dass ein Operator dort nicht implementiert werden muss. Stattdessen muss nur die Signatur als rechtsseitiger Typ und Ergebnistyp spezifiziert werden. Dazu wird zunächst die Basisklasse erweitert.

```
1 abstract class DataType extends AST {
2   type OP = (List[String], DataType, DataType)
3   def operators: List[OP]
```

Beispiel 6.21: Schnittstelle für Operatoren

Da nur Signaturen benötigt werden, die beispielsweise für =, <, <=, > und >= gleich sind, kann einer Operatorsignatur vom Typ `OP` mehrere Namen für einen

rechtsseitigen Typ und Ergebnistyp zugewiesen werden. Die Operatoren eines Datentyps werden dann als Liste solcher Signaturen definiert.

Über die Liste der Signaturen lässt sich eine Methode implementieren, die für einen gegebenen Namen eines Operators und einen gegebenen rechtsseitigen Typ die bestmögliche Variante eines überladenen Operators liefert.

Dabei werden zunächst die Operatorsignaturen auf passenden Namen und rechtsseitige Typen gefiltert. Da durch Überladung mit unterschiedlichen Datentypen derselben Hierarchie ggf. mehrere Signaturen zutreffen, wird der speziellste rechtsseitige Typ ermittelt. Falls ein solcher Typ existiert, wird die passende Operatorsignatur nachgeschlagen.

```
4  def findOperator(op: String, rhs: DataType): OP = {
5   val matching = operators.filter(_ match {
6    case (ops, rhs2, _) => ops.contains(op) && rhs2.isSuperTypeOf(rhs)
7   })
8   val optBestRhs = mostSpecificType(matching.map(_._2))
9   optBestRhs.flatMap { bestRhs =>
10   matching.find(_._2 == bestRhs).map { (ops, rhs, result) =>
11    (ops, rhs, result.asNullable(nullable || rhs.nullable))
12    }
13   }
14  }
15 }
```

Beispiel 6.22: Nachschlagen von Operatoren

Falls eine der beiden Seiten Nullwerte annehmen kann, wird der Ergebnistyp der Signatur in den dazugehörigen Typ mit Nullwerten überführt. Andernfalls werden auch für den Ergebnistyp keine Nullwerte zugelassen.

Typberechnung

Über eine Schnittstelle für typisierbare Elemente des abstrakten Syntaxbaums, lässt sich eine Methode fordern, die für übergebene Umgebungsinformationen der Abfrage versucht, den Typ zu berechnen. Die Umgebung enthält Informationen, die aus der Analyse des Tabellenausdrucks einer Abfrage gewonnen wurden. Für die Berechnung von Typen ist die enthaltene Symboltabelle der Spalten notwendig, um die Datentypen von in Ausdrücken verwendeten Spalten nachzuschlagen.

```
1  trait Typeable[+T] { AST =>
2    def calcType(scope: Env): Compile[T]
3  }
4  abstract class ValueExpression extends AST with Typeable[DataType]
```

Beispiel 6.23: Schnittstelle für Typberechnung

Das Ergebnis der Typberechnung ist entweder ein Typfehler oder ein generischer Typ. Zwar wird die Typvariable von `Typeable` in allen Ausdrücken über Werte mit `DataType` belegt, allerdings muss auch der Typ der Projektion einer Abfrage berechnet werden können. Die Projektion einer Abfrage kann entweder eine Liste von Ausdrücken (jeweils eine Spalte im Ergebnis) und Einträgen der Form `actors.*` (alle Spalten der jeweiligen Tabelle) sein. Alternativ werden bei Abfragen der Form `SELECT * FROM ...` alle Spalten des ganzen Tabellenausdrucks ausgelesen. Beide Varianten sind als Liste von Datentypen typisierbar.

Werden in Abfragen eingebettete Scala-Variablen zunächst nicht berücksichtigt, kann die Berechnung des Typs für einen Ausdruck durch rekursive Berechnung für die inneren Ausdrücke und Kombination zu einem Ergebnistyp implementiert werden.

Bei einem Ausdruck der Form `CAST(<value expression> AS <data type>)` ist der Rückgabetyp direkt gegeben.

```
1  case class Cast(expr: ValueExpression, targetType: DataType) extends ValueExpression {
2    def calcType(scope: Env) = expr.calcType(scope).map { sourceType =>
3      targetType.asNullable(sourceType.nullable))
4    }
5    def toSQL = "CAST(" + expr + " AS " + targetType + ")"
6  }
```

Dennoch muss der Datentyp des inneren Ausdrucks berechnet werden, da für den Ergebnistyp übernommen werden muss, ob dieser Nullwerte annehmen kann.

Bei Prädikaten der Form `<value expression> IS NULL` ist das Ergebnis statisch definiert als boolescher Wert, wobei Nullwerte nicht auftreten können.

```
7  case class IsNull(expr: ValueExpression, not: Boolean) extends Predicate {
8    def calcType(env: Env) = expr.calcType(env).map { _ =>
9      DfltBooleanType.asNullable(false)
10   }
11   def toSQL = expr + " IS" + (if (not) " NOT" else "") + " NULL"
12 }
```

Allerdings muss auch hier der Typ des inneren Ausdrucks validiert werden, um etwaige Typfehler zu entdecken, auch wenn der berechnete Datentyp verworfen werden kann.

Bei der Validierung eines Prädikats der Form `<value expression> BETWEEN <value expression> AND <value expression>` müssen zunächst die Datentypen der inneren Ausdrücke berechnet werden.

```
13 case class Between(expr: ValueExpression, not: Boolean,
14   min: ValueExpression, max: ValueExpression) extends Predicate {
15   def calcType(env: Env) = {
16     TypeCheck.typeAll(List(expr, min, max), env).flatMap { types =>
17       val (exprT, minT, maxT) = (types(0), types(1), types(2))
18       if (exprT.findOperator("<", minT).isDefined) {
19         if (exprT.findOperator("<", maxT).isDefined) {
20           Compiled(DfltBooleanType.asNullable(types.map(_.nullable).reduceLeft(_ || _)))
21         } else {
22           Error(min.pos, "There is no order defined on expression's type" +
23             "%s and maximum's type %s".format(exprT, maxT))
24         }
25       } else {
26         Error(max.pos, "There is no order defined on expression's type" +
27           "%s and minimum's type %s".format(exprT, minT))
28       }
29     }
30   }
31   def toSQL = expr + (if (not) " NOT" else "") + " BETWEEN " + min + " AND " + max
32 }
```

Beispiel 6.24 Berechnungen von Typen

Falls in keinem Ausdruck ein Typfehler aufgetreten ist, wird in den Zeilen 18 und 19 geprüft ob die Datentypen von Minimum und Maximum mit dem Typ des geprüften Ausdrucks vergleichbar sind. Dazu wird nachgeschlagen, ob der Operator `<` auf dem Typ des Ausdrucks und Typen von Minimum und Maximum definiert ist. Falls eine Ordnung auf den Typen vorhanden ist, kann das Prädikat als Wahrheitswert bestimmt werden. Dieser kann genau dann `NULL` als Wert annehmen, wenn ein Typ der inneren Ausdrücke Nullwerte annehmen kann (Zeile 20).

Scala-Variablen

Die Berechnung der Datentypen von Ausdrücken und damit das Validieren von Abfragen sowie das Bestimmen der Ergebnistypen von Abfragen ist wesentlich komplexer, wenn eingebettete Scala-Variablen berücksichtigt werden.

```
1 def loadUser(submittedLogin: String) = {
2   val query = <?sql SELECT * FROM actors
3                             WHERE login = { submittedLogin } ?>
4  ...
5 }
```

Beispiel 6.25 Abfrage mit Scala-Variable

Der Typ der Variable `submittedLogin` in **Beispiel 6.25** kann durch das Typsystem von Scala anhand der Signatur von `loadUser` problemlos als `String` bestimmt werden. Die Berechnung der Typen durch den Scala-Übersetzer erfolgt allerdings nach der Syntaxanalyse (vgl. **Abbildung 3.1**), nach der direkt die Phase des Plug-ins durchläuft. Der Typ von `submittedLogin` ist daher während der Übersetzung der Abfrage nicht bekannt und muss in den Folgephasen des Scala-Übersetzers validiert werden. Dazu wird eine lokale Variable im erzeugten `BagStatement` angelegt, wie in Abschnitt 5.2 schon skizziert wurde. Dazu müssen wiederum die Datentypen berechnet werden, die für den Ausdruck `{ submittedLogin }` in SQL und `submittedLogin` in Scala erwartet werden. Nur dann kann auch die Typberechnung des Plug-ins fortgeführt werden.

Im Fall von **Beispiel 6.25** ist der Gleichheitsoperator auf dem Datentyp `VARCHAR` der Spalte `login` definiert auf `VARCHAR` und `CHAR`. Die beiden Typen werden praktischerweise beide auf `String` in Scala abgebildet, während die Ergebnisse der Operatorenüberladungen praktischerweise in beiden Fällen ein boolescher Wert ist.

Eine eindeutige Bestimmung ist leider nicht immer möglich, wie folgendes Beispiel zeigt.

```
<?sql SELECT * FROM actors WHERE size > { min } ?>
```

Da der Operator `>` für `DECIMAL` (den Typ der Spalte `size`) für alle numerischen SQL-Typen definiert ist, die alle auf unterschiedliche Scala-Typen abgebildet werden, gibt es keinen eindeutigen erwarteten Datentypen für `min` in Scala. Aus den möglichen Varianten wird der allgemeinste Typ ausgewählt, so dass hier für `{ min }` der SQL-Typ `NUMERIC` und für `min` der Scala-Typ `java.lang.Number` erwartet wird. In diesem Fall ist die Wahl des allgemeinsten Typs unproblematisch, da das Ergebnis der Operatoranwendung in allen Fällen einen booleschen Wert liefert.

Ein allgemeinster Typ muss aber nicht jedem Fall existieren, wie das folgende zwar unrealistische, aber mögliche Beispiel zeigt.

```
<?sql SELECT * FROM actors WHERE { x } < { y } ?>
```

Für `{ x }` und `{ y }` sind einerseits das Kreuzprodukt aus allen numerischen Typen als Belegung möglich. Dabei wäre eine Belegung beider Ausdrücke mit `NUMERIC` die allgemeinste Variante. Allerdings ist `<` auch als lexikalische Ordnung über Zeichenketten definiert. Eine Belegung von `{ x }` und `{ y }` mit `VARCHAR` wäre aber weder allgemeiner noch spezieller als die Variante mit `NUMERIC`. Da sich also keine allgemeinste Belegung berechnen lässt, wird ein Typfehler ausgegeben.

Falls dieselbe Scala-Variable mehrfach in einer Anfrage verwendet wird, ist eine Unifikation notwendig.

```
1 val query = <?sql
2 SELECT * FROM actors
3 WHERE size > { x } AND SUBSTR(login, { x }) LIKE { s }
4 ?>
```

Beispiel 6.26 Mehrfache Verwendung einer Variable

Während `{ x }` durch den Vergleich mit der Spalte `size` einerseits jeden numerischen Typ annehmen könnte, wird durch das Auftreten von `{ x }` als zweites Argument für `SUBSTR` der SQL-Typ `INTEGER` und damit ein `x: Int` in Scala erwartet. Für die Variable `s` kann wieder der Scala-Typ `String` erwartet werden.

Der Typ eines komplexen Ausdrucks ist also nicht zwangsläufig eindeutig bestimmbar, sondern abhängig von der Belegung der enthaltenen Scala-Variablen. Die Klasse `ScalaVarBindings` modelliert die Ergebnistypen in Abhängigkeit von der Belegung der Scala-Variablen. Instanzen enthalten die in einem Ausdruck auftretenden Variablen `vars` und eine Abbildung von Belegungen auf Ergebnistypen `options`.

```
1 case class ScalaVarBindings[+T](
2   vars: List[EmbeddedScalaVariable],
3   options: Map[Map[String, DataType], T])
```

Beispiel 6.27 Belegungen von Scala-Variablen

Die Typberechnung für den Ausdruck der `WHERE`-Klausel aus **Beispiel 6.26** erfolgt dann wie in **Abbildung 6.3** dargestellt.

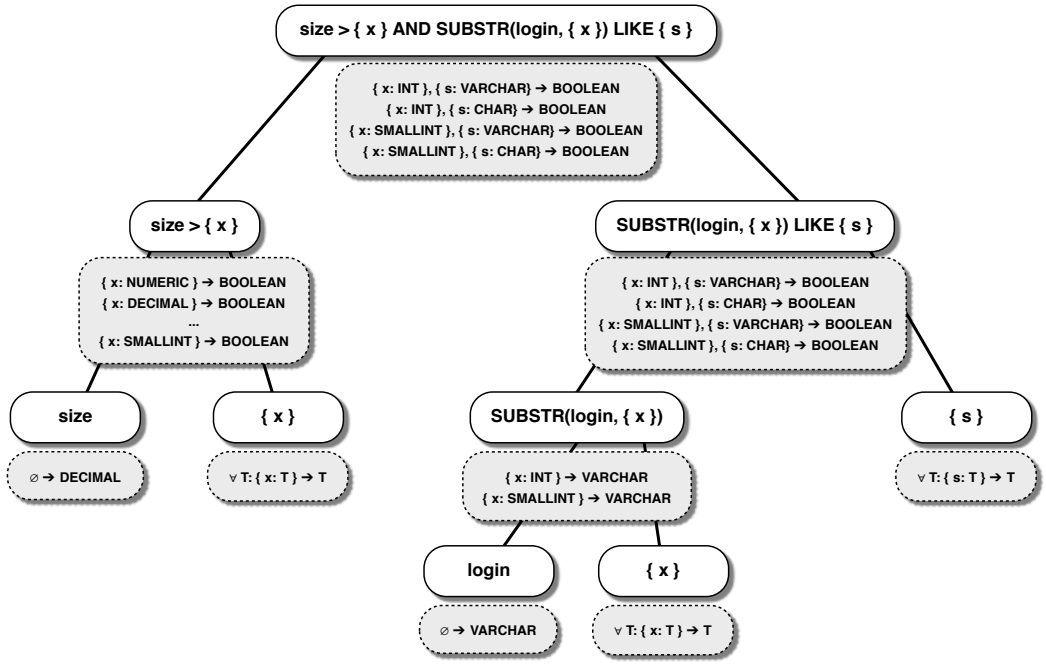


Abbildung 6.3.: Typprüfung mit Variablenbelegungen

Das Ergebnis der Typberechnung entspricht dann dem folgenden Ausdruck unter Verwendung der Klasse `ScalaVarBindings`.

```
ScalaVarBindings(
    List(EmbeddedScalaVariable("{ x }"), EmbeddedScalaVariable("{ s }")),
    Map(
        Map("x" -> DfltIntegerType,
            "s" -> DfltVariableStringType) -> DfltBooleanType,
        Map("x" -> DfltIntegerType,
            "s" -> DfltFixedStringType)    -> DfltBooleanType,
        Map("x" -> DfltShortType,
            "s" -> DfltVariableStringType) -> DfltBooleanType,
        Map("x" -> DfltShortType,
            "s" -> DfltFixedStringType)    -> DfltBooleanType))
```

Die Basisklasse für Ausdrücke wird dementsprechend um Belegungen erweitert. Eine Typberechnung muss nun Ergebnistypen in Abhängigkeit von Belegungen der Scala-Variablen liefern. Dazu müssen alle Klassen für Ausdrücke angepasst werden.

```
1 abstract class ValueExpression extends AST
2   with Typeable[ScalaVarBindings[DataType]]
```

```
3
4  case class OperatorApplication(lhs: ValueExpression, operator: String,
5    rhs: ValueExpression) extends ValueExpression {
6    def calcType(env: Env) = {
7      lhs.calcType(env).flatMap { lhsBindings =>
8        rhs.calcType(env).flatMap { rhsBindings =>
9          permutations(lhsBindings, rhsBindings) { (lhsType, rhsType) =>
10           lhsType.findOperator(operator, rhsType) match {
11             case Some((_, _, result)) => Compiled(result)
12             case _ => Error(pos, "No operator %s is defined on %s" +
13               " and %s".format(operator, lhsType, rhsType))
14           }
15         }
16       }
17     }
18   }
19   def toSQL = lhs + " " + operator + " " + rhs
20 }
```

Beispiel 6.28 Operatorapplikation mit Variablenbelegungen

Die Hilfsfunktion `permutations` konstruiert dabei alle Möglichkeiten aus Kombinationen von Belegungen und führt auf der Basis der daraus resultierenden Typen eine Typprüfung durch. Die Kapselung der Belegungen von Variablen mit Datentypen durch `permutations` hat den Vorteil, dass der zuvor geschriebene Code für die Typprüfung weiterverwendet werden kann.

Falls keine Belegung möglich ist, da eine Variable in zwei zu kombinierenden Belegungen auftritt und nicht unifiziert werden kann, wird ein Typfehler zurückgegeben. In das Ergebnis (neue Belegungen von Scala-Variablen) werden nur die kombinierten Belegungen übernommen, deren Typprüfungen zu einem Ergebnis und keinem Typfehler geführt haben. Führen alle kombinierten Belegungen zu Fehlern, wird versucht, repräsentativ den Typfehler der allgemeinsten Belegung zu ermitteln.

Das implementierte Verfahren ist zugegebenermaßen eine der ineffizientesten Lösungen, die denkbar sind. Allerdings liegt der Fokus an dieser Stelle noch nicht auf der Performanz, sondern auf der Machbarkeit einer solchen Einbettung von Scala-Variablen.

Die Verwendung des allgemeinsten Typs führt dazu, dass bei numerischen Operationen das Ergebnis in der Regel auf `java.lang.Number` abgebildet werden muss. Für `WHERE`-Klauseln, in denen die meisten Scala-Variablen in Abfragen

zu erwarten sind, ist das unproblematisch. Werden Variablen in der Projektion verwendet und beeinflussen dadurch den Scala-Typ einer Abfrage, kann ein Ausdruck der Form `CAST({ any } AS INTEGER)` verwendet werden. Allerdings wird dabei auf die typsichere Übergabe verzichtet, da für `{ any }` jeder Typ und damit `any: Any` in Scala erwartet wird.

6.6. Zielcode

Wenn die Typen in der Projektion einer Abfrage erfolgreich berechnet werden konnten, wird die allgemeinste Belegung der möglicherweise auftretenden Scala-Variablen berechnet. Existiert diese, so kann auch der Zielcode erzeugt werden.

Dazu wird eine dynamische Unterklasse von `BagStatement` erzeugt.

```
1 val query = new scalasql.lib.BagStatement[Tuple6[Long, java.lang.String,
      Option[Array[Byte]], Option[scala.math.BigDecimal], java.lang.
     String]] {
2     val _1: Int = x;
3     val _2: Int = x;
4     val _3: java.lang.String = s;
5     val params = scala.List(_1, _2, _3);
6     val toSQL: java.lang.String = "SELECT * FROM actors\nWHERE size >
         ?   AND SUBSTR(login,   ?  ) LIKE   ?  \n"
7   };
8 }
```

Beispiel 6.29 Zielcode

Die Unterklasse wird parametrisiert mit einem Tupel der Stelligkeit, die der Anzahl der projizierten Spalten entspricht. Das Tupel selbst wird mit den Scala-Typen der einzelnen Spalten parametrisiert.

Für jedes Auftreten einer Scala-Variable wird eine lokale unveränderliche Variable angelegt, deren Typ als der berechnete erwartete Typ deklariert wird. Der Wert dieser Variable wird auf die an die Abfrage zu übergebende Variable gesetzt. Damit die Variablen von der Oberklasse `Statement` an *JDBC* übergeben werden können, wird der Wert von `params` auf die Liste dieser lokalen Variablen gesetzt.

In der ursprünglichen Abfrage müssen die Scala-Variablen durch den Platzhalter `?` ersetzt werden. Damit die Länge der SQL-Abfrage gleich bleibt, was bei Laufzeitfehlern hilfreich sein kann, wird der Platzhalter mit Leerzeichen auf die Länge der ersetzten Variable erweitert.

Damit Fehler im erzeugten Code der Abfrage zugeordnet werden können, werden die Positionen im erzeugten Code auf die Position der Abfrage (XML-Prozessoranweisung) im ursprünglichen Quellcode gesetzt. Ausnahme dabei bilden die lokalen Variablen für die Parameter. Diese werden an die Positionen der jeweiligen Ausdrücke in der Abfrage gebunden, was bei falscher Verwendung zu hilfreichen Fehlermeldungen führt.

```
test/scalasql/Test.scala:25: error: type mismatch;
 found   : Int
 required: java.lang.String
WHERE size > { x } AND SUBSTR(login, { x }) LIKE { s }
                                                 ^
one error found
```

7. Fortgeschrittene Abbildungen

Im Folgenden werden einige spezielle Erweiterungen entwickelt, die auf der Basis der zuvor vorgestellten Einbettung realisiert werden können. Sie dienen im Wesentlichen dazu, die unterstützten Abfragen praxisfähiger zu machen.

7.1. IN-Prädikate

Über Prädikate der Form `<value expression> IN (<in predicate value>)` ist es in SQL möglich, zu prüfen, ob der Wert eines Ausdrucks in einer Menge von Werten passenden Typs vorhanden ist. Als Menge können entweder Ausdrücke aufgezählt (`query1` in **Beispiel 7.1**) oder eine Unterabfrage angegeben werden, die eine Spalte des passenden Typs projiziert (`query2`).

```
1 val query1 = <?sql SELECT * FROM actors WHERE login IN ('cwu', 'cwulf', { someLogin }) ?>
2 val query2 = <?sql SELECT * FROM actors WHERE login IN (SELECT parent FROM actor) ?>
```

Beispiel 7.1: `IN`-Prädikate

Auch wenn sich über eingebettete Scala-Variablen wie `{ someLogin }` in **Beispiel 7.1** dynamische Werte in eine solche Liste integrieren lassen, kann nur eine exakte Anzahl von Werten übergeben werden. Es ist nicht möglich, eine dynamische Liste mit einer dynamischen Anzahl an Werten zu übergeben.

Dafür wäre allerdings eine Notation mit einer Scala-Variable für die Liste der Werte vorstellbar.

```
<?sql SELECT * FROM actors WHERE login IN { logins } ?>
```

Da der Datentyp `VARCHAR` der Spalte `login` auf den Scala-Typ `String` abgebildet wird, könnte für `logins` ein Ausdruck vom Typ `Iterable[String]`, der Oberklasse für Datenbehälter in Scala, erwartet werden. Die enthaltenen Werte sollen dann zur Laufzeit als Aufzählung in die Abfrage integriert werden.

Dazu wird zunächst die Syntaxanalyse erweitert, dass in einer weiteren Regel für `<in predicate value>` auch eingebettete Scala-Variablen akzeptiert werden. Die Klasse `EmbeddedScalaVariable` für Scala-Variablen im abstrakten Syntaxbaum erbt dazu zusätzlich von dem `trait InPredicateValue`.

Für diesen Fall muss auch ein zusätzlicher SQL-Datentyp implementiert werden, der für den Wert eines `IN`-Prädikats erwartet werden kann. Für diesen Datentyp gelten spezielle Eigenschaften: Er wird nur für diesen Zweck benutzt, definiert keine Operatoren und kann nicht mit anderen Datentypen verglichen werden. Er kann nicht in der Projektion einer Abfrage (nur in einem `IN`-Prädikat, nicht aber direkt als Ergebnis) auftreten. Die Abbildung auf den Scala-Typ dient also nur der Übergabe an *JDBC*. Der Datentyp ist dynamisch, da der dazugehörige Scala-Typ `Iterable[T]` parametrisiert und abhängig von der Verwendung im `IN`-Prädikat ist.

Diese Eigenschaften können über einen zusätzlichen Typ abgedeckt werden, der für Einzelfälle verwendet werden kann.

```
1 case class SpecialType(scalaType: ScalaType) extends DataType {
2   val nullable = true
3   def asNullable(n: Boolean) = this
4   def isSuperTypeOf(that: DataType) = this.scalaType == that.scalaType
5 }
```

Beispiel 7.2: Datentyp für spezielle Ausdrücke

Der Datentyp `SpecialType` definiert sich durch den bei der Instanziierung übergebenen Scala-Typ, auf den er abgebildet wird. Für `{ login }` kann damit der folgende Datentyp erwartet werden:

```
SpecialType(ScalaType("Iterable", List(ScalaType("String", Nil))))
```

Der Form halber wird der Datentyp immer inklusive Nullwerte deklariert, auch wenn dies nie relevant ist, da er zu allen anderen Datentypen orthogonal ist. Eine Instanz von `SpecialType` ist nur Obertyp bzw. äquivalent zu einer weiteren Instanz, die auf denselben Scala-Typ abgebildet wird.

Es ist über keine Klasse von Java geschweige denn `Iterable` von Scala möglich, die Werte eines `IN`-Prädikats an ein `PreparedStatement` von *JDBC* zu übergeben. Die Werte müssen zur Laufzeit in die Zeichenkette der SQL-Abfrage im `BagStatement` integriert werden.

Dazu wird die Bibliothek um eine Klasse erweitert, die ein `Iterable` dekoriert und die enthaltenen Werte zu Übergabe an *JDBC* präpariert.

```
1 class InPredicateValue[+E](values: Iterable[E]) {
2   def toSQL = values.map(SQLInjection.escape).mkString("(", ",", ")")
3 }
```

Beispiel 7.3: Werte für `IN`-Prädikate

Die Werte werden mit Kommata separiert und mit Klammern versehen. Dabei werden Zeichenketten besonders behandelt, damit *SQL-Injection* verhindert werden kann.

Für `{ login }` in **Beispiel 7.1** wird dann der folgende Typ erwartet.

```
SpecialType(
    ScalaType("scalasql.lib.InPredicateValue",
        List(ScalaType("String", Nil))))
```

Damit der Programmierer dennoch direkt `Iterable` verwenden kann, wird die Klasse `Statement` um eine passende implizite Konversion erweitert.

```
1 abstract class Statement[T](implicit val manifest: Manifest[T]) {
2  ...
3  /* Implementation */
4  ...
5   implicit def iterable2inPredicateValue[E](xs: Iterable[E]) =
6     new InPredicateValue(xs)
7 }
```

Beispiel 7.4: Datentyp für spezielle Ausdrücke

Dadurch dass für jeden Scala-Variable eine lokale Variable im Zielcode eingefügt wird, ist die implizite Konversion immer sichtbar.

Damit Werte vom Typ `InPredicateValue` nicht als Parameter an *JDBC* übergeben werden, was zu einem Fehler führen würde, müssen die Datentypen und die Erzeugung des Zielcodes so erweitert werden, dass unterschiedliche Varianten der Übergabe möglich sind.

```
1 abstract class DataType extends AST {
2  ...
3   val jdbcParameter = true
4 }
```

```
5 case class SpecialType(scalaType: ScalaType) extends DataType {
6  ...
7   override val jdbcParameter = false
8 }
```

Beispiel 7.5: Unterschiedliche Übergabe an *JDBC*

Die Variable `jdbcParameter` definiert für einen Datentyp, ob Werte per Platzhalter `?` an das `PreparedStatement` von *JDBC* übergeben werden sollen. Der Standardwert wird für `SpecialType` überschrieben, so dass ein eingebettete Scala-Variable dieses Typs per Platzhalter `%` und `format` in die Abfrage eingefügt wird.

Für die Abfrage `query1` in **Beispiel 7.1** wird dann der folgende Zielcode erzeugt:

```
1 val query1 = new BagStatement[...] {
2     val _1: InPredicateValue[String] = someLogins;
3     val params = scala.List();
4     val toSQL = "SELECT * FROM actors WHERE login IN %s ".format(_1);
5   };
6 }
```

Beispiel 7.6: Zielcode für `IN`-Prädikat

Dabei ist zu beachten, dass die Länge der Zeichenkette, die zur Laufzeit an *JDBC* gesendet wird, in der Regel nicht mehr der Länge im Quellcode entsprechen kann.

7.2. Anbieterspezifische Konstrukte

Auch wenn der SQL-Standard sehr komplex ist, müssen Anbieter von Datenbanken nicht alle Spezifikationen erfüllen. Daher gibt es auch zahlreiche Unterschiede zwischen verschiedenen Implementierungen des Standards. Sollen beispielsweise Schlüsselwörter oder mit Leerzeichen getrennte Wörter als Bezeichner für Spalten oder Tabellen verwendet werden, gibt es in den gängigen Datenbanksystemen unterschiedliche Notationen:

- `"User Table"."User"` für *PostgreSQL*, *Oracle Database* und weitere
- `‘User Table‘.‘User‘` für *MySQL*
- `[User Table].[User]` für *MS SQL Server*

Werden solche speziellen Bezeichner benötigt, ist also eine Unabhängigkeit vom Anbieter eines Datenbanksystems nicht gegeben.

Ein erster Praxistest des hier vorgestellten Plug-ins mit dem Datenbanksystem *PostgreSQL*, das dafür bekannt ist, dem SQL-Standard sehr konform zu sein, ist sofort an der Unterstützung der Datentypen gescheitert. In *PostgreSQL* existieren anstelle der Typen `CLOB` und `BLOB` die praktisch äquivalenten aber anders benannten Datentypen `TEXT` und `BYTEA`.

Für synthetische Schlüssel werden oftmals fortlaufende Nummern verwendet, die über Sequenzen automatisch vom Datenbanksystem vergeben werden. Über den Ausdruck `NEXT VALUE FOR <sequence generator name>` wird die nächste Nummer der Sequenz transaktionssicher ermittelt. Wird dieser Ausdruck dann als Standardwert für die Schlüsselspalte einer Tabelle definiert, muss die Spalte beim Einfügen in die Tabelle nicht angegeben werden. Das Datenbanksystem trägt automatisch einen eindeutigen Wert ein. *PostgreSQL* stellt allerdings statt der herkömmlichen Notation eine Funktion `nextval('SequenceName')` bereit, welcher der Name einer Sequenz als Zeichenkette übergeben werden muss. Sequenzen können also nicht direkt verwendet werden.

Als Abkürzung stellt *PostgreSQL* allerdings `SERIAL`-Datentypen bereit, die `INTEGER`-Typen entsprechen, für deren Spalten automatisch eine Sequenz und ein Standardwert eingerichtet werden. Allerdings ist auch `SERIAL` kein herkömmlicher SQL-Typ, sondern eine Erweiterung von *PostgreSQL*.

Um das Plug-in praxistauglich zu machen, müssen unterschiedliche Dialekte von SQL unterstützt werden. Eine Erweiterung um spezifische Konstrukte und Datentypen von Anbietern wird im Folgenden am Beispiel von *PostgreSQL* realisiert.

Plug-in-Argument

Zunächst muss es dem Programmierer möglich sein, ein bestimmtes Datenbanksystem zu aktivieren, für das eingebettete Abfragen transformiert werden sollen. Dazu wird ein weiteres Plug-in-Argument für die Kommandozeile vorgesehen.

```
1 class ScalaSQLPlugin(val global: Global) extends Plugin {
2  var vendor: Option[String] = None
3
```

```
4  override def processOptions(options: List[String],
5    error: String => Unit) {
6     for (option <- options) {
7       if (option.startsWith("vendor:")) {
8         vendor = Some(option.substring("vendor:".length))
9       } else if (option.startsWith("schema:")) {
10        schemaFileName = option.substring("schema:".length)
11      } else {
12        error("Option %s not understood.".format(option))
13      }
14    }
15  }
16  override val optionsHelp: Option[String] = Some(
17    "-P:scalasql:vendor:name set the optional vendor to use: pg\n" +
18    "-P:scalasql:schema:file set the file name of the schema to use\n")
19 ...
20 }
```

Beispiel 6.7: Argument für den Anbieter

Über den folgenden Aufruf des Scala-Übersetzers lässt sich dann *PostgreSQL* als Anbieter spezifizieren.

```
~$ scalac -Xplugin:plugin-scalasql.jar
  -P:scalasql:vendor:pg SourceFile.scala
```

Als Konvention kann der Anbieter auch in der Datei mit dem zu verwendenden Schema definiert werden. Dazu muss die erste Zeile der Textdatei einen SQL-Kommentar der Form `-- vendor` enthalten:

```
-- pg
CREATE TABLE scalasql.public.actors (
actor_id BIGSERIAL PRIMARY KEY,
login VARCHAR(128) NOT NULL UNIQUE,
image bytea,
size DECIMAL(5,2) CHECK (size IS NULL OR size > 0),
parent VARCHAR DEFAULT CURRENT_USER NOT NULL,
vcard xml NOT NULL
);
```

Es ist natürlich möglich, sowohl das Argument `-P:scalasql:vendor:` zu übergeben als auch ein Schema mit einer solchen Angabe des Anbieters zu verwenden. In dem Fall wird der im Schema definierte Anbieter verwendet.

Bei Angabe eines Anbieters werden spezifische Parser-Komponenten gesucht, die statt der herkömmlichen Parser für Schemata und Abfragen eingesetzt werden.

Spezifische Parser-Komponenten

Bei der Angabe von `pg` als Anbieter wird geprüft, ob die Parser-Komponenten `pgDDLParser` und `pgQueryExpressionParser` existieren. Diese müssen sich im Paket `scalasql.plugin.parser.vendor` befinden und von den Standardkomponenten für die Syntaxanalyse erben.

In den anbieterspezifischen Parser-Komponenten können durch das gezielte Überschreiben von Methoden neue Konstrukte hinzugefügt oder deaktiviert werden.

```
1 class pgDDLParser extends DDLParser with pgParsers
2 class pgQueryExpressionParser extends QueryExpressionParser
3   with pgParsers
4 trait pgParsers extends ValueExpressionParsers {
5   // override val delimiterStart = "\""
6   // override val delimiterEnd = "\""
7   override def dataTypes = new pgDataTypes
8   override def definedFunctions = new pgFunctions
9 }
```

Beispiel 7.8: Parser-Komponenten für *PostgreSQL*

Die Parser-Komponenten für *PostgreSQL* erben von der jeweiligen Standardkomponente und binden zusätzlich den `trait pgParsers` ein. Ein Überschreiben der Trennzeichen für Bezeichner ist für *PostgreSQL* nicht notwendig, da Anführungszeichen, die von den meisten Datenbanksystemen verwendet werden, als Konvention vorgesehen sind. Über den `trait` werden allerdings die Methoden überschrieben, welche die unterstützten Datentypen und Funktionen definieren.

Spezifische Datentypen

Die Klasse `pgDataTypes` erbt von der Standardklasse und fügt durch Überschreiben von `noLength` neue Deklarationen für Datentypen ohne Genauigkeit

hinzu. Um die Standardtypen `CLOB` und `BLOB` zu deaktivieren, muss die Methode `optionalLength` überschrieben und alle weiterhin unterstützten Datentypen mit optionaler Länge neu deklariert werden.

```
1  class pgDataTypes extends DataTypes {
2   override def noLength = super.noLength ::: noLengthAdditional
3   val noLengthAdditional = List(
4    (List("INT2"),new ShortType(_,false)),
5    (List("SERIAL","INT4","SERIAL4"),new IntegerType(_,false)),
6    (List("BIGSERIAL","INT8","SERIAL8"),new LongType(_,false)),
7    (List("BYTEA"),new BinaryType(_,false,None)),
8    (List("TEXT"),new TextType(_,false,None)),
9    (List("FLOAT4"),new FloatType(_,false,None)),
10   (List("FLOAT8"),new DoubleType(_,false)))
11
12  override def optionalLength = List(
13   (List("CHAR VARYING","CHARACTER VARYING","VARCHAR"), new VariableStringType(_,false,_)),
14   (List("CHARACTER","CHAR"),new FixedStringType(_,false,_)))
15 }
```

Beispiel 7.9: Datentypen für *PostgreSQL*

Dadurch dass `SERIAL` aus Sicht des Plug-ins wie der Datentyp `INTEGER` behandelt werden kann und `TEXT` sowie `BYTEA` als Alias für `CLOB` bzw. `BLOB` interpretiert werden können, müssen keine neuen Klassen für Datentypen implementiert werden.

Im folgenden Abschnitt wird der Dialekt für *PostgreSQL* allerdings um einen neuen Datentyp für XML und dazugehörige Konstrukte erweitert.

7.3. Datentyp für XML-Dokumente

Die Sprache Scala enthält eine hervorragende Unterstützung für die *Extensive Markup Language*. Ohne die Unterstützung von XML-Literalen in Quellprogrammen wäre die Einbettung von SQL-Abfragen in XML-Prozessoranweisungen auch nicht möglich.

Auch vom SQL-Standard wird eine Erweiterung für XML spezifiziert, deren Implementierung aber nicht erforderlich ist, um den Standard zu erfüllen. In *PostgreSQL* existiert allerdings eine native Unterstützung in Form eines eigenen Datentyps `XML` mit einem Prädikat und einer Reihe von Funktionen.

Als Zeichenketten an *JDBC* übergebene XML-Elemente können nicht direkt für Ausdrücke vom Typ `XML` verwendet werden. Die Zeichenketten müssen über die Funktion `XMLPARSE` umgewandelt werden.

```
XMLPARSE(DOCUMENT '<support><version>8.4</version><version>8.3<version></support>')
XMLPARSE(CONTENT '<version>8.4</version><version>8.3<version>')
```

Im Modus `DOCUMENT` wird nur ein XML-Element akzeptiert, das ein gültiges Dokument darstellt. Um eine Sequenz von XML-Elementen zu speichern, kann der Modus `CONTENT` verwendet werden.

Analog dazu enthält die Bibliothek von Scala die Klassen `scala.xml.Elem` und `scala.xml.NodeSeq`. Ziel der Unterstützung von XML im Plug-in ist es, Ausdrücke des Typs `XML` in der Projektion einer Abfrage als `NodeSeq` auszulesen und Scala-Objekte vom Typ `Elem` oder `NodeSeq` als `DOCUMENT` bzw. `CONTENT` an das Datenbanksystem zu übergeben.

Dazu wird das Plug-in um den Datentyp `XML` erweitert, während die Bibliothek um die Projektion und Übergabe von XML-Objekten erweitert wird.

Erweiterung der Datentypen

`XML`-Ausdrücke in SQL sollen direkt als `NodeSeq` ausgelesen werden, was in der Bibliothek realisiert werden kann. Allerdings können Objekte vom Scala-Typ `NodeSeq` nicht direkt an *JDBC* übergeben werden, weshalb eine Hilfsklasse wie `InPredicateValue` im Fall der `IN`-Prädikate benötigt wird. Es ist also in diesem Fall ein Unterschied zwischen dem Scala-Typ der Projektion (*JDBC* an Scala) und dem Typ für eingebettete Scala-Variablen (Scala an *JDBC*) notwendig. Dazu muss zunächst die Basisklasse für Datentypen angepasst werden.

```
1 abstract class DataType extends AST {
2  ...
3   def scalaType: ScalaType
4   def parameterType = scalaType
5 }
```

Beispiel 7.10: Anpassung der Basisklasse für Datentypen

Die Methode `parameterType` definiert den Scala-Typ, der zur Übergabe von Scala-Variablen an *JDBC* verwendet werden soll. Da in der Regel derselbe Typ

für die Projektion verwendet wird, kann `scalaType` als Standardwert definiert werden.

Für den neuen XML-Datentyp wird die Methode `parameterType` dann überschrieben.

```
1  object DfltpgXMLType extends pgXMLType(None, true)
2  class pgXMLType(alias: Option[String], override val nullable: Boolean) extends DataType {
3    override def toSQL = alias.getOrElse("XML")
4    override def asNullable(n: Boolean) = new pgXMLType(alias, n)
5    def isSuperTypeOf(that: DataType) = that.isInstanceOf[pgXMLType] && ratherNullable(that)
6
7    val scalaType = createScalaType("scala.xml.NodeSeq")
8    override def parameterType = createScalaType("scalasql.lib.XMLParameter")
9    override val jdbcParameter = false
10 }
```

Beispiel 7.11: Datentyp für XML

Die Unterscheidung zwischen Dokumenten und Sequenzen soll in der Klasse `XMLParameter` erfolgen, die in der Bibliothek implementiert wird. Da auch diese Klasse nicht von *JDBC* unterstützt werden kann, werden deren Instanzen analog zu `InPredicateValue` direkt als Zeichenkette in die Abfrage integriert und dazu `jdbcParameter = false` konfiguriert.

Auf der Basis des neuen Datentyps lassen sich Funktionen von *PostgreSQL* über XML-Ausdrücke unterstützen. Es lässt sich allerdings auch das Prädikat der Form `<value expression> IS [NOT] DOCUMENT` implementieren. Das Prädikat prüft, ob es sich bei einem XML-Ausdruck um ein Dokument oder eine Sequenz von XML-Elementen handelt.

Dazu muss die Parser-Komponente für *PostgreSQL* angepasst werden.

```
1  trait pgParsers extends ValueExpressionParsers {
2   ...
3    override def postfixPredicate(expr: ValueExpression) =
4      docPredicate(expr) | super.postfixPredicate(expr)
5
6    def docPredicate(expr: ValueExpression): Parser[ValueExpression] =
7      "IS" ~ opt("NOT") ~ "DOCUMENT" @@ {
8        case _ ~ not ~ _ => IsDocument(expr, not.isDefined)
9      }
10 }
```

Beispiel 7.12: Erweiterte Parser-Komponente für *PostgreSQL*

Für das Prädikat wird eine zusätzliche Parser-Methode implementiert. Diese wird als neue Regel zu den herkömmlichen Regeln für Prädikate hinzugefügt.

Damit der neue Datentyp für XML genutzt werden kann, muss die Bibliothek um die Klasse `XMLParameter` erweitert werden.

Erweiterung der Bibliothek

Über die Klasse `XMLParameter` werden Objekte der Typen `Elem` und `NodeSeq` unterschieden und für die Übergabe an *JDBC* vorbereitet.

```
1 class XMLParameter(value: NodeSeq) {
2  override def toString = {
3   val xmlType = if (param.isInstanceOf[Elem]) "DOCUMENT" else "CONTENT"
4   "XMLPARSE(%s %s)".format(xmlType, SQLInjection.escape(value.toString))
5  }
6 }
```

Beispiel 7.13: *Decorator* für XML-Elemente

Das dekorierte XML-Objekt vom Typ `NodeSeq` (schließt als Oberklasse auch `Elem` ein) wird als Zeichenkette konvertiert und als Argument für die Funktion `XMLPARSE` eingesetzt, deren Modus je nach Typ des Objekts ausgewählt wird. Die Zeichenkette muss auch hier behandelt werden, um *SQL-Injection* zu unterdrücken.

Damit statt der Hilfsklasse direkt Objekte der Klassen `Elem` und `NodeSeq` verwendet werden können, wird analog zu `InPredicateValue` eine implizite Konversion in `Statement` implementiert. XML-Objekte lassen sich nun von Scala an eine Abfrage übergeben.

Damit XML-Ausdrücke auch in der Projektion unterstützt werden, muss die in Abschnitt 5.3 vorgestellte generische Projektion erweitert werden.

```
1  def mkExtractor[C](col: Int, manifest: Manifest[_]): ResultSet => C = {
2   if (manifest.erasure == classOf[Option[_]]) { rs =>
3    val valueType = manifest.typeArguments.first
4    val some = Some(mkExtractor(column, valueType)(rs))
5    val result = if (rs.wasNull) None else some
6    result.asInstanceOf[C]
7   } else if (manifest.erasure == classOf[NodeSeq]) { rs =>
8    XML.loadString(rs.getString(column)).asInstanceOf[T]
```

```
9   } else { rs =>
10   rs.getObject(col).asInstanceOf[C]
11  }
12 }
```

Beispiel 7.14: Erweiterung der Projektion

Dazu wird in den Zeilen 7 - 9 von **Beispiel 7.14** eine zusätzliche Regel für den Typ `NodeSeq` eingefügt. Spalten dieses Typs werden als Zeichenkette aus dem Ergebnis der Abfrage extrahiert und über den XML-Parser der Scala-Bibliothek eingelesen.

8. Andere Ansätze und Ausblick

Durch die Notwendigkeit von Datenbanken in der heutigen Software-Entwicklung gibt es eine Reihe anderer Ansätze aus Industrie und Forschung, relationale Datenbanken besser mit Programmiersprachen der *Java Virtual Machine* zu integrieren.

Das Werkzeug SQLJ [9], das als *Object Language Bindings (SQL/OLB)* in den SQL-Standard aufgenommen wurde, kommt dem Konzept dieser Arbeit aus Sicht des benutzenden Programmierers sehr nahe, da SQL-Anfragen direkt in Java-Programme eingebettet werden. Allerdings basiert SQLJ auf einem Präprozessor, für den eine komplette Syntaxanalyse für die syntaktisch komplexe Sprache Java benötigt wird. Die eingebetteten Anfragen müssen gegen ein laufendes Datenbanksystem validiert werden, nicht gegen eine Schemadeklaration. Es wird zwar eine Analyse und Typprüfung der SQL-Abfragen durchgeführt, allerdings keine typsichere Weiterverarbeitung der Ergebnisse unterstützt.

Mit der Persistenzkomponente von *Lift* [1] wird eine typsichere eingebettete Sprache für SQL-Abfragen in Scala bereitgestellt. Allerdings können dort nur SQL-Abfragen unterstützt werden, die in Scala ausgedrückt werden können. Es lässt sich in der Abfragesprache von *Lift* kein relationaler Verbund ausdrücken. Stattdessen muss mit äquivalenten aber ineffizienten Unterabfragen gearbeitet werden, die im besten Fall bei der Optimierung im Datenbanksystem als Verbünde erkannt und effizient ausgewertet werden. Dies ist allerdings nicht bei allen Datenbanksystemen der Fall.

Der Ansatz des Projekts *ScalaQL* [6] kommt dem Konzept dieser Arbeit aus Sicht der Implementierung sehr nahe, da auch dort ein Plug-in für den Scala-Übersetzer eingebettete Anfragen transformiert. Die eingebetteten Anfragen werden allerdings nicht in SQL formuliert, sondern in herkömmlichen Scala-Ausdrücken, die mit einer Annotation versehen werden. Das Plug-in transformiert die annotierten Ausdrücke, die dadurch eine völlig andere Bedeutung

erhalten als die jeweiligen Ausdrücke ohne Annotation. Das ist zwar auch bei den hier vorgestellten Anfragen in XML-Prozessoranweisungen der Fall, allerdings können diese besser durch den Programmierer von herkömmlichen Scala-Ausdrücken unterschieden werden.

Ausblick

Als weitere Arbeit an dem hier vorgestellten Projekt ist zunächst eine höhere Abdeckung des SQL-Standards geplant. Insbesondere die hier nur skizzierte Umsetzung von Anweisungen zum Einfügen, Aktualisieren und Löschen von Daten ist notwendig, damit das Plug-in praxistauglich wird. Auch die restlichen Datentypen des Standards wie Datums- und Zeitangaben sollten umgesetzt werden.

Ferner können weitere fortgeschrittene Abbildungen identifiziert und implementiert werden, um eine nahtlosere Integration von Scala-Programmen und darin enthaltenen SQL-Anfragen zu erreichen. Es wäre beispielsweise möglich, Array-Typen in SQL typsicher auf die jeweiligen Typen in Scala abzubilden.

Auch die Analyse der Tabellenausdrücke von Abfragen kann optimiert und dadurch eine bessere Abbildung auf Scala erreicht werden. Wird statisch erkannt, dass alle Spalten eines Primär- oder anderweitig eindeutigen Schlüssels für einen Tabellenausdruck an Werte gebunden sind, dann kann ein (bis zu) einzeiliges Ergebnis vom Typ `RowStatement` inferiert werden (vgl. Abschnitt 5.3). Der Verbund vom Typ `LEFT [OUTER] JOIN` wird häufig verwendet, um zwei Tabellen zu laden, die in einer *1 : 0,1*- oder *1 : n*-Beziehung stehen. Hier wäre eine Abbildung der Spalten der rechtsseitigen Tabelle auf `Option[T]` bzw. `List[T]` möglich.

9. Zusammenfassung

In dieser Arbeit wurde eine Technik vorgestellt, welche die Einbettung von in SQL geschriebenen Datenbankanfragen in Programme der Sprache Scala ermöglicht. Eingebettete Anfragen werden zur Übersetzungszeit der Scala-Programme analysiert und bezüglich ihrer Typen geprüft. Parameter können typsicher aus dem Scala-Programm an das Datenbanksystem übergeben werden. Die Ergebnisse solcher Abfragen können in Scala-Programmen typsicher weiterverarbeitet werden.

Die eingebetteten Anfragen werden in XML-Prozessoranweisungen codiert, die von Scala nativ unterstützt werden. Solche Ausdrücke lassen sich eindeutig vom herkömmlichen Scala-Code unterscheiden, sind aber dennoch Teil der Syntax von Scala. Dies hat den Vorteil, dass keine Anpassung des Scala-Übersetzers notwendig war. Stattdessen konnte die Einbettung über ein Plug-in für den Scala-Übersetzer realisiert werden, das eingebettete Anfragen identifiziert und transformiert. Die Realisierung als Plug-in bringt im Gegensatz zu einer Umsetzung als Präprozessor eine gewisse Unabhängigkeit von Änderungen an der Sprache Scala mit sich. Nur bei Änderungen in der Sprache, die das Plug-in betreffen, ist eine Anpassung notwendig. Zusätzlich wurde eine Laufzeitbibliothek entwickelt, die auf der Basis von *JDBC* (als Standardtechnik für Datenbankzugriff im Umfeld der *Java Virtual Machine*) Klassen für typsichere SQL-Anfragen bereitstellt.

Repräsentativ wurden im Rahmen der Arbeit Abfragen in SQL implementiert. Auch Anweisungen zum Manipulieren von Daten sind auf der Basis des derzeitigen Plug-ins schnell realisierbar.

Am Beispiel von `IN`-Prädikaten wurde gezeigt, dass sich das Zusammenspiel von Scala und SQL in speziellen Fällen weiter optimieren lässt und damit die sogenannte objektrelationale Unverträglichkeit weiter reduziert werden kann. Zuletzt wurde eine Erweiterung für verschiedene Dialekte von SQL realisiert, um das Plug-in praxistauglicher zu machen. Am Beispiel von *PostgreSQL*

wurden spezielle Datentypen und Konstrukte eines Datenbanksystems unterstützt.

Das hier vorgestellte Plug-in ist noch ausbaufähig, bietet aber dennoch das Potenzial für den Einsatz in Software-Projekten.

A. Installation

Für das Plug-in wird Scala in der Version 2.8 oder höher benötigt. Das Plug-in kann als binäre Distribution als `scalasql-plugin.jar` aus dem Repository[1] heruntergeladen werden.

Um ein Programm mit eingebetteten Anfragen über die Kommandozeile zu übersetzen, muss dem Scala-Übersetzer `scalac` das Plug-in über die Anweisung `-Xplugin` und optional das Schema über die Anweisung `-P:scalasql:schema` übergeben werden. Damit bei der Übersetzung die Klassen der Laufzeitbibliothek bekannt sind, muss die Bibliothek über die Anweisung `-classpath` für den Übersetzer sichtbar gemacht werden.

```
~$ scalac
  -classpath pathToPlugin/plugin-scalasql.jar
  -Xplugin:pathToPlugin/plugin-scalasql.jar
  -P:scalasql:schema:pathToPSchema/schema.sql
   SourceFile.scala
```

Wird ein Programm mit eingebetteten Anfragen gestartet, muss sich das Plug-in ebenfalls im *Classpath* der JVM befinden, damit die Klassen der Bibliothek verwendet werden können.

Verwendung mit *PostgreSQL*

Um Programme zu starten, die mit die Erweiterung für *PostgreSQL* über `-P:scalasql:vendor:pg` transformiert wurden, wird zur Laufzeit der passende *JDBC*-Treiber[2] benötigt.

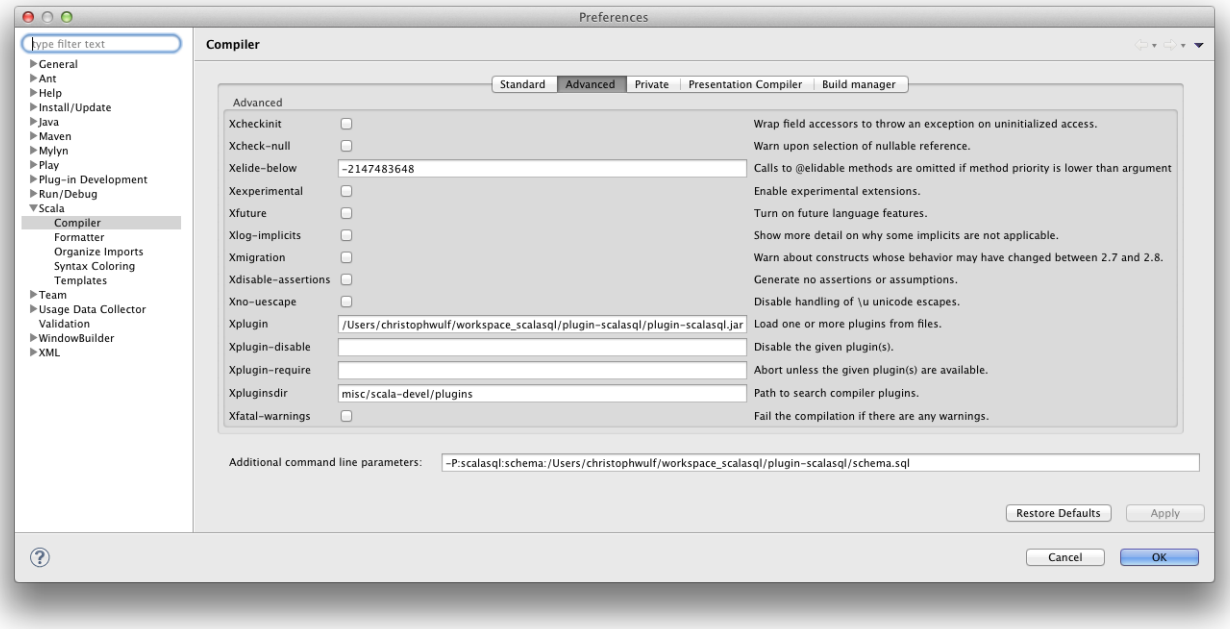
Eingebettete Abfragen können dann für eine Verbindung zu einer Instanz von *PostgreSQL* ausgeführt werden.

[1] `https://redmine.clinical-registry.com/projects/scalasql/repository/`, Ordner `plugin-scalasql/`
[2] `http://jdbc.postgresql.org/`

```
1  object Test {
2    val loginNames = List("cw", "cwu", "cwulf")
3
4    val query = <?sql SELECT actor_id, login FROM actors WHERE login IN { loginNames } ?>
5
6    def main(args: Array[String]) {
7      Class.forName("org.postgresql.Driver")
8      val connection = java.sql.DriverManager.getConnection(
9        "jdbc:postgresql://localhost/scalasql",
10       "user",
11       "password")
12     val result = query.execute(connection)
13     result.foreach {
14       case (id, login) => println("User %d has got login %s".format(id, login))
15     }
16     connection.close
17   }
18 }
```

Konfiguration für *Eclipse*

Um das Plug-in komfortabel in der Entwicklungsumgebung *Eclipse* zu verwenden, muss diese zunächst um die *Scala IDE*[3] erweitert werden. Dann kann das Übersetzer-Plug-in in den Einstellungen für den Scala-Compiler in *Eclipse* aktiviert werden. Über den Punkt *Additional command line parameters* können Schema und Anbieter spezifiziert werden.



[3] http://scala-ide.org/

B. Grammatik

In diesem Anhang wird der unterstützte Teil der SQL-Grammatik in Backus-Naur-Form angegeben. Die Abschnitte entsprechen der Aufteilung in Parser-Komponenten gemäß Abbildung 6.3.

Bezeichner

```
<identifier> ::= <identifier part> [ { . <identifier part> }...]

<column name list> ::= <column name> [ { <comma> <column name> }... ]
<column name> ::= <identifier>
```

Ausdrücke (Werte)

```
<value expression> ::=
    <bool operator application>
  | <predicate primary>

<predicate primary> ::=
    <exists predicate>
  | <operator application>
  | <value expression primary>
  | <value expression> <postfix predicate>

<value expression primary> ::=
      <parenthesized value expression>
    | <function application>
    | <column reference>
    | <literal>
    | <embedded scala expression>
    | <special sql expression>

<parenthesized value expression> ::= <left paren> <value expression> <right paren>

<column reference> ::= <basic identifier chain>

<basic identifier chain> ::= <identifier chain>

<identifier chain> ::= <identifier> [ { <period> <identifier> }... ]
```

```
<function application> ::= <simple identifier> <left paren> [ <value expression> [ { <
    comma> <value expression> }... ] ] <right paren>

<special sql expression> ::=
    <case expression>
  | <cast specification>
  | <null specification>

<case expression> ::= <case abbreviation> | <case specification>

<case abbreviation> ::=
    NULLIF <left paren> <value expression> <comma> <value expression> <right paren>
  | COALESCE <left paren> <value expression> { <comma> <value expression> }... <right
      paren>

<case specification> ::= <simple case> | <searched case>

<simple case> ::= CASE <case operand> <simple when clause>... [ <else clause> ] END

<case operand> ::= <value expression>

<searched case> ::= CASE <searched when clause>... [ <else clause> ] END

<simple when clause> ::= WHEN <when operand> THEN <result>

<searched when clause> ::= WHEN <search condition> THEN <result>

<when operand> ::= <value expression>

<result> ::= <result expression> | NULL

<result expression> ::= <value expression>

<else clause> ::= ELSE <result>

<cast specification> ::= CAST <left paren> <cast operand> AS <cast target> <right paren>

<cast operand> ::= <value expression>

<cast target> ::= <data type>

<null specification> ::= NULL

<postfix predicate> ::=
  | <null predicate part 2>
  | <between predicate part 2>
  | <in predicate part 2>

<null predicate> ::= <row value predicand> <null predicate part 2>

<null predicate part 2> ::= IS [ NOT ] NULL

<between predicate> ::= <row value predicand> <between predicate part 2>
```

```
<between predicate part 2> ::= [ NOT ] BETWEEN [ ASYMMETRIC | SYMMETRIC ] <row value
    predicand> AND <row value predicand>

<in predicate> ::= <row value predicand> <in predicate part 2>

<in predicate part 2> ::= [ NOT ] IN <in predicate value>

<in predicate value> ::=
    <table subquery>
  | <left paren> <in value list> <right paren>
  | <embedded scala expression>

<in value list> ::= <row value expression> [ { <comma> <row value expression> }... ]

<exists predicate> ::= EXISTS <table subquery>

<literal> ::=
    <character string literal>
  | <numeric literal>
  | <boolean literal>

<boolean literal> ::= TRUE | FALSE | UNKNOWN

<embedded scala expression> ::= <left brace> <scala expression> <right brace>

<table subquery> ::= <subquery>

<subquery> ::= <left paren> <query expression> <right paren>
```

Datentypen

Die Datentypen wurden nicht aus der SQL-Grammatik, sondern aus den Spezifikationen übernommen.

Unterstützt werden:

```
SMALLINT
INTEGER
INT
BIGINT
DOUBLE PRECISION
REAL
FLOAT[(scale)]
CHARACTER LARGE OBJECT[(scale)]
CHAR LARGE OBJECT[(scale)]
CLOB[(scale)]
CHAR VARYING[(scale)]
CHARACTER VARYING[(scale)]
VARCHAR[(scale)]
CHARACTER[(scale)]
CHAR[(scale)]
BINARY LARGE OBJECT[(scale)]
BLOB[(scale)]
NUMERIC[(scale[,precision])]
```

```
DECIMAL[(scale[,precision])]
DEC[(scale[,precision])]
```

Tabellendefinitionen

```
<table definitions> ::= <table definition> [ { <semicolon> <table definition> } ] [ <
    semicolon> ]

<table definition> ::=
  CREATE [ <table scope> ] TABLE <table name> <table contents source>

<table name> ::= <local or schema qualified name>

<table scope> ::= <global or local> TEMPORARY

<global or local> ::= GLOBAL | LOCAL

<table contents source> ::= <table element list>

<table element list> ::= <left paren> <table element> [ { <comma> <table element> }... ] <
    right paren>

<table element> ::=
    <column definition>
  | <table constraint definition>

<column definition> ::=
  <column name> [ <data type> ]
  [ <default clause> ]
  [ <column constraint definition>... ]

<default clause> ::= DEFAULT <default option>

<default option> ::=
    <literal>
  | USER
  | CURRENT_USER
  | CURRENT_ROLE
  | SESSION_USER
  | SYSTEM_USER
  | CURRENT_PATH

<column constraint definition> ::= [ <constraint name definition> ] <column constraint>

<column constraint> ::=
    NOT NULL
  | <unique specification>
  | <references specification>
  | <check constraint definition>

<unique specification> ::= UNIQUE | PRIMARY KEY

<references specification> ::= REFERENCES <referenced table and columns>
```

```
<referenced table and columns> ::= <table name> [ <left paren> <reference column list> <
    right paren> ]

<reference column list> ::= <column name list>

<check constraint definition> ::= CHECK <left paren> <search condition> <right paren>

<search condition> ::= <boolean value expression>

<table constraint definition> ::= [ <constraint name definition> ] <table constraint>

<constraint name definition> ::= CONSTRAINT <constraint name>

<constraint name> ::= <schema qualified name>

<table constraint> ::=
    <unique constraint definition>
  | <referential constraint definition>
  | <check constraint definition>

<unique constraint definition> ::=
    <unique specification> <left paren> <unique column list> <right paren>

<unique column list> ::= <column name list>

<referential constraint definition> ::= FOREIGN KEY <left paren> <referencing columns> <
    right paren> <references specification>

<referencing columns> ::= <reference column list>

<reference column list> ::= <column name list>
```

Abfragepezifikationen

```
<query specification> ::= SELECT [ <set quantifier> ] <select list> <table expression>

<set quantifier> ::= DISTINCT | ALL

<select list> ::= <asterisk> | <select sublist> [ { <comma> <select sublist> }... ]

<select sublist> ::= <derived column> | <qualified asterisk>

<qualified asterisk> ::= <asterisked identifier chain> <period> <asterisk>

<asterisked identifier chain> ::= <asterisked identifier> [ { <period> <asterisked
    identifier> }... ]

<asterisked identifier> ::= <identifier>

<derived column> ::= <value expression> [ <as clause> ]

<as clause> ::= [ AS ] <column name>

<column name> ::= <identifier>
```

```
<table expression> ::=
  <from clause>
  [ <where clause> ]
  [ <having clause> ]

<from clause> ::= FROM <table reference list>

<table reference list> ::= <table reference> [ { <comma> <table reference> }... ]

<table reference> ::= <table primary or joined table>

<table primary or joined table> ::= <table primary> | <joined table>

<table primary> ::=
    <table or query name> [ [ AS ] <correlation name> [ <left paren> <derived column list>
         <right paren> ] ]
  | <left paren> <joined table> <right paren>

<joined table> ::=
    <cross join>
  | <qualified join>
  | <natural join>
  | <union join>

<cross join> ::= <table reference> CROSS JOIN <table primary>

<qualified join> ::= <table reference> [ <join type> ] JOIN <table reference> <join
     specification>

<join type> ::= INNER | <outer join type> [ OUTER ]

<outer join type> ::= LEFT | RIGHT | FULL

<join specification> ::= <join condition> | <named columns join>

<named columns join> ::= USING <left paren> <join column list> <right paren>

<join column list> ::= <column name list>

<join condition> ::= ON <column name list>

<natural join> ::= <table reference> NATURAL [ <join type> ] JOIN <table primary>

<union join> ::= <table reference> UNION JOIN <table primary>

<derived column list> ::= <column name list>

<where clause> ::= WHERE <search condition>

<search condition> ::= <boolean value expression>

<having clause> ::= HAVING <search condition>

<search condition> ::= <boolean value expression>
```

Ausdrücke (Abfragen)

```
<query expression> ::= [ <with clause> ] <query expression body>

<with clause> ::= WITH [ RECURSIVE ] <with list>

<with list> ::= <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
  <query name> [ <left paren> <with column list> <right paren> ]
  AS <left paren> <query expression> <right paren>

<with column list> ::= <column name list>

<query expression body> ::= <non-join query expression>

<non-join query expression> ::=
    <non-join query term>
  | <query expression body> UNION [ ALL | DISTINCT ] <query term>
  | <query expression body> EXCEPT [ ALL | DISTINCT ] <query term>

<non-join query term> ::=
    <non-join query primary>
  | <query term> INTERSECT [ ALL | DISTINCT ] <query primary>

<non-join query primary> ::= <simple table> | <left paren> <non-join query expression> <
    right paren>

<simple table> ::= <query specification>

<query term> ::= <non-join query term>

<query primary> ::= <non-join query primary>
```

Funktionen

Die folgenden Funktionen werden exemplarisch unterstützt.

Funktion	Rückgabetyp
CHAR_LENGTH(CHARACTER VARYING)	INTEGER
CHARACTER_LENGTH(CHARACTER VARYING)	INTEGER
OCTET_LENGTH(CHARACTER VARYING)	INTEGER
BIT_LENGTH(CHARACTER VARYING)	INTEGER
TRIM(CHARACTER VARYING)	entspricht Argumenttyp
TRIM(CHARACTER LARGE OBJECT)	CHARACTER LARGE OBJECT
ABS(NUMERIC)	entspricht Argumenttyp
MOD(NUMERIC, NUMERIC)	entspricht Argumenttypen

Unterstützte Funktionen

C. Erweiterung für PostgreSQL

Über die Erweiterung zur Unterstützung von *PostgreSQL* stehen teils andere, teils zusätzliche Datentypen und Funktionen sowie ein weiteres Prädikat zur Verfügung.

Ausdrücke (Werte)

Da XML als Datentyp unterstützt wird, wird in der Syntaxanalyse auch das Prädikat `xml IS [NOT] DOCUMENT` unterstützt. Dieses prüft, ob es sich bei einem XML-Datum um ein valides Dokument oder um ein Fragment eines XML-Dokuments handelt. Die Regel für Postfix-Prädikate wird dementsprechend erweitert.

```
<postfix predicate> ::=
  | <null predicate part 2>
  | <between predicate part 2>
  | <in predicate part 2>
  | <document predicate part 2>

<document predicate> ::= <row value predicand> <document predicate part 2>

<document predicate part 2> ::= IS [ NOT ] DOCUMENT
```

Datentypen

Anstelle von `CLOB` und `BLOB` werden `TEXT` und `BYTEA` unterstützt.

Zusätzlich unterstützt werden:

```
INT2
SERIAL
INT4
SERIAL4
BIGSERIAL
INT8
SERIAL8
```

FLOAT4
FLOAT8
XML

Funktionen

Die folgenden Funktionen werden zusätzlich unterstützt.

Funktion	Rückgabetyp
CEILING(NUMERIC)	entspricht Argumenttyp
CEIL(NUMERIC)	entspricht Argumenttyp
DEGREES(DECIMAL)	DECIMAL
EXP(NUMERIC)	entspricht Argumenttyp
FLOOR(NUMERIC)	entspricht Argumenttyp
LN(NUMERIC)	entspricht Argumenttyp
LOG(NUMERIC)	entspricht Argumenttyp
LOG(NUMERIC, NUMERIC)	NUMERIC
POWER(DECIMAL, DECIMAL)	DECIMAL
RADIANS(DECIMAL)	DECIMAL
ROUND(NUMERIC)	entspricht Argumenttyp
ROUND(NUMERIC, INTEGER)	NUMERIC
SIGN(NUMERIC)	entspricht Argumenttyp
SETSEED(DECIMAL)	INTEGER
SQRT(NUMERIC)	entspricht Argumenttyp
TRUNC(NUMERIC)	entspricht Argumenttyp
TRUNC(NUMERIC, INTEGER)	NUMERIC
SUBSTR(VARCHAR, INTEGER)	CHARACTER VARYING
SUBSTR(VARCHAR, INTEGER, INTEGER)	CHARACTER VARYING
SUBSTR(CLOB, INTEGER, INTEGER)	CHARACTER LARGE OBJECT

Unterstützte Funktionen von *PostgreSQL*

Funktion	Rückgabetyp
XMLCOMMENT(CHARACTER VARYING)	XML
XMLCONCAT(XML, ...)	XML
XMLFOREST(ANY, ...)	XML

Unterstützte Funktionen von *PostgreSQL*

Die XML-Funktionen `XMLCONCAT` und `XMLFOREST`, die per Spezifikation eine beliebige Anzahl von Argumenten erwarten, werden zunächst nur mit bis zu 10 Argumenten unterstützt.

Funktion	Rückgabetyp
`ACOS(DOUBLE PRECISION)`	`DOUBLE PRECISION`
`ASIN(DOUBLE PRECISION)`	`DOUBLE PRECISION`
`ATAN(DOUBLE PRECISION)`	`DOUBLE PRECISION`
`ATAN2(DOUBLE PRECISION)`	`DOUBLE PRECISION`
`COS(DOUBLE PRECISION)`	`DOUBLE PRECISION`
`COT(DOUBLE PRECISION)`	`DOUBLE PRECISION`
`SIN(DOUBLE PRECISION)`	`DOUBLE PRECISION`
`TAN(DOUBLE PRECISION)`	`DOUBLE PRECISION`

Unterstützte trigonometrische Funktionen

Literaturverzeichnis

[1] Chen-Becker, D., Danciu, M., and Weir, T. *The Definitive Guide to Lift*. Apress, 2009.

[2] Codd, E. F. A relational model of data for large shared data banks. *Communications ACM 13*, 6 (1970), 377–387.

[3] Emir, B., Odersky, M., and Williams, J. Matching objects with patterns. In *ECOOP 2007 – Object-Oriented Programming, volume 4609 of LNCS* (2007), Springer, pp. 273–298.

[4] et.al., M. O. An overview of the scala programming language. Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique FÈdÈrale de Lausanne (EPFL), 2006.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman, 1995.

[6] Garcia, M., Izmaylova, A., and Schupp, S. Extending scala with database query capability. *Journal of Object Technology 9(4)* (Juli 2010), 45–68.

[7] Ireland, C., Bowers, D., Newton, M., and Waugh, K. A classification of object-relational impedance mismatch. In *DBKDA '09. First International Conference on Advances in Databases, Knowledge, and Data Applications* (March 2009), pp. 36–43.

[8] Klein, H.-J. Null values in relational databases and sure information answers. In *Semantics in Databases, LNCS 2582*, K.-D. S. L. Bertossi, G. Katona and B. Thalheim, Eds. Springer, 2002, pp. 102–121.

[9] Melton, J., and Eisenberg, A. *Understanding SQL and Java Together*. Morgan Kaufmann, 2000.

[10] Wulf, C. Type-safe sql embedded in scala. Vorgestellt auf den Scala Days 2010, http://days2010.scala-lang.org, April 2010.