Christian Breimann

Anfragebearbeitung in einem Datenbank-Kernsystem für Geo-Anwendungen unter Verwendung einer generischen Komponente zur Anfrageoptimierung

2004

Fach Informatik

Anfragebearbeitung in einem Datenbank-Kernsystem für Geo-Anwendungen unter Verwendung einer generischen Komponente zur Anfrageoptimierung

Inaugural-Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften im Fachbereich Mathematik und Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Christian Breimann
aus Datteln

2004

Dekan:	Prof. Dr. Klaus H. Hinrichs
Erster Gutachter:	Prof. Dr. Klaus H. Hinrichs
Zweiter Gutachter:	Prof. Dr. Guido Wirtz
Tag der mündlichen Prüfung:	09.07.2004
Tag der Promotion:	21.07.2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Beispielverzeichnis

Kurzfassung

Datenbanksysteme kommen heutzutage in vielen Anwendungsbereichen zum Einsatz. Viele dieser Einsatzgebiete – beispielsweise im Umfeld objektorientierter Datenbanksysteme – stellen hohe Anforderungen an die Flexibilität und Erweiterbarkeit eines Datenbanksystems. Neben der Erweiterung um neue Datensätze und der Verwendungsmöglichkeit benutzerdefinierter Datentypen müssen etwa auch neue Indexstrukturen zur Beschleunigung der Anfragebearbeitung, anwendungsbezogene Algorithmen und Operationen zur Erzeugung von Anfrageergebnissen sowie neue Schlüsselwörter für die verwendete Anfragesprache zur Berücksichtigung der Eigenheiten einer Anwendungsdomäne in ein bestehendes Datenbanksystem eingebettet werden können.

Die vorliegende Arbeit beschreibt ausgehend von einer Darstellung eines objektorientierten Datenbank-Kernsystems die dort eingesetzten Konzepte zur Anfragebearbeitung. Dabei wird die Erweiterbarkeit des Systems besonders betont, um die Einsatzmöglichkeiten und die Anpassbarkeit auch für bisher nicht vorgesehene Anwendungsbereiche zu verdeutlichen. Ebenso wird bereits zu Beginn der Arbeit die Kooperation der im weiteren Verlauf vorgestellten Komponenten sowie ihre Bedeutung für die Anfragebearbeitung im dargestellten Datenbank-Kernsystem erläutert.

Als erstes werden kurz eine Anfragesprache für die Extraktion und Manipulation von in einer Datenbank gespeicherten Daten sowie eine Definitionssprache für die Beschreibung neuer Klassen aus der Anwendungsdomäne vorgestellt. Beide Sprachen werden bereits in anderen Arbeiten besprochen, sodass im Rahmen dieser Arbeit nur die wesentlichen Eigenschaften thematisiert werden und eine Einordnung bezüglich weiterer Anfragesprachen für Datenbanksysteme erfolgt.

Im Anschluss wird die Ausführungsebene des Datenbank-Kernsystems vorgestellt. Sie enthält beispielsweise die Algorithmen zur eigentlichen Extraktion von Daten aus einer Datenbank sowie zu deren Manipulation und Kombination. Weiterhin steuert sie die Anwendung dieser Verfahren. Daher beschreibt diese Arbeit ausgehend von einer generellen Erläuterung der Konzepte der Ausführungsebene in Datenbanksystemen die hier konkret vorliegenden Algorithmen sowie ihr Zusammenwirken. Diese Kooperation der Algorithmen zur Berechnung eines Anfrageergebnisses kann durch einen Ausführungsplan beschrieben werden, dessen grundsätzlicher Aufbau daher ebenfalls präsentiert wird.

Sowohl Anfragen an das Datenbanksystem als auch Ausführungspläne der Ausführungsebene werden intern durch Ausdrücke jeweils einer Algebra repräsentiert. Die Definition dieser Algebren folgt dabei einem Ansatz, der größtmögliche Erweiterbarkeit nicht nur der eingesetzten Operatoren, sondern auch des zugrunde liegenden Typsystems gewährleistet. Durch die algebraische Repräsentation von Anfragen und Ausführungsplänen wird zudem sichergestellt, dass die fehlenden Schritte zur Bearbeitung einer Anfrage im Wesentlichen aus der Abbildung eines Algebraausdrucks zur Beschreibung eines Ausdrucks der Anfragesprache auf einen Algebraausdruck zur internen Darstellung eines Ausführungsplans bestehen.

Bei dieser Abbildung ist es jedoch aus Effizienzgründen erforderlich, zu einer gegebenen Anfrage einen hinsichtlich der für ihre vollständige Beantwortung benötigten Zeit möglichst guten Ausführungsplan zu erzeugen. Zu diesem Zweck wird eine neue erweiterbare generische Komponente zur Anfrageoptimierung vorgestellt, die erstmals eine einfache Beschreibung sowohl der erforderlichen Transformationsschritte als auch der Optimierungsstrategie durch textuelle Optimierungsregeln ermöglicht. Dadurch werden zum einen eine leichte Änderbarkeit der Optimierungsstrategie und zum anderen eine einfache Berücksichtigung von Erweiterungen der algebraischen Repräsentation im Rahmen des Optimierungsprozesses sichergestellt.

Schließlich werden die wesentlichen Erweiterungsmöglichkeiten bezüglich der einzelnen Komponenten zur Anfragebearbeitung und deren Auswirkungen auf die übrigen Komponenten thematisiert. Eine wesentliche und notwendige Erweiterung liegt dabei in der Formulierung einer möglichst guten Optimierungsstrategie für die bisher vom dargestellten objektorientierten Datenbank-Kernsystem unterstützten Anfragen.

Danksagung

In den vergangenen Jahren haben mich zahlreiche Menschen im beruflichen und privaten Umfeld unterstützt, ohne deren Anteilnahme und Hilfe diese Arbeit sicherlich nicht entstanden wäre.

Prof. Dr. Klaus H. Hinrichs hat durch die Möglichkeit zur Mitarbeit in seiner Arbeitsgruppe und die besondere Wertschätzung und Anerkennung meiner gesamten Tätigkeit die Grundlagen für die Entstehung meiner Arbeit geschaffen. Daneben hat er meine Forschung und meine sonstige Arbeit in den zurückliegenden Jahren stets in vollem Umfang unterstützt und gefördert.

Prof. Dr. Guido Wirtz ist mir während seiner Zeit in Münster stets ein hervorragender Lehrer und Kollege gewesen. Einen Großteil meines im Studium erworbenen Fachwissens verdanke ich seinen lebendigen und humorvoll gestalteten Vorlesungen und Seminaren. Zudem freue ich mich, dass er ohne zu zögern das Korreferat für diese Arbeit übernommen hat.

Dr. Ludger Becker hat mich auf fachlicher und persönlicher Ebene begleitet und meine Fragen stets bereitwillig und ausführlich beantwortet. Daneben hat er mich in jeglicher Hinsicht und trotz seiner oft knappen Zeit nach Kräften unterstützt. Ihm verdanke ich die Motivation für mein Arbeitsgebiet sowie unzählige weitere Erleichterungen, für deren einzelne Würdigung hier kein Raum ist.

Henrik Blunck hat mir durch seine wertvollen Hinweise und Anregungen ebenfalls sehr bei der Erstellung dieser Arbeit geholfen.

Die weiteren Mitglieder unserer Arbeitsgruppe – Evelyn Egelkamp, Michael Jacob, Timo Ropinski und Dr. Jan Vahrenhold – haben mich durch ihre jederzeit offenen Türen sowie zahlreiche Gespräche und Diskussionen in meiner Arbeit unterstützt und mir viele Erleichterungen verschafft.

Weiterhin gebührt mein Dank den ehemaligen Kollegen und allen Studierenden, die mit ihren Arbeiten wesentlich zur Begründung und Weiterentwicklung des OOGDM-Projekts beigetragen haben. Während meiner Arbeit in Münster waren Katrin Boege, Jan Budde, Ralph Carrie, Björn Eilers, Andreas Jäger, Vladislav Melnikov, Iris Puke, Take Ringena, Holger Schmidt und Olaf Ziemann am OOGDM-Projekt beteiligt und haben mir durch fruchtbare Diskussionen neue Ideen und Sichtweisen gezeigt.

Die übrigen Mitarbeiter des Instituts für Informatik, und in besonderem Maße die Mitarbeiter der IVV 5, haben mir immer bei auftretenden Problemen geholfen. Viele von ihnen sind durch ihre freundliche Art und ihre Hilfsbereitschaft zu einem sehr angenehmen Teil meines täglichen Lebens geworden.

Daneben gilt mein besonderer persönlicher Dank meinen Eltern Brigitte und Hubert Breimann, die mich auf meinem ganzen bisherigen Lebensweg begleitet haben. Ihre beständige Unterstützung hat es mir erst ermöglicht, meinen Weg zu gehen und meine Ziele zu erreichen. Weiterhin sind mir Maria und Georg sowie Theresa Kemmer zu sehr guten Freunden geworden, die mir in vielerlei Hinsicht eine Hilfe sind.

Zu guter Letzt wendet sich mein besonderer persönlicher Dank jedoch an meine zukünftige Frau Patricia Kemmer, ohne deren Vertrauen und Liebe ich diese Arbeit sicherlich nicht in dieser Form hätte erstellen können.

Kapitel 1

Einleitung – Überblick über die Anfragebearbeitung in Datenbanksystemen

Datenbanksysteme kommen heutzutage in vielen Anwendungsbereichen zum Einsatz; einen Überblick über ihre verschiedenen Einsatzgebiete liefern beispielsweise Elmasri und Navathe [EN00, 1], Silberschatz *et. al*. [SKS02, 1.1] oder Vossen [Vos99, 1]. Ein *Datenbanksystem* besteht dabei im Wesentlichen aus den *Datenbanken*, die die eigentlichen Daten enthalten, und dem *Datenbankmanagementsystem*, also der Software, die diese Daten verwaltet. Dazu wurden vor allem relationale Datenbanksysteme entwickelt, die ihre Daten in Relationen (in der Praxis auch *Tabellen* genannt) speichern, die jeweils aus Tupeln (in der Praxis auch *Zeilen* genannt) bestehen [SKS02, 3]. Derartige Tupel wiederum können nur Werte zu einfachen Datentypen wie beispielsweise Zeichenketten mit festgelegter maximaler Länge oder Zahlen enthalten [GMUW02, 6.6]. Bereits boolesche Werte mussten über einen langen Zeitraum durch andere Konzepte simuliert werden; ein entsprechender Datentyp wurde von den meisten kommerziellen Datenbankmanagementsystemen erst im vergangenen Jahrzehnt bereitgestellt. In der Vergangenheit wurde vor allem Wert darauf gelegt, dass sich die eigentlichen Datenbestände leicht verändern und abfragen lassen. Allerdings zeigte sich mit dem Beginn des letzten Jahrzehnts, dass auch das Datenbankmanagementsystem selbst veränderbar sein muss [SKS02, 23]. Kommerzielle Systeme haben dem vor allem Rechnung getragen, indem sie ihre verwendeten Datenmodelle durch die Möglichkeit zur Verwendung benutzerdefinierter Datentypen erweiterbar gestaltet haben [SKS02, 25 ff].

Derartige objektrelationale Datenbanksysteme nutzen die Vorteile herkömmlicher relationaler Datenbanksysteme – wie beispielsweise ihre weite Verbreitung, die Möglichkeiten zur adäquaten Präsentation von Daten und ihr sehr gutes Leistungsverhalten im Hinblick auf die benötigte Rechenzeit zur Beantwortung einer Anfrage – aus und ergänzen das zugrunde liegende Datenmodell um objektorientierte Konzepte [Vos99, 9.4]. So ermöglicht etwa die Definition neuer komplexer Datentypen und zugehöriger Werte eine leichte Abbildung von Objektzuständen auf Einträge in einer objektrelationalen Datenbank, indem jedes Attribut der Objekte eine Entsprechung in einem definierten Datentyp findet. Durch die Möglichkeit zur Spezifikation geschachtelter und abgeleiteter Datentypen mit gleichzeitiger Verwendung mengenwertiger Attribute können für analoge Konzepte objektorientierter Programmiersprachen leicht Entsprechungen in der Datenbank definiert werden. Einen weitergehenden Überblick über die Entwicklung und die Funktionalität objektrelationaler Datenbanksysteme geben beispielsweise Elmasri und Navathe [EN00, 13], Silberschatz *et. al.* [SKS02, 9] sowie Vossen [Vos99, 9.4].

Anzumerken bleibt allerdings, dass bei Anbindung eines objektrelationalen Datenbanksystems an ein objektorientiertes Anwendungsprogramm stets eine Abbildung der Objekte aus der Anwendung auf Tabelleninhalte in der Datenbank erfolgen muss, um diese Objekte persistent zu speichern. Zur Vermeidung einer derartigen Abbildung bieten sich objektorientierte Datenbankmanagementsysteme an, die eine direkte Speicherung von Objekten und ihren Beziehungen untereinander ermöglichen [SKS02, 8]. Eine umfassende Einführung in objektorientierte Datenbanksysteme liefert zum Beispiel Heuer [Heu97].

Vor allem im Umfeld objektorientierter Datenbankmanagementsysteme existieren viele Anwendungsbereiche, für die die oben angeführten Erweiterungsmöglichkeiten nicht ausreichen, sodass für bestimmte Einsatzgebiete auch andere Komponenten eines Datenbankmanagementsystems erweiterbar sein müssen.

So erfordert etwa die Speicherung mehrdimensionaler Daten die Bereitstellung räumlicher Indexstrukturen [GG98] und die Verfügbarkeit spezieller Prädikate für die Anfragesprache.

Solche mehrdimensionalen Daten treten zum Beispiel in einem Geographischen Informationssystem [Voi97, 2] auf. Dieses kann etwa der Verwaltung von Daten aus den Geowissenschaften dienen, um beispielsweise meteorologische Simulationen durchzuführen [Voi97, 9.1] oder Navigationssysteme zu realisieren [SKS02, 23.3.3.2]. Die Daten in einem Geographischen Informationssystem liegen entweder als Rasterdaten – etwa als Resultat eines Satellitenfotos – oder in Form von Vektordaten – zum Beispiel durch Konstruktion aus grundlegenden geometrischen Objekten wie Punkten, Liniensegmenten oder Dreiecken – vor. Vektordaten finden sich auch in CAD-Systemen, sodass dort ebenfalls mehrdimensionale Daten zu speichern sind [SKS02, 23.3.3].

Die Verwendung mehrdimensionaler Daten in einem Datenbanksystem umfasst jedoch nicht nur den Einsatz entsprechender Datenstrukturen zur effizienten Verwaltung dieser großen Datenmengen [Vit01], sondern erfordert auch effiziente Algorithmen zur Berechnung von Ergebnissen einer an ein Datenbanksystem gestellten Anfrage. In diesem Zusammenhang ist häufig die Berechnung einer räumlichen Verbundoperation [Bre00, 3] von Interesse. Typische Aufgaben umfassen dabei etwa die Berechnung des Schnitts von zwei in Form von Vektordaten vorliegenden Objektmengen, beispielsweise zur Bestimmung aller Flüsse, die zumindest teilweise innerhalb eines Staates verlaufen, oder die Auswertung von Richtungsprädikaten, um etwa alle östlich einer Stadt verlaufenden Flüsse zu ermitteln. Ein Datenbankmanagementsystem zur Verwaltung mehrdimensionaler Daten muss daher auch geometrische Operationen für sehr große Mengen von Objekten mit komplexen Typen effizient durchführen können. Eine Übersicht über theoretische und praktische Resultate auf diesem Gebiet findet sich zum Beispiel in einer Arbeit von Breimann und Vahrenhold [BV03].

Alle an einem Datenbankmanagementsystem vorgenommenen Erweiterungen müssen auch im Rahmen der gesamten Anfragebearbeitung berücksichtigt werden [SKS02, 23.3]. Da bereits für die zuvor erwähnten beispielhaften Erweiterungen zur Unterstützung mehrdimensionaler Daten eine Vielzahl räumlicher Indexstrukturen und Verfahren zur Auswertung komplexer geometrischer Prädikate zur Verfügung stehen [BV03], ist es ohne genaue Kenntnis aller in Frage kommenden Anwendungsbereiche nicht möglich, bereits während der Entwicklung eines Datenbankmanagementsystems alle erforderlichen Datentypen, Indexstrukturen und Algorithmen zu integrieren.

Diese Arbeit beschreibt wesentliche Anforderungen und Lösungsmöglichkeiten in diesem Bereich unter Zuhilfenahme eines objektorientierten Datenbankkernsystems für Anwendungen in den Geowissenschaften. Es werden dabei alle wesentlichen Aspekte der Anfragebearbeitung betrachtet, wobei jedoch zwei Schwerpunkte gesetzt werden. Zum einen wird dabei die Ausführungsebene, auf der die konkreten Algorithmen zur Berechnung des Anfrageergebnisses angewendet werden, betrachtet. Zum anderen wird eine neue Komponente zur erweiterbaren generischen Anfrageoptimierung vorgestellt, die es ermöglicht, Optimierungsstrategien leicht zu formulieren, zu erweitern und auf andere Datenbankmanagementsysteme zu übertragen. Dazu werden in diesem Kapitel die ersten Grundlagen gelegt.

1.1 Ein Datenmodell für Geo-Datenbanksysteme und seine prototypische Umsetzung

Damit die in dieser Arbeit vorgestellten Ansätze und Ergebnisse nicht losgelöst von einem konkreten Datenbankmanagementsystem beschrieben werden, sondern auch ihre Anbindung an ein konkretes System gezeigt werden kann, wird nachfolgend ein objektorientiertes Datenbankkernsystem beschrieben, das vor allem für Anwendungen in den Geowissenschaften geeignet ist. Abschnitt 1.1.1 beschreibt hierzu das zugrunde liegende Datenmodell, während Abschnitt 1.1.2 eine existierende prototypische Umsetzung genauer darstellt. Dieses Datenbankkernsystem eignet sich besonders gut für die Präsentation der im Rahmen dieser Arbeit entwickelten Ansätze, weil in beiden Fällen die gute Erweiterbarkeit sowie der Einsatz in unterschiedlichen Anwendungsbereichen von besonderer Bedeutung ist.

1.1.1 OOGDM – Ein Datenmodell für Geo-Datenbanksysteme

Mit OOGDM (**O**bject-**O**riented **G**eo-**D**ata **M**odel) stellt Voigtmann ein erweiterbares objektorientiertes Datenmodell für Datenbank-Anwendungen in den Geowissenschaften vor [Voi97, 3 ff].

Eine Kernforderung an OOGDM ist die Bereitstellung wesentlicher Datentypen, die in vielen geowissenschaftlichen Anwendungen eine Rolle spielen. Daher definiert Voigtmann für OOGDM neben grundle-

genden Datentypen für Zahlen und Zeichenketten verschiedene Typen für geometrische Objekte, so zum Beispiel für Punkte, Polylinien und Gitter sowohl im 2- als auch im 3-Dimensionalen [Voi97, 3]. Insgesamt unterscheidet er bei der Definition der Datentypen im Wesentlichen 2- und 3-dimensionale Datentypen sowohl für raster- als auch für vektorbasierte Repräsentationsformen. Daneben zeigt Voigtmann, wie sich geometrische und topologische Prädikate in das Datenmodell einbinden lassen [Voi97, 4]. Zudem unterstützt OOGDM Zeitstempel für Attribute und Objekte, wobei mit dem Gültigkeitszeitraum – der so genannten *valid time* – sowie dem Erfassungszeitpunkt – der so genannten *transaction time* – zwei verschiedene Zeittypen berücksichtigt werden [Voi97, 5].

Ein weiteres Hauptanliegen bei der Definition von OOGDM war seine Erweiterbarkeit, damit dieses Datenmodell in möglichst vielen unterschiedlichen Anwendungsgebieten Verwendung finden kann. So ist es im Vorfeld kaum möglich zu ermitteln, welche speziellen Datentypen spätere Anwendungen benötigen werden [Voi97, 9]. Die objektorientierte Struktur des Datenmodells erleichtert in diesem Zusammenhang wesentlich das Einfügen neuer Datentypen. Boege [Boe02] und Puke [Puk03] verwenden und erweitern beispielsweise OOGDM, um Repräsentationen auf der Grundlage einer Menge diskreter Messungen der Objektstandorte sowie zugehörige Operationen und Prädikate für sich bewegende zeitvariante räumliche Objekte zu definieren.

Aus Sicht der Anfragebearbeitung sind vor allem die ebenfalls schon von Voigtmann definierten Sprachen OOGQL [Voi97, 6.2] und OOGDM-ODL [Voi97, 6.1] als Bestandteil von OOGDM von Interesse. Bei OOGQL handelt es sich um eine Anfragesprache zur Manipulation der in einer Datenbank enthaltenen Objekte, während OOGDM-ODL der Angabe von Klassendefinitionen für auf OOGDM basierende Anwendungen dient. Beide Sprachen werden in Kapitel 2 näher beschrieben.

Für den restlichen Verlauf dieser Arbeit sind jedoch die weiteren Eigenschaften sowie die übrige Ausdruckskraft von OOGDM nicht von Interesse, daher werden sie hier nicht besprochen; eine ausführliche Darstellung findet sich bei Voigtmann [Voi97, 3].

1.1.2 GOODAC – Eine prototypische Umsetzung von OOGDM

Eine Einführung in GOODAC (**G**eo **O**bject-**O**riented **DA**tabase **C**ore) [Voi97, 7] ist jedoch für den weiteren Verlauf dieser Arbeit erforderlich, weil sich diese Arbeit mit wesentlichen Aspekten der Anfragebearbeitung in GOODAC befasst. GOODAC realisiert das Datenmodell OOGDM und soll dabei als implementiertes Kernsystem für Anwendungen in den Geowissenschaften dienen. Dabei ist GOODAC ebenfalls auf Erweiterbarkeit ausgelegt, damit es bei einem Einsatz innerhalb eines konkreten Anwendungsprogramms an die Erfordernisse der entsprechenen Anwendungsdomäne angepasst werden kann [Voi97, 7.1.2]. So können beispielsweise leicht neue Algorithmen zur Berechnung des Anfrageergebnisses auf der Ausführungsebene (siehe auch Abschnitt 3.2) in das System integriert werden [Rin02, 5.2 f]. Eine weitergehende Übersicht der Erweiterungsmöglichkeiten von GOODAC ausgehend von den Ergebnissen dieser Arbeit findet sich in Kapitel 6.

GOODAC besitzt dabei wesentliche Eigenschaften eines Datenbankmanagementsystems; so sind beispielsweise Mechanismen zur Verwaltung von Zugriffsrechten [Mel02, 5.2] ebenso in GOODAC integriert worden wie Constraints und Trigger [DBVH97]. Allerdings sind bisher nur rudimentäre Ansätze zur Anfragebearbeitung in GOODAC vorhanden [Rin02], sodass ein wesentliches Ziel dieser Arbeit darin besteht, die fehlenden Komponenten zu ergänzen. Hierbei wird wiederum viel Wert auf die Erweiterbarkeit gelegt; insbesondere die neu entwickelte Komponente zur erweiterbaren generischen Anfrageoptimierung (siehe auch Kapitel 5) ermöglicht in diesem Zusammenhang nicht nur eine einfache Erweiterung und Veränderung der eingesetzten Optimierungsstrategie unter Berücksichtigung anderer Ergänzungen von OOGDM und GOODAC, sondern kann auch an andere Datenbankmanagementsysteme angebunden werden, um die für GOODAC entworfenen Optimierungstechniken darauf übertragen und mit konkurrierenden Ansätzen vergleichen zu können.

Die Speicherung der von GOODAC zu verwaltenden persistenten Objekte erfolgt durch das objektorientierte Datenbanksystem ObjectStore [Pro03], bei dem es sich gewissermaßen um eine Datenbank-Erweiterung von C++ handelt, sodass ObjectStore Instanzen von C++-Klassen direkt ohne Verwendung einer Anfragesprache speichern und verwalten kann [Vos99, 13.2.2]. Zugriffe auf diese persistenten Objekte werden über Indexstrukturen durchgeführt, deren Verwendung beispielsweise Melnikov [Mel02, 4.4] genauer beschreibt. Darüber hinausgehende Ansätze – wie sie etwa Schiwietz [Sch93] ausgehend von der Analyse verschiedener Zerlegungsverfahren für komplexe Geo-Objekte vorstellt – finden in GOODAC derzeit keine Verwendung.

Benutzergruppen im Rahmen von GOODAC

Im Rahmen von GOODAC werden drei verschiedene Benutzergruppen unterschieden [Voi97, 7.1]. Sie werden an dieser Stelle nur kurz vorgestellt, während ihre Rollen im Rahmen der Anfragebearbeitung innerhalb von GOODAC an den entsprechenden Stellen dieser Arbeit beschrieben werden.

Die (gewöhnlichen) Benutzer kommen nur als reine Anwender mit dem System in Kontakt, indem sie beispielsweise Anfragen an ein Datenbanksystem stellen. In der Regel benutzen sie dazu Anwendungsprogramme, sodass keine direkte Interaktion mit dem System erfolgt.

Derartige Anwendungsprogramme werden von einem Anwendungsentwickler (einem so genannten *application developer* [Voi97, 7.1.2]) entworfen. Dieser muss neben dem Datenmodell OOGDM vor allem die in Kapitel 2 vorgestellten Sprachen OOGQL und OOGDM-ODL beherrschen; er kommt also ebenfalls nur am Rande mit GOODAC in Kontakt. Die aus Sicht der Anfragebearbeitung wichtigsten von ihm durchgeführten Erweiterungen bestehen in der Definition neuer Klassen für Anwendungsprogramme, die während der Anfragebearbeitung zu berücksichtigen sind. Die Auswirkungen seiner Modifikationen beschränken sich in der Regel auf die Anwendungsprogramme in seinem Verantwortlichkeitsbereich.

Ein Systemprogrammierer (ein so genannter *system programmer* [Voi97, 7.1.2]) ist schließlich dafür verantwortlich, tiefgreifende Erweiterungen an der Repräsentationsschicht von GOODAC durchzuführen. Diese Ebene enthält beispielsweise Indexstrukturen und Algorithmen zur Verwaltung und zum Auslesen von Objekten sowie den Systemkatalog und die Anbindung an ObjectStore [Voi97, 7.1.1]. Dabei kann ein Systemprogrammierer beispielsweise neue Indexstrukturen bereitstellen, die verwendete Strategie für die Anfrageoptimierung verändern oder neue Knoten für ausführbare Anfragegraphen (siehe auch Kapitel 3) definieren. Die von ihm durchgeführten Modifikationen haben in der Regel systemweite Auswirkungen, daher benötigt er tiefgreifende Kenntnisse über GOODAC.

Alle drei Benutzergruppen werden im weiteren Verlauf dieser Arbeit erwähnt, sodass ihre Interaktion mit GOODAC auch an unterschiedlichen Beispielen verdeutlicht wird. Einige weitere Aufgaben dieser Benutzer erwähnen zudem Melnikov [Mel02], Ringena [Rin02] und Ziemann [Zie02] im Rahmen ihrer Beschreibung der Restrukturierung und Ergänzung der Paketstruktur von GOODAC.

1.2 Wesentliche Konzepte der Anfragebearbeitung in Datenbanksystemen

In diesem Abschnitt wird die Anfragebearbeitung in Datenbanksystemen thematisiert. Dazu wird zuerst der grundsätzliche Ablauf ausgehend von relationalen Datenbanksystemen beschrieben (Abschnitt 1.2.1), bevor ein Überblick über die Anfragebearbeitung in GOODAC (Abschnitt 1.2.2) gegeben wird, der zudem den Aufbau dieser Arbeit genauer darstellt und motiviert.

1.2.1 Bestehende Ansätze – Anfragebearbeitung in Datenbanksystemen

Die Anfragebearbeitung in relationalen Datenbanksysteme lässt sich nach Silberschatz *et. al.* [SKS02, 13.1] wie in Abbildung 1.1 darstellen. Der Prozess der Anfragebearbeitung wird dabei in drei wesentliche Schritte eingeteilt:

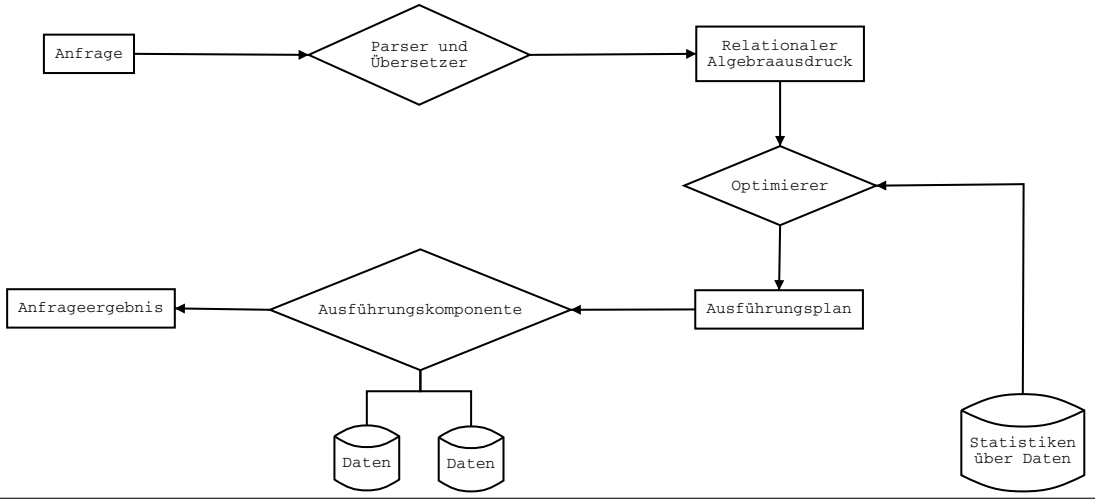
Parsen und Übersetzen: In diesem Schritt wird eine in einer – oft deklarativen – Anfragesprache – beispielsweise SQL – vorliegende Anfrage auf syntaktische Korrektheit untersucht, mit dem Systemkatalog abgeglichen und in ein internes Format – etwa die relationale Algebra – überführt.

Optimierung: Hier wird ein in einem internen Format vorliegender Ausdruck – beispielsweise ein Ausdruck der relationalen Algebra – unter Berücksichtigung unterschiedlicher Faktoren und Konzepte – etwa Heuristiken, Kostenmodelle und Statistiken über frühere Anfragen – optimiert und in einen so genannten Ausführungsplan übersetzt.

Ausführung: Zum Ende der Anfragebearbeitung wird ein Ausführungsplan abgearbeitet. In diesem Schritt werden gemäß der durch den Ausführungsplan gegebenen Beschreibung die angeforderten Daten aus der Datenbank ausgelesen, gegebenenfalls bearbeitet und als Anfrageergebnis zurückgeliefert.

Silberschatz *et. al.* beschreiben diesen Ablauf näher durch Beispiele [SKS02, 13.1], mit deren Hilfe sie zeigen, wie eine in SQL [SKS02, 4] gestellte Anfrage in die relationale Algebra [SKS02, 3] übersetzt und anschließend optimiert wird [SKS02, 14], um den dann erstellten Ausführungsplan abzuarbeiten

Abbildung 1.1 Ablauf der Anfragebearbeitung nach Silberschatz *et. al.* [SKS02, Abbildung 13.1]



[SKS02, 13], damit schließlich das Ergebnis der gegebenen Anfrage zurückgeliefert wird. Daran wird auch eine Aufgabe vor allem SQL-basierter relationaler Datenbanksysteme verdeutlicht, nämlich eine deklarative SQL-Anfrage bereits im Rahmen der Übersetzung in einen Algebraausdruck der relationalen Algebra in eine prozedurale Anfrage überführen zu müssen, um eine Grundlage für die Erzeugung eines generell prozeduralen Ausführungsplans zu schaffen [Vos99, 15].

Dieser grundlegende Ablauf wird auch von anderen Autoren – beispielsweise Boenigk [Boe03, 12.3.1], Connolly und Begg [CB02, 20], Garcia-Molina *et. al.* [GMUW02, 15 f], Kemper und Eickler [KE01, 8] sowie Vossen [Vos99, 15] – in leicht abgewandelter Form beschrieben, sodass er als allgemein anerkanntes Konzept der Anfragebearbeitung zumindest in relationalen Datenbanksystemen gelten kann.

Aber auch die Anfragebearbeitung in Datenbanksystemen eines anderen Typs, also beispielsweise in objektorientierten Datenbanksystemen, folgt häufig diesem Ansatz. So beschreiben Yu und Meng das Konzept der Anfragebearbeitung in objektorientierten Datenbanksystemen als eine Erweiterung des oben vorgestellten Konzepts für relationale Datenbanksysteme, die vor allem die durch das objektorientierte Modell zu berücksichtigenden Erweiterungen gegenüber dem relationalen Modell betrifft [YM98, 2]. Daneben beschreiben Kemper und Moerkotte [KM94] sowie weitere Autoren im Rahmen der Sammlung von Freytag *et. al.* [FMV94] die Anfragebearbeitung in objektbasierten Datenbanksystemen ebenfalls als eine Weiterentwicklung unter Beibehaltung der wesentlichen Ansätze aus relationalen Datenbanksystemen.

Insgesamt kann also festgehalten werden, dass die Anfragebearbeitung sowohl in relationalen als auch in objektorientierten Datenbanksystemen häufig dem in Abbildung 1.1 gezeigten Konzept oder einer entsprechenden Weiterentwicklung folgt.

1.2.2 Neue Ansätze – Anfragebearbeitung in GOODAC

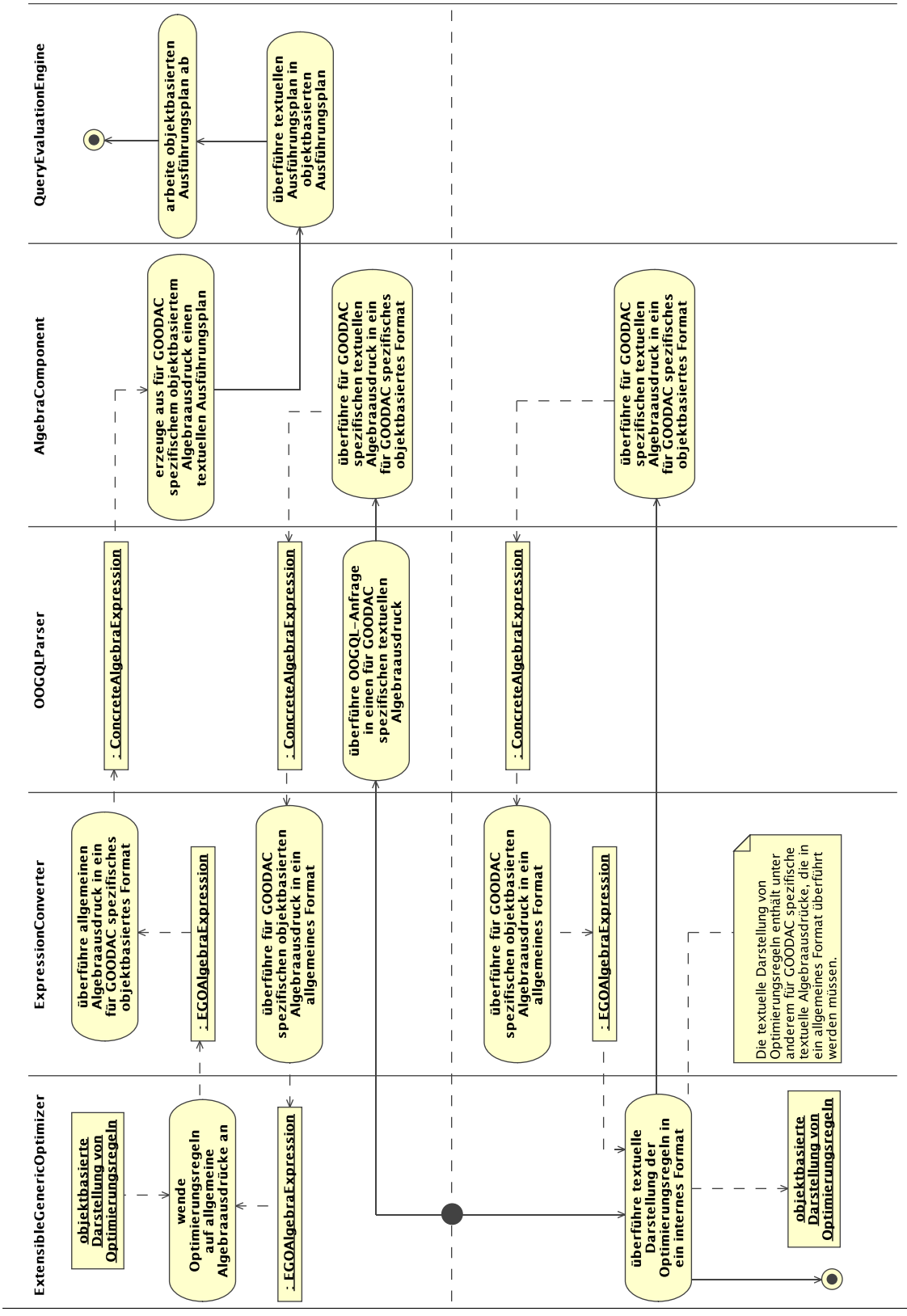
Aus diesem Grund ist auch für GOODAC ein entsprechendes Konzept für die Anfragebearbeitung entwickelt worden. Dieses Vorgehen soll im Folgenden grob beschrieben werden, wobei jeweils Verweise auf die entsprechenden Stellen dieser Arbeit gegeben werden, die sich näher mit den für die einzelnen Schritte der Anfragebearbeitung zuständigen Komponenten beschäftigen. Dadurch wird insbesondere die Bedeutung der nachfolgenden Kapitel für die Anfragebearbeitung in GOODAC herausgestellt.

Grober Verlauf der Bearbeitung einer Anfrage

Der obere Teil des in Abbildung 1.2 gezeigten Aktivitätsdiagramms stellt den Prozess der Anfragebearbeitung in GOODAC graphisch dar. Es wird insbesondere deutlich, dass die Anfragebearbeitung im Wesentlichen von fünf unterschiedlichen Komponenten durchgeführt wird:

AlgebraComponent: Die Algebrakomponente in GOODAC ist für die algebraische Repräsentation von Anfragen verantwortlich. Neben der algebraischen Darstellung von OOGQL-Ausdrücken existiert auch eine Algebra zur Repräsentation von Ausführungsplänen. Diese Algebren werden in Kapitel 4 beschrieben.

Abbildung 1.2 Grobe Skizze des Ablaufs der Anfragebearbeitung in GOODAC



ExpressionConverter: Diese Konvertierungskomponente ist in der Lage, für GOODAC spezifische Algebraausdrücke in ein internes generisches Format – in so genannte *generische Algebraausdrücke* – zu überführen, das von der eingesetzten erweiterbaren generischen Optimierungskomponente verarbeitet werden kann. Ebenso kann die Konvertierungskomponente Ausdrücke aus dem generischen Format in für GOODAC spezifische Algebraausdrücke transformieren. Eine genauere Darstellung findet sich in Abschnitt 5.3.1.

ExtensibleGenericOptimizer: Diese neuartige Komponente sorgt für die Optimierung von Anfragen. Dabei kommt ein regelbasierter Ansatz zum Tragen, der sowohl regelbasierte Transformationen generischer Algebraausdrücke – also der Repräsentation für GOODAC spezifischer Algebraausdrücke im Rahmen des Optimierungsprozesses – als auch eine auf Regeln gegründete Festlegung der Optimierungsstrategie ermöglicht. Die einzusetzenden Regeln können dabei auch zur Ausführungszeit der Optimierungskomponente definiert und verändert werden. Eine ausführliche Präsentation dieses Konzepts und der Komponente EGO (**E**xtensible **G**eneric **O**ptimizer) liefert Kapitel 5. Dort wird insbesondere deutlich, dass sowohl die regelbasierte Formulierung der Optimierungsstrategie als auch die Möglichkeit zur Veränderung der Regelmenge zur Ausführungszeit der Optimierungskomponente eine wesentliche Neuerung gegenüber anderen Ansätzen zur regelbasierten Anfrageoptimierung darstellen.

OOGQL-Parser: Benutzer stellen Anfragen an das Datenbankkernsystem GOODAC in der speziell dafür entwickelten deklarativen Anfragesprache OOGQL. Der OOGQL-Parser ist dafür verantwortlich, diese Anfragen auf Korrektheit zu überprüfen und einen korrespondierenden deskriptiven Algebraausdruck zu erzeugen, den die Algebrakomponente verarbeiten kann. Die Anfragesprache OOGQL und dieser Parser werden in Kapitel 2 näher vorgestellt.

QueryEvaluationEngine: Die Ausführungskomponente für Ausführungspläne ist für den letzten Schritt der Anfragebearbeitung verantwortlich. Sie muss einen vorliegenden textuellen Ausführungsplan verarbeiten und die dort aufgelisteten Operationen in der angegebenen Reihenfolge ausführen, um das Anfrageergebnis zurückzuliefern. Sie wird genauer in Kapitel 3 beschrieben.

Im Aktivitätsdiagramm (siehe Abbildung 1.2) wird weiterhin das Zusammenspiel dieser Komponenten verdeutlicht. Eine genauere Darstellung findet sich in den zuvor angesprochenen weiteren Kapiteln dieser Arbeit. Der obere Teil dieses Aktivitätsdiagramms zeigt, dass eine in OOGQL vorliegende Anfrage zuerst durch den OOGQL-Parser in einen textuellen Algebraausdruck einer für GOODAC spezifizierten Algebra überführt wird. Dabei erfolgt zuerst eine Überprüfung der OOGQL-Anfrage auf syntaktische Korrektheit. Weiterhin wird mit Hilfe des Systemkatalogs eine Typprüfung vorgenommen, bevor das Ergebnis dieses Schritts in Form eines deskriptiven Algebraausdrucks erzeugt und die Arbeit der Algebrakomponente angestoßen wird.

Die Algebrakomponente überführt nun diesen textuell vorliegenden deskriptiven Algebraausdruck in eine objektbasierte Darstellung, die sich zur weiteren Verarbeitung eignet. Diese objektbasierte Repräsentation des deskriptiven Algebraausdrucks wird danach von der Konvertierungskomponente in ein internes generisches Format überführt, auf dem der generische Optimierer arbeiten kann. Als weitere Eingabe erwartet dieser einen Satz von Optimierungsregeln, die sowohl eine Beschreibung der zu verwendenden Strategie als auch die einzusetzenden Transformationsvorschriften enthalten müssen. Nach erfolgter Optimierung überführt die Konvertierungskomponente das Ergebnis, ebenfalls ein Algebraausdruck im generischen Format, wiederum in einen für GOODAC spezifischen objektbasierten ausführbaren Algebraausdruck. Hierbei handelt sich nun allerdings nicht mehr um eine algebraische Repräsentation der ursprünglichen OOGQL-Anfrage, sondern um eine Darstellung eines Ausführungsplans, der die Berechnung des Anfrageergebnisses beschreibt.

Die Algebrakomponente stellt daraufhin sicher, dass aus dieser algebraischen Repräsentation eines Ausführungsplans eine entsprechende textuelle Darstellung erzeugt wird, die die Ausführungskomponente weiterverarbeiten kann. Die Ausführungskomponente ist im Weiteren dafür verantwortlich, diese textuelle Darstellung eines Ausführungsplans wiederum in ein objektbasiertes Format zu überführen, durch das schließlich die Abarbeitung des Ausführungsplans angestoßen wird, um das Anfrageergebnis zurückzuliefern.

Die bei diesem Ablauf auftretenden textuellen Darstellungen von Zwischenergebnissen dienen der besseren Trennung der einzelnen Komponenten, der damit verbundenen einfacheren Definition von Schnittstellen sowie der leichteren Fehlersuche und Weiterentwicklung des Systems. Grundsätzlich wäre es auch

möglich, aus einer bestehenden objektbasierten Darstellung – etwa für eine algebraische Repräsentation eines Ausführungsplans – die nachfolgend benötigte objektbasierte Darstellung – hier beispielsweise für Ausführungspläne – direkt zu erzeugen.

Der zuvor mit Hilfe des in Abbildung 1.2 gezeigten Aktivitätsdiagramms beschriebene Ablauf der Anfragebearbeitung in GOODAC kann auch unter stärkerer Betonung des Zustands, den eine Anfrage durchläuft, erläutert werden. In der zuvor erfolgten Beschreibung wurden hingegen eher die beteiligten Komponenten herausgestellt. Weil zudem im weiteren Verlauf dieser Arbeit oft auf diese unterschiedlichen Zustände eingegangen wird, sollen diese im Folgenden in der zeitlichen Abfolge ihres Auftretens bei der Bearbeitung einer Anfrage nochmals beschrieben werden. In diesem Zusammenhang wird erneut auf die Teile dieser Arbeit verwiesen, in denen eine genauere Beschreibung dieser Zustände erfolgt.

OOGQL-Ausdruck: Ein Benutzer stellt eine Anfrage in Form eines OOGQL-Ausdrucks. Diese Repräsentationsform einer Anfrage stellt daher den üblichen Kommunikationsweg zwischen einem Benutzer und der Komponente zur Anfragebearbeitung in GOODAC dar. Die Anfragesprache OOGQL wird in Kapitel 2 genauer beschrieben.

Textuelle Darstellung eines deskriptiven Algebraausdrucks: Ein deskriptiver Algebraausdruck bildet in seiner textuellen Darstellung die erste interne Repräsentation einer Anfrage. Insbesondere beschreibt er die Anfrage bereits prozedural und liefert damit ein erstes allgemeines Vorgehen zur Ermittlung des Anfrageresultats. Die deskriptive Algebra wird in Kapitel 4 ausführlich beschrieben.

Objektbasierte Darstellung eines deskriptiven Algebraausdrucks: Damit ein deskriptiver Algebraausdruck leicht verarbeitet werden kann, wird er in eine objektbasierte Darstellung überführt. Diese wird ebenfalls kurz in Kapitel 4 thematisiert.

Generischer objektbasierter Algebraausdruck: Ein generischer Algebraausdruck bildet die Grundlage zur Optimierung einer Anfrage. Die hier auftretende Objektdarstellung verwendet ausschließlich Klassen der in dieser Arbeit erstmals beschriebenen Komponente zur erweiterbaren generischen Anfrageoptimierung. Nur auf dieser Repräsentation kann diese generische Optimierungskomponente arbeiten, sodass die objektbasierte Darstellung eines deskriptiven Algebraausdrucks in diese Repräsentation überführt werden muss. Als Ergebnis des Optimierungsprozesses wird ebenfalls ein generischer objektbasierter Algebraausdruck erzeugt, der jedoch einem ausführbaren Algebraausdruck entspricht. Die Klassen zur Erzeugung generischer Algebraausdrücke sowie die eingesetzte neue Komponente zur erweiterbaren generischen Anfrageoptimierung werden in Kapitel 5 vorgestellt.

Objektbasierte Darstellung eines ausführbaren Algebraausdrucks: Diese objektbasierte algebraische Darstellung wird im Anschluss an den Optimierungsprozess erzeugt. Ein derartiger Ausdruck stellt eine algebraische Repräsentation eines textuellen Ausführungsplans dar. Er wird im Rahmen des Optimierungsprozesses anstelle eines Ausführungsplans benutzt, um die Anfrageoptimierung durch die ausschließliche Verwendung von Algebraausdrücken zu vereinfachen. Eine Beschreibung der ausführbaren Algebra liefert Kapitel 4.

Textueller Ausführungsplan: Ein textueller Ausführungsplan wird aus der objektbasierten Darstellung eines ausführbaren Algebraausdrucks erzeugt. Er beschreibt das konkrete Vorgehen zur Ermittlung des Anfrageresultats, indem er die einzusetzenden Algorithmen sowie die Reihenfolge ihrer Anwendung genau auflistet. Textuelle Ausführungspläne eignen sich auch zur Angabe durch den Systemprogrammierer, falls er nach der Integration neuer Algorithmen und Indexstrukturen in GOODAC Testläufe für diese hinzugefügten Bestandteile durchführen möchte. Eine Beschreibung des Aufbaus und der Verwendung textueller Ausführungspläne liefert Kapitel 3.

Objektbasierter Ausführungsplan: Ein objektbasierter Ausführungsplan stellt sein textuelles Pendant intern dar. Die beteiligten Objekte sorgen dafür, dass die für das Anfrageergebnis erforderlichen Daten gemäß des durch den Ausführungsplan spezifizierten Vorgehens aus der Datenbank ausgelesen und verarbeitet werden, um das Anfrageergebnis zu erzeugen. Diese objektbasierten Ausführungspläne werden ebenfalls in Kapitel 3 besprochen.

Anfrageergebnis: Beim Anfrageergebnis handelt es sich schließlich um das Resultat der ursprünglich vom Benutzer gestellten Anfrage. In Kapitel 3 wird erläutert, dass sich ein derartiges Ergebnis beispielsweise wiederum in einer Datenbank speichern oder in textueller Form ausgeben lässt.

Verarbeitung von Regeldefinitionen

Die Erzeugung der zuvor im Rahmen der Beschreibung des Optimierungsprozesses erwähnten Regeln ist im unteren Teil von Abbildung 1.2 dargestellt. Hierbei werden im Wesentlichen textuelle Regeldefinitionen verarbeitet, um eine objektbasierte Repräsentation der Regeln zu erzeugen. Allerdings treten im Rahmen dieser Regeldefinitionen auch für GOODAC spezifische Algebraausdrücke auf – beispielsweise bei Angabe bestimmter algebraischer Transformationsvorschriften –, sodass hier ebenfalls eine Übersetzung in das bereits zuvor angeführte interne generische Format für Algebraausdrücke erfolgen muss.

Zu diesem Zweck überführt die Algebrakomponente einen textuell gegebenen für GOODAC spezifischen Algebraausdruck zuerst in eine objektbasierte Darstellung, bevor die Konvertierungskomponente dafür sorgt, dass dieses Zwischenergebnis in das generische Format überführt wird. Dieser generische Algebraausdruck kann anschließend bei der Erzeugung eines Regelobjekts aus seiner textuellen Definition und der späteren Anwendung der dadurch repräsentierten Regel verwendet werden.

Weiterführende Randaspekte hinsichtlich der Anfragebearbeitung in GOODAC

Im Rahmen dieser Arbeit werden nur die für die Bearbeitung einer einzelnen Anfrage in GOODAC erforderlichen Schritte beschrieben. Daneben spielen in Datenbanksystemen weitere Gesichtspunkte eine Rolle, die im Rahmen von GOODAC an anderer Stelle beschrieben werden oder aber noch nicht entwickelt worden sind. So gibt Melnikov beispielsweise einen Einblick in die Transaktions- und Sitzungsverwaltung [Mel02, 5] im Rahmen von GOODAC. Weitere Konzepte finden sich etwa in den in Abschnitt 1.1.2 zitierten Arbeiten.

Kapitel 2

OOGQL und OOGDM-ODL – Anfrage- und Objektdefinitionssprache für GOODAC

Anfragen in objektorientierten Datenbanksystemen liegen ebenso wie in relationalen [EN00, 8] und objektrelationalen [EN00, 13] Datenbanksystemen häufig in Form eines Ausdrucks einer entsprechenden Anfragesprache (einer so genannten **D**ata **M**anipulation **L**anguage) [SKS02, 1.5] vor. Diese Anfragesprachen stellen die meistgenutzte Schnittstelle zwischen einem Benutzer und dem zum Einsatz kommenden Datenbanksystem dar [GMUW02, 1.2.2]. Weil in den nachfolgenden Kapiteln der Ablauf der Anfragebearbeitung in GOODAC vorgestellt wird, sollen im Folgenden vor allem die in diesem Zusammenhang wesentlichen Aspekte objektorientierter Anfragesprachen – insbesondere der in GOODAC zum Einsatz kommenden Anfragesprache OOGQL – behandelt werden.

Abschnitt 2.1 geht daher näher auf derartige Anfragesprachen ein, bevor in Abschnitt 2.2 die im Rahmen von GOODAC Verwendung findende Anfragesprache OOGQL näher vorgestellt wird. Im Anschluss beschäftigt sich Abschnitt 2.3 näher mit der Übersetzung von OOGQL-Ausdrücken in ein internes Format – eine speziell für GOODAC entwickelte Algebra (siehe auch Kapitel 4). Abschnitt 2.4 beschließt das Kapitel mit einer kurzen Beschreibung der Objektdefinitionssprache OOGDM-ODL und ihrer Bedeutung im Rahmen der Anfragebearbeitung.

Auf eine weitergehende Erläuterung der Sprachen OOGQL und OOGDM-ODL durch Beispiele wird an dieser Stelle aus Platzgründen verzichtet. Umfangreiche Beispiele finden sich bei Voigtmann [Voi97, 7]; auch Eilers [Eil03] und Jäger [Jäg03b] illustrieren ihre Verwendung durch Beispiele.

2.1 Anfragesprachen in objektorientierten Datenbanksystemen

Anfragesprachen in objektorientierten Datenbanksystemen lassen sich nach Heuer [Heu97, 10] im Wesentlichen in drei Kategorien einteilen, wobei Heuer zudem jeweils einige Beispiele für Vertreter dieser Kategorien aufführt:

Algebraische Sprachen: sind in Anlehnung an die Relationale Algebra definiert und verfolgen demzufolge ein prozedurales Konzept zur Formulierung von Anfragen.

SQL-artige Sprachen: erweitern SQL um Konzepte objektorientierter Datenbanksysteme. Manche dieser Anfragesprachen bleiben kompatibel zum eigentlichen SQL-Sprachstandard [SKS02, 4] für relationale Datenbanksysteme, während andere bis auf die Verwendung einiger aus SQL bekannter Schlüsselwörter keine Ähnlichkeit mit dem SQL-Sprachstandard aufweisen.

Regelbasierte Sprachen: stellen eine Möglichkeit zur Formulierung komplexer Anfragen, insbesondere durch die Bereitstellung von Rekursion, dar. Sie bauen häufig auf regelbasierten Anfragesprachen für relationale Datenbanksysteme wie beispielsweise Datalog [GMUW02, 10] auf.

Laut Vossen [Vos99, 13] ist abzusehen, dass sich in diesem Zusammenhang SQL-artige Sprachen – insbesondere OQL [CB00, 4], die etwa von Garcia-Molina *et. al.* ausführlich beschrieben wird [GMUW02,

9] – am weitesten verbreiten werden. Vor allem die Einführung von Rekursion und neuer Mengentypen im Rahmen des am weitesten verbreiteten Standards SQL:1999 [EM99] führt zu einer weiteren Konzentration auf SQL in relationalen Datenbanksystemen. Von dieser herausragenden Bedeutung der relationalen Anfragesprache SQL profitieren demzufolge die SQL-artigen Sprachen in objektorientierten Datenbanksystemen, obwohl beispielsweise OQL in ihrer endgültigen Version auf dem vorherigen Standard SQL:92 basiert [CB00, 4]. Zudem finden immer mehr objektorientierte Konzepte Einzug in SQL, sodass die Verwendung SQL-artiger Sprachen in objektorientierten Datenbankmanagementsystemen zunehmend einfach und selbstverständlich wird. Beispielsweise unterstützt der Standard SQL:1999 bereits Methodendefinitionen, Vererbung und Polymorphie, während der Standard SQL:2003 die Sprache etwa um einen Typ zur Realisierung von Multimengen erweitert [Tür03]. Eine Übersicht über die objektorientierten Eigenschaften bestimmter SQL-Dialekte liefert beispielsweise Petkovic [Pet03].

Sowohl Vossen [Vos99, 13.1] als auch Heuer [Heu97, 6.3.2] stellen zudem einige allgemeine Anforderungen für Anfragesprachen in objektorientierten Datenbanksystemen auf. Ihrer Meinung nach gewährleisten nur Anfragesprachen, die diesen Anforderungen entsprechen, einen optimalen Einsatz der Vorteile eines objektorientierten Datenbanksystems. Die Anforderungen umfassen die folgenden Punkte:

Abgeschlossenheit: Das Ergebnis einer Anfrage kann in einer neuen Anfrage verwendet werden.

Adäquatheit: Alle Eigenschaften des zugrunde liegenden Datenmodells werden berücksichtigt.

Anwendungsunabhängigkeit: Die Anfragesprache ist unabhängig von einzelnen Anwendungsbereichen.

Deskriptivität: Die Sprache hat ein hohes Abstraktionsniveau und unterstützt mengenorientierte Zugriffe.

Effizienz: Jede Operation ist effizient implementierbar, und Berechnungen erfolgen in endlicher Zeit.

Erweiterbarkeit: Jede Erweiterung des zugrunde liegenden Datenmodells lässt sich abbilden.

Formale Semantik: Die Operationen der Sprache sind formal definiert.

Mächtigkeit: Die Sprache überwindet Einschränkungen relationaler Sprachen, etwa der von SQL.

Optimierbarkeit: Die Anfragen lassen sich unter Verwendung von Regeln und Strategien optimieren.

Orthoganilität: Die Grundoperationen sind beliebig miteinander kombinierbar.

Sicherheit: Das Ergebnis jeder Anfrage ist endlich.

Vollständigkeit: Es wird mindestens die Mächtigkeit relationaler Anfragesprachen erreicht.

Eine Vorstellung und Untersuchung existierender Anfragesprachen für objektorientierte Datenbanksysteme bezüglich dieser Konzepte würde an dieser Stelle zu weit führen, zudem liefern Vossen [Vos99, 13.1] und Heuer [Heu97, 10] in diesem Zusammenhang bereits einen Vergleich einiger Anfragesprachen für objektorientierte Datenbanksysteme. Weiterhin diskutiert schon Voigtmann [Voi97, 6.3] die Vor- und Nachteile einiger Anfragesprachen zur Unterstützung geographischer und temporaler Datenmodelle. Er stellt unter anderem fest, dass derartige Anfragesprachen insbesondere Anwendungsunabhängigkeit unterstützen müssen, weil andere Ansätze – wie sie beispielsweise Egenhofer [Ege94] verfolgt – mangels allgemeiner Verwendungsmöglichkeiten nur eine untergeordnete Rolle spielen.

Zhan [Zha94] beispielsweise stellt in diesem Zusammenhang ein Konzept vor, um Anfragesprachen sowohl hinsichtlich ihrer Syntax als auch bezüglich ihrer Semantik einfach spezifizieren und zugehörige Anfragen automatisch in ein internes Format überführen zu können, das von einer bestehenden Komponente zur Anfragebearbeitung weiterverarbeitet werden kann. Allerdings werden an die eingesetzten Spezifikationen sehr strenge Anforderungen gestellt, die nahezu jeglichen Erweiterungen der Komponente zur Anfragebearbeitung entgegenstehen. Zudem wurde das von Zhan vorgestellte Konzept nur in Grundzügen als Aufsatz für einige ausgewählte Datenbankmanagementsysteme implementiert, sodass an eine bestehende Realisierung von Zhans Konzept gestellte Anfragen im Wesentlichen auf Anfragen an das zugrunde liegenden Datenbankmanagementsystem abgebildet werden. In diesem Zusammenhang wurden etwa andere Komponenten der Anfragebearbeitung – zum Beispiel zur Optimierung einer Anfrage oder zur Berechnung des Anfrageergebnisses – unberücksichtigt gelassen. Zhan selbst [Zha94] sieht daher die Untersuchung der Anfrageoptimierung zur Gewährleistung einer tiefergehenden Integration seines Konzepts in die Komponente zur Anfragebearbeitung eines Datenbankmanagementsystem als wesentliches Forschungsvorhaben an.

OQL nimmt unter den SQL-artigen Sprachen neben weiteren Anfragesprachen für objektorientierte Datenbanksysteme wie beispielsweise O^2QL [Kla92] und ESQL2 [GV92] eine besondere Rolle ein, da diese Anfragesprachen keine Sprachkonstrukte im Vergleich zur Anfragesprache SQL für relationale Datenbanksysteme entfernen und somit kompatibel zu diesem Standard bleiben [Heu97, 10.2]. Das führt insbesondere zu einer leichteren Verwendung von Anfragesprachen wie OQL, weil die Menge der zugelassenen Ausdrücke im Vergleich zu SQL lediglich erweitert wird. Einzig und allein diejenigen Elemente aus SQL, die lediglich syntaktischen Zucker (siehe auch Abelson und Sussman [AS85, 1.1.3] in Anlehnung an Landins Einführung dieses Begriffs [Lan98]) darstellen – wie zum Beispiel die explizite Aufforderung zur Durchführung einer Verbundoperation anstelle der reinen Angabe der Eingaberelationen und einer Selektionsbedingung – finden bei der Definition neuer Anfragesprachen kaum Berücksichtigung. Eine ausführliche Beschreibung einer derartigen Ergänzung von SQL um objektorientierte Konzepte liefern beispielsweise van den Bussche und Heuer [vdBH93].

Cattell und Barry [CB00, 4] beschreiben OQL näher und gehen auch auf die oben formulierten Anforderungen ein. Es wird deutlich, dass diese durch OQL weitgehend abgedeckt werden: Alle Anforderungen werden zumindest teilweise verwirklicht. Allerdings verfolgt OQL einen objekterhaltenden und nur eingeschränkt objekterzeugenden Ansatz, inbesondere werden keine dynamische Klassifizierung und keine dynamischen Typen unterstützt [Heu97, 10.2.5].

Diese Eigenschaften beziehen sich auf die Beschreibung des Resultats einer Anfrage; so führen etwa objekterzeugende Ansätze zur Erzeugung neuer Ergebnisklassen für die Beschreibung der ebenfalls neu angelegten Objekte eines Anfrageergebnisses. Die Definition dieser Ergebnisklassen lässt sich gewöhnlich aus den Klassendefinitionen der Ursprungsobjekte ableiten. Objekterhaltende Ansätze lassen die Objektidentität der ursprünglich in der Datenbank gespeicherten Objekte unverändert, sodass beispielsweise bessere Vergleichsmöglichkeiten zwischen dem Anfrageergebnis und den Ursprungsobjekten sowie einfachere Untersuchungen von Veränderungen des Objektzustands gegeben sind. Weiterhin wird unter dynamischer Klassifizierung in diesem Zusammenhang verstanden, dass Objekte im Verlauf einer Anfrage einer anderen als ihrer urspünglichen Klasse zugeordnet werden können, während mit dynamischen Typen die Möglichkeit zur Veränderung des Zustandstyps für Objekte im Anfrageergebnis gemeint ist. Heuer [Heu97, 6.3] gibt eine umfassende Beschreibung der möglichen Eigenschaften von Anfragesprachen sowie Beispiele für objekterzeugende und objekterhaltende Operationen unter Berücksichtigung dynamischer Klassifizierung und dynamischer Typen.

2.2 Die Anfragesprache OOGQL

Die in GOODAC eingesetzte Anfragesprache OOGQL (**O**bject-**O**riented **G**eo **Q**uery **L**anguage) [Voi97, 6.2] stellt eine Erweiterung der OQL zur besseren Unterstützung räumlicher und temporaler Konzepte dar. Sie profitiert von den oben angeführten Vorteilen der OQL, beispielsweise ihrer Orthogonalität, für die unter anderem Heuer [Heu97, 10.2.5] Beispiele anführt.

Voigtmann [Voi97, 6.2 f] diskutiert bereits die Beziehung der Anfragesprache OOGQL zum zugrunde liegenden Datenmodell OOGDM und führt einen Vergleich mit zahlreichen anderen Anfragesprachen für Geoinformationssysteme durch, während schon Riedel [Rie94] im Rahmen der Präsentation der Anfragesprache EXTRACT andere Anfragesprachen in objektorientierten Datenbankmanagementsystemen vor allem hinsichtlich ihrer Einbettung in den gesamten Prozess der Anfragebearbeitung analysiert. Eine weitere Übersicht temporaler Datenbanksysteme und zugehöriger Anfragesprachen liefern zudem Wu *et. al.* [WJW98]. Daher sollen an dieser Stelle nur die seit der von Voigtmann beschriebenen Fassung erfolgten Änderungen der Anfragesprache OOGQL dokumentiert werden.

Diese Änderungen wurden insbesondere erforderlich, da sich im Rahmen der Entwicklung einer Komponente zur lexikalischen und syntaktischen Analyse von OOGQL-Ausdrücken (siehe auch Abschnitt 2.3) zeigte, dass die von Voigtmann entworfene Version der OOGQL einige Mehrdeutigkeiten aufwies oder bestimmte als Beispiel angeführte Ausdrücke [Voi97, 6.4] keine korrekten OOGQL-Ausdrücke darstellten. Auch die Abbildung von Zugriffen auf Kollektionen von Objekten sowie die Darstellung der Erzeugung neuer Kollektionen konnten nicht in der ursprünglichen Fassung umgesetzt werden, sodass an dieser Stelle ebenfalls Änderungen an der von Voigtmann definierten Version der OOGQL erforderlich wurden. Weitere Einzelheiten sind teilweise den Arbeiten von Ringena [Rin02, 5.4], Eilers [Eil03] und Carrie [Car04] zu entnehmen.

Um die für einen effektiven Einsatz der OOGQL im Rahmen der Anfragebearbeitung von GOODAC erforderlichen Änderungen zu dokumentieren, liefern die Abbildungen 2.1, 2.2 und 2.3 einen Überblick

Abbildung 2.1 Produktionen für den Aufbau von OOGQL-Ausdrücken in EBNF, Teil 1

⟨program⟩	::=	⟨query⟩ ';' | ⟨class_name_or_iterator_or_var⟩ ':=' ⟨sfw_query⟩ ';'
⟨query⟩	::=	⟨sfw_query⟩ | ⟨update⟩ | ⟨insertion⟩ | ⟨removal⟩
⟨expression_list⟩	::=	⟨expression⟩ | ⟨expression_list⟩',' ⟨expression⟩
⟨expression⟩	::=	'(' ⟨sfw_query⟩ ')' | ⟨elementary_expr⟩ | ⟨collection_expr⟩ | ⟨construction_expr⟩ | ⟨conversion_expr⟩ | ⟨operation_expr⟩ | ⟨bool_expr⟩ | ⟨number_expr⟩
⟨elementary_expr⟩	::=	⟨class_name_or_iterator_or_var⟩ | ⟨attribute⟩ | ⟨firstlast_element_expr⟩ | ⟨ith_element_expr⟩ | ⟨literal⟩
⟨collection_expr⟩	::=	⟨list_constr_expr⟩ | ⟨bag_constr_expr⟩ | ⟨array_constr_expr⟩ | ⟨set_constr_expr⟩ | ⟨list2set_expr⟩ | ⟨flatten_expr⟩ | ⟨distinct_expr⟩ | ⟨subcollection_expr⟩ | ⟨collection_composition_expr⟩
⟨extended_collection_expr⟩	::=	⟨collection_expr⟩ | ⟨class_name_or_iterator_or_var⟩
⟨collection_composition_expr⟩	::=	'(' ⟨collection_expr⟩ ⟨binary_set_op⟩ ⟨collection_expr⟩ ')'
⟨binary_set_op⟩	::=	BINARY_SET_OP | '+'
⟨sfw_query⟩	::=	⟨select_cmd⟩ ⟨valid_cmd⟩ ⟨from_cmd⟩ ⟨where_cmd⟩ ⟨order_by_cmd⟩ ⟨group_by_cmd⟩
⟨select_cmd⟩	::=	SELECT [SNAPSHOT] [DISTINCT] ⟨select_part⟩
⟨valid_cmd⟩	::=	[VALID ⟨temporal_expression⟩]
⟨from_cmd⟩	::=	FROM ⟨query_range_list⟩
⟨where_cmd⟩	::=	[WHERE ⟨bool_expr⟩]
⟨order_by_cmd⟩	::=	[ORDERBY ⟨order_criteria_list⟩]
⟨group_by_cmd⟩	::=	[GROUPBY ⟨property_list⟩ [HAVING ⟨bool_expr⟩]]
⟨update⟩	::=	APPLY ⟨method_expr_list⟩ [⟨from_cmd⟩ ⟨where_cmd⟩]
⟨insertion⟩	::=	INSERT ⟨object_expr⟩ [⟨from_cmd⟩ ⟨where_cmd⟩] ⟨valid_cmd⟩
⟨removal⟩	::=	DELETE ⟨class_name⟩ ⟨where_cmd⟩ ⟨valid_cmd⟩
⟨select_part⟩	::=	⟨construction_expr⟩ | ⟨collection_expr⟩ | ⟨conversion_expr⟩ | ⟨operation_expr⟩ | ⟨elementary_expr⟩ | ⟨aggregate_expr⟩ | '∗'
⟨construction_expr⟩	::=	⟨object_expr⟩ | ⟨struct_constr_expr⟩
⟨object_expr⟩	::=	⟨class_name_or_user_def_op⟩ '(' [⟨temporal_property_list⟩ | ⟨expression_list⟩] ')'
⟨temporal_property_list⟩	::=	⟨temporal_property_def⟩ | ⟨temporal_property_def⟩ ',' ⟨temporal_property_list⟩
⟨temporal_property_def⟩	::=	IDENTIFIER ':' ⟨expression⟩ ⟨valid_cmd⟩
⟨property_list⟩	::=	⟨class_property_list⟩ | ⟨alt_property_list⟩ | ⟨simple_property_list⟩
⟨class_property_list⟩	::=	⟨class_property_def⟩ | ⟨class_property_def⟩ ',' ⟨class_property_list⟩
⟨class_property_def⟩	::=	IDENTIFIER ':' ⟨expression⟩
⟨alt_property_list⟩	::=	⟨alt_property_def⟩ | ⟨alt_property_def⟩ ',' ⟨alt_property_list⟩
⟨alt_property_def⟩	::=	⟨expression⟩ AS IDENTIFIER
⟨simple_property_list⟩	::=	⟨expression⟩ | ⟨expression⟩ ',' ⟨simple_property_list⟩
⟨class_name⟩	::=	IDENTIFIER
⟨relationship_name⟩	::=	IDENTIFIER
⟨attribute⟩	::=	⟨class_name_or_iterator_or_var⟩ '.' [⟨path⟩ '.'] ⟨attribute_name⟩
⟨path⟩	::=	⟨path_expr⟩ | ⟨path_expr⟩ '.' ⟨path⟩
⟨path_expr⟩	::=	⟨relationship_name⟩ | ⟨method⟩
⟨attribute_name⟩	::=	IDENTIFIER
⟨struct_constr_expr ⟩	::=	STRUCT '(' ⟨property_list⟩ ')'
⟨set_constr_expr⟩	::=	SET '(' ⟨expression⟩ ')'
⟨list_constr_expr⟩	::=	LIST '(' ⟨expression⟩ ')'
⟨bag_constr_expr⟩	::=	BAG '(' ⟨expression⟩ ')'
⟨array_constr_expr⟩	::=	ARRAY '(' ⟨expression⟩ ')'
⟨ith_element_expr⟩	::=	⟨collection_expr⟩ '[' INT_LITERAL ']'
⟨subcollection_expr⟩	::=	⟨collection_expr⟩ '[' INT_LITERAL ':' INT_LITERAL ']'
⟨firstlast_element_expr⟩	::=	FIRST_OR_LAST_ELEM_OP '(' ⟨collection_expr ⟩ ')'
⟨conversion_expr⟩	::=	⟨element_expr⟩ | ⟨typing_expr⟩
⟨element_expr⟩	::=	ELEMENT '(' ⟨collection_expr⟩ ')'

Abbildung 2.2 Produktionen für den Aufbau von OOGQL-Ausdrücken in EBNF, Teil 2

⟨list2set_expr⟩	::=	LISTTOSET '(' ⟨collection_expr⟩ ')'
⟨flatten_expr ⟩	::=	FLATTEN '(' ⟨collection_expr⟩ ')'
⟨distinct_expr⟩	::=	DISTINCT '(' ⟨collection_expr⟩ ')'
⟨typing_expr⟩	::=	CAST '(' ⟨class_name⟩ ')' '[' ⟨expression⟩ ']'
⟨operation_expr⟩	::=	⟨method_expr⟩ | ⟨global_operation_expr⟩ | ⟨temporal_operation⟩
⟨method_expr_list⟩	::=	⟨method_expr⟩ | ⟨method_expr⟩ ',' ⟨method_expr_list⟩
⟨method_expr⟩	::=	⟨class_name_or_iterator_or_var⟩ '.' [⟨path⟩ '.'] ⟨method⟩
⟨method⟩	::=	⟨method_name⟩ '(' [⟨expression_list⟩] ')'
⟨method_name ⟩	::=	IDENTIFIER
⟨global_operation_expr ⟩	::=	⟨operation_name⟩ '(' [⟨expression_list⟩] ')'
⟨literal⟩	::=	LITERAL_OR_CONSTANT | DATE_LITERAL
⟨bool_expr⟩	::=	⟨bool_or_expr⟩
⟨bool_or_expr⟩	::=	⟨bool_xor_expr⟩ | ⟨bool_or_expr⟩ OR_OP ⟨bool_xor_expr⟩
⟨bool_xor_expr⟩	::=	⟨bool_and_expr⟩ | ⟨bool_xor_expr⟩ XOR_OP ⟨bool_and_expr⟩
⟨bool_and_expr⟩	::=	⟨bool_unary_expr⟩ | ⟨bool_and_expr⟩ AND_OP ⟨bool_unary_expr⟩
⟨bool_unary_expr⟩	::=	⟨bool_primary_expr⟩ | BOOL_UNARY_OP ⟨bool_primary_expr⟩
⟨bool_primary_expr⟩	::=	⟨comparison_expr⟩ | ⟨bool_collection_expr⟩ | BOOL_LITERAL | '(' ⟨bool_expr⟩ ')'
⟨number_expr⟩	::=	⟨number_or_expr⟩
⟨number_or_expr⟩	::=	⟨number_xor_expr⟩ | ⟨number_or_expr⟩ NUMBER_OR_OP ⟨number_xor_expr⟩
⟨number_xor_expr⟩	::=	⟨number_and_expr⟩ | ⟨number_xor_expr⟩ NUMBER_XOR_OP ⟨number_and_expr⟩
⟨number_and_expr⟩	::=	⟨number_shift_expr⟩ | ⟨number_and_expr⟩ NUMBER_AND_OP ⟨number_shift_expr⟩
⟨number_shift_expr⟩	::=	⟨number_add_expr⟩ | ⟨number_shift_expr⟩ NUMBER_SHIFT_OP ⟨number_add_expr⟩
⟨number_add_expr⟩	::=	⟨number_mult_expr⟩ | ⟨number_add_expr⟩ NUMBER_ADD_OP ⟨number_mult_expr⟩
⟨number_mult_expr⟩	::=	⟨number_pot_expr⟩ | ⟨number_mult_expr⟩ NUMBER_MULT_OP ⟨number_pot_expr⟩
⟨number_pot_expr⟩	::=	⟨number_unary_expr⟩ | ⟨number_pot_expr⟩ NUMBER_POT_OP ⟨number_unary_expr⟩
⟨number_unary_expr⟩	::=	⟨number_primary_expr⟩ | NUMBER_UNARY_OP ⟨number_primary_expr⟩
⟨number_primary_expr ⟩	::=	⟨aggregate_expr⟩ | INT_LITERAL | FLOATING_PT_LITERAL | '(' ⟨number_expr ⟩ ')'
⟨aggregate_expr⟩	::=	AGGREGATE_OP '(' ⟨expression⟩ ')' | COUNT '(' '∗' ')'
⟨bool_collection_expr⟩	::=	⟨in_set_expr⟩ | ⟨quantification_expr⟩
⟨quantification_expr⟩	::=	⟨forall_expr⟩ | ⟨exists_expr⟩ | ⟨exists_or_all_comp_expr⟩ | ⟨exists_bool_expr⟩
⟨forall_expr⟩	::=	FORALL IDENTIFIER IN ⟨extended_collection_expr⟩ ':' ⟨predicate⟩
⟨exists_expr⟩	::=	EXISTS IDENTIFIER IN ⟨extended_collection_expr⟩ ':' ⟨predicate⟩
⟨exists_or_all_comp_expr⟩	::=	'(' ⟨expression⟩ ⟨comparison_predicate⟩ EXISTS_OR_ALL_COMP_OP ⟨extended_collection_expr⟩ ')'
⟨exists_bool_expr⟩	::=	EXISTS '(' ⟨expression⟩ ')' | UNIQUE '(' ⟨expression⟩ ')'
⟨in_set_expr⟩	::=	⟨elementary_expr⟩ IN ⟨extended_collection_expr⟩
⟨predicate⟩	::=	⟨bool_expr⟩
⟨query_range_list⟩	::=	⟨query_range⟩ | ⟨query_range⟩ ',' ⟨query_range_list⟩
⟨query_range⟩	::=	⟨expression⟩ [[AS] ⟨class_name_or_iterator_or_var⟩]
⟨order_criteria_list⟩	::=	⟨order_criterion⟩ | ⟨order_criterion⟩ ',' ⟨order_criteria_list⟩
⟨order_criterion⟩	::=	⟨attribute⟩ ASC | ⟨attribute⟩ DESC
⟨temporal_expression⟩	::=	'(' ⟨temporal_bool_expr⟩ ')' | ⟨temporal_operation⟩
⟨comparison_expr⟩	::=	'(' ⟨expression⟩ ⟨comparison_predicate⟩ ⟨expression⟩ ')'
⟨operation_name⟩	::=	⟨built_in_operation⟩

Abbildung 2.3 Produktionen für den Aufbau von OOGQL-Ausdrücken in EBNF, Teil 3

‹class_name_or_iterator_or_var›	::=	IDENTIFIER
‹comparison_predicate›	::=	COMPARISON_OP | ‹topological_predicate› | DIRECTION_PREDICATE | TEMPORAL_PREDICATE
‹topological_predicate›	::=	TOPOLOGICAL_PREDICATE | TOPOLOGICAL_PREDICATE_OR_OP
‹class_name_or_user_def_op›	::=	IDENTIFIER
‹built_in_operation›	::=	TOPOLOGICAL_PREDICATE_OR_OP | GEOMETRIC_OPERATION
‹temporal_operation›	::=	TEMP_OP_SYSTEM_NO_ARGS | TEMP_OP_METHOD_NO_ARGS '(' ‹expression › ')' | TEMP_OP_CONST_ONE_ARG '(' ‹expression › ')' | TEMP_OP_PERIOD '(' ‹expression › [',' ‹expression ›] ')' | TEMP_OP_CONST_TWO_ARGS '(' ‹expression › ',' ‹expression › ')' | TEMP_OP_CONST_THREE_ARGS '(' ‹expression › ',' expression › ',' ‹expression › ')'
‹temporal_bool_expr›	::=	‹t_bool_or_expr›
‹t_bool_or_expr›	::=	‹t_bool_xor_expr› | ‹t_bool_or_expr› OR_OP ‹t_bool_xor_expr›
‹t_bool_xor_expr›	::=	‹t_bool_and_expr› | ‹t_bool_xor_expr› XOR_OP ‹t_bool_and_expr›
‹t_bool_and_expr›	::=	‹t_bool_unary_expr› | ‹t_bool_and_expr› AND_OP ‹t_bool_unary_expr›
‹t_bool_unary_expr›	::=	‹temporal_expression› | BOOL_UNARY_OP ‹temporal_expression›

über die Produktionen einer kontextfreien Grammatik [Sch97] zur Beschreibung aller zulässigen OOGQL-Ausdrücke. Die dort zum Einsatz kommenden Terminalsymbole werden an dieser Stelle nicht erläutert, weil ihre Bedeutung für die Anfragebearbeitung nebensächlich ist. Eine Übersicht ist in den oben angeführten Arbeiten zur OOGQL zu finden.

OOGQL-Anfragen können im Rahmen von GOODAC zum einen als Ad-hoc-Anfragen und zum anderen als in Anwendungsklassen eingebettete Anfragen gestellt werden [Voi97, 7.2.3]. Im Rahmen dieser Arbeit wird vor allem auf die Realisierung von Ad-hoc-Anfragen eingegangen. Im Folgenden soll die Verwendung eingebetteter Anfragen kurz erläutert werden. Dabei wird deutlich, dass bereits definierte eingebettete Anfragen aus Sicht der Komponente zur Anfragebearbeitung auch als Ad-hoc-Anfragen aufgefasst werden können, sodass eine Konzentration auf diese Art von Anfragen im Rahmen dieser Arbeit keine Einschränkung darstellt.

Die Verwendung eingebetteter Anfragen wird bereits von Jäger [Jäg03b, 3.3 f] näher beschrieben. Dabei wird im Wesentlichen ein OOGQL-Ausdruck als Anfrage in einem Anwendungsprogramm – beispielsweise innerhalb einer Methodendefinition innerhalb einer Klassendefinition – aufgeführt und bei Bedarf an die Komponente zur Anfragebearbeitung übergeben, um das zugehörige Anfrageergebnis im Anwendungsprogramm verwenden zu können. Im Rahmen der Anfragebearbeitung ist es ferner denkbar, eingebettete Anfragen bereits bei ihrer Definition in einen objektbasierten Ausführungsplan zu überführen, der anschließend in der Datenbank gespeichert werden kann. Anstelle der eingebetteten Anfrage würde dann ein Verweis auf diesen Ausführungsplan im Anwendungsprogramm gespeichert. Durch die Verfolgung eines derartigen Konzepts wird die Anfragebearbeitung deutlich beschleunigt, weil vor allem der Aufwand für den Optimierungsschritt entfällt. Allerdings können Veränderungen an der Datenbank dazu führen, dass der zuvor erzeugte objektbasierte Ausführungsplan keine effiziente Auswertung der Anfrage ermöglicht; Veränderungen von Klassendefinitionen und der Wegfall der Korrektheitsprüfung können in diesem Konzept dazu führen, dass eine Anfrage nicht mehr korrekt beantwortet wird. Daher werden in GOODAC nur OOGQL-Ausdrücke als eingebettete Anfragen gespeichert, die bei jeder Auswertung den kompletten Weg der Anfragebearbeitung (siehe auch Abschnitt 1.2.2), also vor allem die Korrektheitsprüfung und den Optimierungsschritt, durchlaufen. Einen ähnlichen Ansatz gibt es mit Embedded SQL schon für relationale Datenbankmanagementsysteme; eine ausführliche Beschreibung liefern zum Beispiel Connolly und Begg [CB02, 21.1].

2.3 Überführung von OOGQL-Ausdrücken in ein internes Format

Um die Bearbeitung einer Anfrage zu vereinfachen, muss sie in ein internes Format – in GOODAC eine von Carrie [Car04] definierte algebraische Darstellung – überführt werden [EN00, 18]. Diese Aufgabe übernimmt in GOODAC der so genannte *OOGQL-Parser*. Eine ausführliche Beschreibung einer frühen Version dieser Komponente geben bereits Ringena [Rin02, 5.4] und Eilers [Eil03], daher werden hier nur einige wesentliche Konzepte näher vorgestellt. Eine ausführliche Darstellung des internen Formats – der so genannten deskriptiven Algebra –, in das der OOGQL-Parser eine OOGQL-Anfrage überführt, findet sich in Abschnitt 4.3. In diesem Zusammenhang beschreibt schon Carrie [Car04] unter anderem beispielhaft für jede Klasse von OOGQL-Schlüsselwörtern einen entsprechenden Teilausdruck der von ihm konzipierten Algebra.

Das wesentliche Ziel der Anwendung des OOGQL-Parsers liegt in der Erzeugung eines Algebraausdrucks. Durch diese interne Darstellung der Anfrage in einer prozeduralen, formal definierten Repräsentationsform wird der weitere Ablauf der Anfragebearbeitung stark vereinfacht. Insbesondere die Komponente zur Anfrageoptimierung profitiert vom Übergang zu einer prozeduralen Darstellung: Erst hier sind die Anwendung von Heuristiken und die Berechnung von Kosten effektiv möglich [YM98, 2.3]. Aber auch für die Ausführungsebene bringt diese Übersetzung Vorteile mit sich, schließlich wird durch die frühe Korrektheits- und Typprüfung gewährleistet, dass das Ergebnis der Anfrage bestimmt werden kann. Durch die klare Unterscheidung zwischen der gegebenen deklarativen OOGQL-Anfrage und dem vom OOGQL-Parser erzeugten Algebraausdruck werden zudem die Verantwortungsbereiche der an der Anfragebearbeitung beteiligten Komponenten in GOODAC sowie die vorhandenen Schnittstellen betont.

Neben der Bereitstellung eines Algebraausdrucks besteht wie zuvor erwähnt ein weiteres Ziel der Übersetzung einer OOGQL-Anfrage in einen Algebraausdruck in der Überprüfung ihrer Korrektheit hinsichtlich ihrer Syntax und Semantik. Zu diesem Zweck findet neben einer lexikalischen und syntaktischen Analyse – siehe auch die Beschreibung dieser Schritte im Rahmen der Erzeugung objektbasierter Ausführungspläne in Abschnitt 3.2.4 – auch eine Typprüfung statt. Mit Hilfe des Systemkatalogs [Elv98] kann so untersucht werden, ob die auftretenden Typen – beispielsweise bei Parametern für Methodenaufrufe – kompatibel und alle Bezeichner – etwa für Klassennamen oder Attribute einer Klasse – gültig sind. Zusätzlich müssen – zum Beispiel für Rückgabewerte eines Methodenaufrufs – entsprechende Typinformationen aus dem Systemkatalog ausgelesen und in den zu erzeugenden Algebraausdruck aufgenommen werden.

Im Rahmen der Transformation einer OOGQL-Anfrage in einen Algebraausdruck werden nahezu alle Schlüsselwörter – insbesondere alle auch in SQL vorhandenen – auf generische Operationen [Heu97, 6.3.1] wie beispielsweise Selektionen oder Projektionen abgebildet. Diese haben gegenüber objektspezifischen Operationen den Vorteil, dass ihr Verhalten im Rahmen der Anfragebearbeitung bekannt und durch keine benutzerdefinierte Klasse veränderbar ist. Einige durch die Verwendung des Datenmodells OOGDM bedingte Schlüsselwörter – wie beispielsweise in temporalen oder topographischen Prädikaten – müssen allerdings auf objektspezifische Operationen [Heu97, 6.3.1] – also Methodenaufrufe – abgebildet werden, weil ihre Auswertung sowohl die Funktionalität einer konkreten Klasse als auch den Zustand einer zugehörigen Instanz berücksichtigen muss [Car04, 2]. Damit Benutzer bei der Definition ihrer Anwendungsklassen die Auswertung dieser Prädikate möglichst wenig berücksichtigen müssen, enthält das Datenmodell OOGDM bereits eine Vielzahl von Basisklassen, für die GOODAC die entsprechende Funktionalität bereithält. Anwendungsklassen können in diesem Zusammenhang – etwa durch Verwendung von Vererbung oder Assoziationen – von dieser Vielfalt profitieren.

2.4 Die Objektdefinitionssprache OOGDM-ODL

Eine Datendefinitionssprache (eine so genannte **D**ata **D**efinition **L**anguage) dient der Beschreibung des Aufbaus einer Datenbank, des so genannten Datenbank-Schemas [SKS02, 1.5]. In objektorientierten Datenbanksystemen wird oft eine Objektdefinitionssprache – beispielsweise ODL [CB00, 3.2] – eingesetzt, um den Aufbau der Datenbank – insbesondere die auftretenden Klassen, ihre Eigenschaften, ihre Funktionalität und ihre Beziehungen untereinander – zu beschreiben.

In GOODAC können Anwendungsentwickler die bereitgestellte Objektdefinitionssprache, die so genannte OOGDM-ODL (**O**bject **D**efinition **L**anguage of OOGDM) verwenden, um im Rahmen neuer Anwendungsprogramme zugehörige Klassen mit entsprechenden Attributen, Methoden und Beziehungen untereinander zu definieren. Neben einer Beschreibung der OOGDM-ODL diskutiert Voigtmann [Voi97, 6.1] sowohl ihren Bezug zur Objektdefinitionssprache ODL [CB00, 3.2] als auch ihre Einsatzgebiete, während

Jäger [Jäg03b] ausgehend von seiner eigenen Arbeit [Jäg03a] sowie den Überlegungen von Lange [Lan96] Implementierungsdetails für eine Verwendung von OOGDM-ODL in GOODAC beschreibt. Daher soll an dieser Stelle nur die Bedeutung der OOGDM-ODL in Hinblick auf die Anfragebearbeitung in GOODAC näher beleuchtet werden.

Die vom Anwendungsentwickler erstellten Klassendefinitionen werden von einer eigenen Vorverarbeitungskomponente, einem *ODL2CC* genannten Precompiler [Jäg03b], weiterverarbeitet. In diesem Zusammenhang werden die für die Anfragebearbeitung wesentlichen Aufgaben erledigt.

So registriert die Vorverarbeitungskomponente beispielsweise alle aus ihren zugehörigen Definitionen neu erzeugten Klassen mit ihren Attributen, Methoden und Beziehungen zu anderen Klassen beim Systemkatalog [Jäg03b, 5.1]. Dadurch wird es möglich, auch in Anwendungsprogrammen gestellte OOGQL-Anfragen, die Verweise auf diese Klassen oder zugehörige Instanzen beinhalten, erfolgreich auf Korrektheit zu überpüfen.

Weiterhin wird für jede neu erzeugte Klasse automatisch eine virtuelle Funktionstabelle [Str98, 2.5.5] bereitgestellt, mit deren Hilfe im Rahmen der Abarbeitung objektbasierter Ausführungspläne alle innerhalb dieser Klasse definierten Methoden über die Methoden `callMethod` und `getMethod` zugänglich sind [Jäg03b, 2.2]. Eine genauere Darstellung dieses Konzepts findet sich in der Beschreibung der Voraussetzungen zur Abarbeitung eines beispielhaften objektbasierten Ausführungsplans in Abschnitt 3.2.6.

Schließlich wird noch der Kopf einer Methode `tcreate` bereitgestellt, durch deren Verwendung sich neue Instanzen der Klasse ausgehend von einer textuellen Beschreibung erzeugen lassen [Jäg03b, 2.2]. Hierbei zählt es allerdings zu den Aufgaben des Anwendungsentwicklers, die korrekte Funktionalität der Methode `tcreate` durch Angabe ihres Rumpfs sicherzustellen. Die Verwendung dieser Methode im Rahmen der Erzeugung objektbasierter Ausführungspläne wird in Abschnitt 3.2.4 thematisiert.

Insgesamt sorgt diese Vorverarbeitungskomponente also dafür, dass ein Anwendungsentwickler neben der Bereitstellung der erforderlichen Funktionalität seiner Klassen nur eine weitere Methode für jede Klasse implementieren muss, damit die Klassen seines neu erzeugten Anwendungsprogramms gleichberechtigt mit den bereits vordefinierten Klassen des Datenmodells OOGDM im Rahmen der Anfragebearbeitung von GOODAC verwendet werden können.

Kapitel 3

Die Ausführungsebene – Berechnung des Anfrageergebnisses in Datenbanksystemen

Nachdem zuvor mit der Beschreibung der Anfragesprache OOGQL der Beginn der Anfragebearbeitung in GOODAC beschrieben wurde, sollen nun diejenigen Bestandteile vorgestellt werden, die zum Ende der Bearbeitung einer Anfrage zum Tragen kommen. Die Erläuterung der algebraischen Repräsentation sowie des eigentlichen Optimierungsprozesses von Anfragen finden sich in Kapitel 4 und Kapitel 5 dieser Arbeit. Vor allem die Beschreibung der algebraischen Repräsentation erfordert zum besseren Verständnis eine gute Kenntnis der Ausführungsebene von GOODAC, daher wird diese in diesem Kapitel eingeführt.

Auf der Ausführungsebene eines Datenbankmanagementsystems wird die bis dahin nur analysierte und optimierte Anfrage ausgeführt. Dazu werden ein vom Anfrageoptimierer erstellter und gegebenenfalls unter verschiedenen Alternativen ausgewählter Ausführungsplan untersucht und abgearbeitet sowie das Anfrageergebnis zurückgeliefert [SKS02, 13.1]. Bei der Abarbeitung eines Ausführungsplans kommen unterschiedliche Algorithmen – beispielsweise zur Durchführung von Selektionen oder zur Berechnung von Verbundoperationen – zum Einsatz. Eine Übersicht häufig zum Einsatz kommender Algorithmen liefern etwa Graefe [Gra93] und Silberschatz *et. al.* [SKS02, 13].

Ein Ausführungsplan kann auch als gerichteter Graph, als so genannter *ausführbarer Anfragegraph*, dargestellt werden. In diesem Fall repräsentieren die Knoten des Graphen konkret ausgewählte Algorithmen und zugehörige Parameterwerte zur Bearbeitung der Anfrage, während die Kanten Informationen über die Art des Datenaustausches zwischen den Knoten darstellen. Häufig hat ein ausführbarer Anfragegraph eine im Wesentlichen baumartige Struktur. Beispiele für derartige baumartige ausführbare Anfragegraphen finden sich bei Garcia-Molina *et. al.* [GMUW02, 16.7.6], Silberschatz *et. al.* [SKS02, 14.4.1] und Vossen [Vos99, 10.1.4]. In diesem Fall entspricht die Ausrichtung der Kanten des Graphen den Pfaden von der Wurzel zu den Blättern des Baums.

Dieses Kapitel befasst sich zunächst mit dem Aufbau der Ausführungsebene in Datenbankmanagementsystemen (Abschnitt 3.1), um sowohl die allgemeine Funktionsweise einer derartigen Komponente darzustellen als auch eine Grundlage für die Beschreibung der Ausführungsebene in GOODAC (Abschnitt 3.2) zu schaffen. Dieser zweite Abschnitt stellt an einem konkreten Beispiel die Struktur und die Auswertung von Ausführungsplänen in einem speziellen objektorientierten Datenbankkernsystem ausführlich vor.

3.1 Ausführungsebene in Datenbankmanagementsystemen

In diesem Abschnitt wird die grundlegende Funktionsweise der Ausführungsebene in Datenbankmanagementsystemen beschrieben. In diesem Zusammenhang werden neben einer allgemeinen Darstellung auf Grundlage verschiedener Lehrbücher (Abschnitt 3.1.1) auch die Ausführungsebenen einiger konkreter Datenbankmanagementsysteme grob dargestellt (Abschnitt 3.1.2). Die Präsentation setzt dabei voraus, dass ein ausführbarer Anfragegraph vorliegt, der einen abzuarbeitenden Ausführungsplan repräsentiert. Eine weitere ausführliche Darstellung der Ausführungsebene in relationalen Datenbankmanagementsystemen liefert Graefe [Gra93].

3.1.1 Konzepte für die Ausführungsebene in Datenbankmanagementsystemen

Für die durch die einzelnen Knoten eines ausführbaren Anfragegraphen repräsentierten Algorithmen stellt ein Datenbankmanagementsystem jeweils eine zugehörige Implementation bereit. Um der Vielzahl der möglichen Kombinationen dieser Algorithmen Rechnung zu tragen, wird ihre Realisierung dahingehend modularisiert, dass die einzelnen Verfahren nahezu beliebig miteinander kombiniert werden können. Eine Übersicht einiger möglicher Algorithmen findet sich sowohl bei Elmasri und Navathe [EN00, 18.2], Garcia-Molina *et. al.* [GMUW02, 15] sowie Silberschatz *et. al.* [SKS02, 13] als auch – im Rahmen der Darstellung der Knotentypen in ausführbaren Anfragegraphen von GOODAC – in Abschnitt 3.2.3. Daher werden sie an dieser Stelle nicht näher betrachtet.

Der Austausch von Daten zwischen diesen Realisierungen der einzelnen Knotentypen erfolgt dabei gewöhnlich nach zwei Verfahren: Entweder werden ermittelte Zwischenergebnisse auf dem Sekundärspeicher gelagert, bevor sie von der nächsten Operation verarbeitet werden (die so genannte *materialization* oder *Materialisierung*), oder es wird eine *Pipeline* zwischen den beteiligten Knoten erzeugt, durch die entsprechende Teilresultate ohne Zuhilfenahme des Sekundärspeichers weitergereicht werden (das so genannte *pipelining* oder *Durchleiten*) [SKS02, 13.7]. Das zweite Vorgehen lässt sich in Abhängigkeit von den Rollen der beteiligten Akteure noch genauer spezifizieren. Wird ein Daten bereitstellender Knoten nur aktiv, falls ein übergeordneter Knoten seine Ergebnisse als Eingabe erwartet, so wird dieses Verfahren als Nachfrage-getriebenes Durchleiten (so genanntes *demand-driven pipelining*) bezeichnet. Andernfalls erzeugt jeder Knoten solange Ergebnisdaten, bis die ausgehende Pipeline keine Elemente mehr aufnehmen kann oder er selbst auf Eingaben untergeordneter Knoten warten muss; diese Alternative wird als Produzenten-getriebenes Durchleiten (so genanntes *producer-driven pipelining*) bezeichnet.

Für jede Kante zwischen zwei Daten verarbeitenden Knoten des ausführbaren Anfragegraphen kann eines dieser drei Verfahren zum Austausch der Teilergebnisse ausgewählt werden. Die Materialisierung wird dabei nur verwendet, falls Durchleiten nicht möglich ist, etwa weil ein Algorithmus vor Beginn seiner Arbeit alle Eingabedaten benötigt. Doch auch hier zeichnet sich die Tendenz zur Verwendung von modifizierten Techniken des Durchleitens an. So können manche Algorithmen so angepasst werden, dass sie zwar weiterhin ohne Kenntnis aller Eingabedaten kein Resultat erzeugen können, aber dennoch in der Lage sind, auf Grundlage nur eines Teils der zu betrachtenden Daten Berechnungen durchzuführen.

Nachfrage-getriebenes Durchleiten stellt die einfacher zu implementierende Alternative der beiden Varianten zur Etablierung einer direkten Pipeline zwischen zwei benachbarten Knoten in einem ausführbaren Anfragegraphen dar [SKS02, 13.7.2.1]. Dieses Verfahren lässt sich beispielsweise als Iterator implementieren, der im Wesentlichen die drei Zugriffoperationen `open`, `next` und `close` zur Initialisierung der Pipeline, zur Anforderung eines neuen Datensatzes beim untergeordneten Knoten und zum Schließen der Pipeline besitzt [SKS02, 13.7.2.1].

Im Gegensatz dazu ermöglicht Produzenten-getriebenes Durchleiten eine hohe Parallelität bei der Abarbeitung eines gegebenen Ausführungsplans, weil selbst benachbarte Knoten des ausführbaren Anfragegraphen im Gegensatz zum zuvor vorgestellen Iterator-Konzept nicht direkt miteinander kommunizieren. Jeder Knoten ist darauf angewiesen, dass eine eingehende Pipeline durch einen anderen Knoten gefüllt wird; er selbst kann diesen Knoten nicht dazu veranlassen. Alle Knoten können daher solange ihrer Arbeit nachgehen, bis ihre ausgehende Pipeline voll oder eine eingehende Pipeline leer ist; sie können in diesem zweiten Fall jedoch nicht über die Pipeline mit untergeordneten Knoten in Kontakt treten, um neue Elemente zu erhalten.

Weil bei Verwendung einer Pipeline zwischen zwei Knoten der übergeordnete Knoten die Größe der insgesamt zu erwartenden Datenmenge nicht exakt bestimmen kann und weil er die Daten zudem nicht als Ganzes, sondern nur nach und nach erhält, wird der Datenfluss zwischen zwei benachbarten Knoten des ausführbaren Anfragegraphen auch als *Strom* bezeichnet [EN00, 18.2.7]. Handelt es sich bei den Daten um Tupel in einem relationalen Datenbanksystem, kann daher auch von einem *Tupelstrom* gesprochen werden, während die Menge der Objekte in einem Zwischenschritt bei der Abarbeitung eines Ausführungsplans in einem objektorientierten Datenbanksystem entsprechend als *Objektstrom* bezeichnet werden kann. In diesem Bild fließen also zwischen den einzelnen Knoten des ausführbaren Anfragegraphen Ströme von Daten. Einige Ströme entspringen dabei an einem die Datenbasis repräsentierenden Knoten – sie entstehen beispielsweise durch das Einlesen einer auf dem Sekundärspeicher gespeicherten Relation in einem relationalen Datenbanksystem –, während andere Ströme durch weitere Knoten erzeugt werden – etwa durch Zusammenführen zweier Ströme bei der Berechnung ihres Kreuzprodukts. Besitzt der ausführbare Anfragegraph weiterhin eine baumartige Struktur, enden einige Ströme an der Wurzel dieses Baums; das Resultat der Durchführung des durch diese Wurzel repräsentierten Algorithmus – zum Beispiel eine Projektion –

stellt somit das Anfrageergebnis dar.

Die bisher vorgestellten Grundlagen der Ausführungsebene in Datenbankmanagementsystemen stammen vor allem aus dem Kontext relationaler Datenbanksysteme. Allerdings weisen Elmasri und Navathe darauf hin, dass objektorientierte Datenbanksysteme diese Techniken ebenfalls einsetzen, wobei allerdings zusätzliche Ansätze realisiert werden, um die besonderen Eigenschaften objektorientierter Datenbanksysteme zu berücksichtigen [EN00, 18]. Auch Yu und Meng [YM98, 2.3] stellen die zusätzlichen Anforderungen an die Anfragebearbeitung in objektorientierten Datenbankmanagementsystemen dar. Sie zeigen zudem durch ihre Beschreibung einer Objektalgebra und des groben Ablaufs der Anfragebearbeitung in objektorientierten Datenbankmanagementsystemen, dass die zuvor präsentierten Techniken in diesem Kontext lediglich erweitert werden. Beispiele für diese zusätzlichen Anforderungen und ihre Umsetzung – etwa das späte Binden (siehe auch Seite 27) – finden sich im Rahmen der Beschreibung der Ausführungsebene in GOODAC in Abschnitt 3.2.

3.1.2 Ausführungsebene in konkreten Datenbankmanagementsystemen

In diesem Abschnitt wird die Ausführungsebene einiger konkreter Datenbankmanagementsysteme näher betrachtet, um zu zeigen, dass die zuvor genannten allgemeinen Methoden in bestehenden Systemen Eingang gefunden haben. Dabei werden zuerst einige Forschungsprototypen betrachtet, bevor kurz zwei kommerzielle Datenbankmanagementsysteme untersucht werden. Es werden nur die im Rahmen der Ausführungsebene entscheidenden Gesichtspunkte näher beleuchtet; andere Aspekte – beispielsweise der Anfrageoptimierung oder der Erweiterbarkeit der Systeme – bleiben unbeachtet. Auch die Leistung in Bezug auf Laufzeit und Speicherplatzbedarf soll hier – im Gegensatz zu Mortensen [Mor02] – nicht näher betrachtet werden.

GENESIS

Batory *et. al.* [BLW88] beschreiben die Ausführungsebene in GENESIS, einem erweiterbaren Datenbankmanagementsystem, das auf einem Funktions-orientierten Datenmodell basiert. Dieses System unterstützt eine strombasierte Auswertung von Ausführungsplänen, wobei jeder Operator der ausführbaren Ebene als *Stromübersetzer* bezeichnet wird, da er als Funktion aufgefasst werden kann, die eine bestimmte Anzahl von Strömen als Argumentwerte erwartet und einen Strom als Ausgabe liefert. Die Ströme in GENESIS folgen dabei dem Prinzip des Nachfrage-getriebenen Durchleitens; der Zugriff erfolgt über Iteratoren. Insbesondere kann in diesem Zusammenhang durch das Schließen der Pipeline zwischen zwei Operatoren die Arbeit der untergeordneten Operation beendet werden, bevor alle Datensätze betrachtet worden sind. Ausführungspläne in GENESIS werden im Rahmen der Ausführungsebene interpretierend abgearbeitet, es erfolgt also keine Übersetzung in ein ausführbares Programm.

EXODUS

Die Ausführungsebene in EXODUS [RC87] ist nur bedingt mit den Ausführungsebenen in herkömmlichen Datenbankmanagementsystemen vergleichbar. In EXODUS wird eine *E* genannte Erweiterung der Programmiersprache C++ verwendet, die es einem Systemprogrammierer ermöglichen soll, das System zu ergänzen. Auch die bereitgestellten Kernfunktionen sind mit Hilfe von E implementiert. Aus diesem Grund müssen etwa vorliegende Ausführungspläne von der Komponente zur Anfrageoptimierung in E formuliert und anschließend kompiliert werden, bevor sie zur Ausführung gelangen. Die eigentliche Abarbeitung eines Ausführungsplans erfolgt auch hier strombasiert nach dem Iteratorkonzept.

Gral

Auch im Rahmen von Gral [Güt89] werden die Operationen der Ausführungsebene modularisiert implementiert. Güting verweist zudem explizit darauf, dass die Komponente zur Auswertung von Ausführungsplänen keine Kenntnis über die Semantik der Operationen besitzt; sie kennt nur die erforderlichen Funktionsnamen zur Initialisierung einzelner die Operationen repräsentierenden Knoten eines vorliegenden ausführbaren Anfragegraphen sowie zur Anforderung weiterer Daten. Allerdings wird hierbei jeder Knoten des ausführbaren Anfragegraphen von einer übergeordneten Instanz innerhalb der Komponente zur Auswertung von Ausführungsplänen direkt angesprochen; eine Kommunikation zwischen den einzelnen Knoten – beispielsweise über eine Pipeline – findet also nicht statt. Das bedeutet, dass Gral zwar ein Nachfragegetriebenes Durchleiten von Datensätzen unterstützt; jedoch wird dieses während der gesamten Abarbeitung

eines Ausführungsplans nicht von den implementierten Operationen selbst, sondern von einer übergeordneten Instanz innerhalb der Komponente zur Auswertung von Ausführungsplänen gesteuert.

Starburst

Im erweiterbaren relationalen Datenbankmanagementsystem Starburst [HFLP89] kommt ebenfalls eine strombasierte Auswertung von Ausführungsplänen nach dem Iteratorkonzept zum Einsatz. Haas *et. al.* begründen an dieser Stelle zudem ausführlich, dass eine algebraische Repräsentation von Ausführungsplänen durch ihre formale Definition die Eingliederung neuer Knotentypen für ausführbare Anfragegraphen in das System erleichtert. Ausführungspläne werden in Starburst als baumartige ausführbare Anfragegraphen beschrieben, wobei jeder Knoten dieses Graphen einen Operator der ausführbaren Ebene repräsentiert [HCL+90].

Volcano

Die Ausführungsebene in Volcano [Gra94] basiert auch auf Strömen, die aufgrund des frei wählbaren Datenmodells sowohl aus Objekten als auch aus Tupeln bestehen können. Der Zugriff auf diese Ströme erfolgt durch Iteratoren; zudem sind alle Operationen als eigenständige Module realisiert, sodass das Hinzufügen weiter Operationen erleichtert wird. Die Ausführungspläne werden durch Ausdrücke einer ausführbaren Algebra repräsentiert. Weiterhin stößt die Komponente zur Auswertung von Ausführungsplänen selbst die Abarbeitung aller Knoten des ausführbaren Anfragegraphen an, sodass sich das eingesetzte Iterator-Konzept zur Kommunikation zwischen benachbarten Knoten auf die Realisierung der Methoden `next` und `close` beschränkt. Ebenso kann die Abarbeitung aller untergeordneten Knoten nicht durch eine Operation selbst beendet werden; nur die übergeordnete Kontrollinstanz kann durch Traversierung des ausführbaren Anfragegraphen die Arbeit der Operationen vorzeitig stoppen. Zudem werden Ausführungspläne in Volcano vor ihrer Abarbeitung kompiliert. Daneben unterstützt die Ausführungsebene in Volcano einige interessante Konzepte, beispielsweise die dynamische Auswahl eines zuvor kompilierten Ausführungsplans.

SECONDO

In SECONDO [GDF+99] werden Ausführungspläne als geschachtelte Listen repräsentiert, die sich in eine Baumdarstellung umformen lassen. Auch hier arbeiten alle Operationen der ausführbaren Ebene auf Strömen und kommunizieren über Iteratoren miteinander. Diese Nachfrage-getriebene Abarbeitung eines in Baumdarstellung vorliegenden Ausführungsplans erfolgt durch die Anforderung eines Ergebnisses bei einem Knoten dieses ausführbaren Anfragegraphen, der anschließend durch die Verwendung eines Iterators dafür sorgt, dass seine untergeordneten Knoten ebenfalls Ergebnisse liefern. Dieses Verfahren setzt sich kaskadierend fort, bis schließlich untergeordnete Knoten die benötigten Daten aus einer Datenbank auslesen und als Ergebnis der Iteratoranwendung an ihren übergeordneten Knoten zurückliefern.

DB2, Oracle und PostgreSQL

Silberschatz *et. al.* geben schließlich einen kurzen Überblick über die Ausführungseben in jüngeren Versionen der Datenbankmanagementsysteme Oracle [SKS02, 25] und DB2 [SKS02, 26]. Beide Systeme besitzen ebenfalls eine modular strukturierte Ausführungsebene. Als interne Repräsentationsform eines Ausführungsplans wählt beispielsweise DB2 einen ausführbaren Anfragegraphen mit baumartiger Struktur, der sich mit Hilfe eines bereitgestellten Werkzeugs graphisch darstellen lässt [Wie01, 6]. Auch Oracle bietet die Möglichkeit zur textuellen und graphischen Darstellung von Ausführungsplänen [HLUA02, 7.2]. In PostgreSQL können textuelle Ausführungspläne zwar ebenfalls dargestellt werden, allerdings ist nur eine textuelle Ausgabe möglich [Pos03, 13.1]. Über den eigentlichen Ablauf der Anfragebearbeitung geben die Hersteller dieser Systeme kaum etwas bekannt; DB2 basiert allerdings unter anderem auf dem zuvor vorgestellten Datenbankmanagementsystem Starburst, sodass einige der dort vorgestellten Konzepte sicherlich auch im Rahmen von DB2 Verwendung finden.

Zusammenfassung

Insgesamt zeigt sich also, dass die zuvor in Abschnitt 3.1.1 dargestellten Konzepte zur Repräsentation und Abarbeitung von Ausführungsplänen in zahlreichen Datenbankmanagementsystemen zur Anwendung

kommen. Es liegt daher nahe, diese bewährten Konzepte auch für die Entwicklung der Ausführungsebene in GOODAC einzusetzen. Dabei muss allerdings stets die Erweiterbarkeit dieser Komponente beachtet werden, damit spätere Ergänzungen in allen für die Anfragebearbeitung in GOODAC verantwortlichen Komponenten – und somit auch auf der Ausführungsebene – berücksichtigt werden können (siehe auch Abschnitt 6.2).

3.2 Ausführungsebene in GOODAC – Darstellung einer konkreten Realisierung

Im Folgenden werden die Struktur und die Funktionsweise der Ausführungsebene in GOODAC beschrieben. Diese realisiert die zuvor für relationale Datenbankmanagementsysteme vorgestellten Techniken und berücksichtigt zudem die Anforderungen an objektorientierte Datenbankmanagementsysteme. Daher repräsentiert die hier vorgestellte Komponente eine konkrete Realisierung dieser allgemein anerkannten Vorgehensweise zur Realisierung der Ausführungsebene in einem Datenbankmanagementsystem. Das theoretische Konzept eines ausführbaren Anfragegraphen findet sich in GOODAC als Ausführungsplan wieder. Ein Ausführungsplan betont im Gegensatz zu einem ausführbaren Anfragegraphen weniger die Struktur, sondern vielmehr die zur Ermittlung des Anfrageergebnisses benötigte Funktionalität. In GOODAC wird zudem zwischen einem *textuellen Ausführungsplan* – für die einfache Beschreibung der einzusetzenden Vorgehensweise zur Ermittlung des Anfrageergebnisses – und einem *objektbasierten Ausführungsplan* – für die interne Repräsentation und als Grundlage der eigentlichen Abarbeitung eines Ausführungsplans – unterschieden.

Die Beschreibung der Funktionsweise der Ausführungsebene in GOODAC beginnt mit der Erläuterung der internen Darstellung ausführbarer Anfragegraphen als textuelle Ausführungspläne (Abschnitt 3.2.1) und als objektbasierte Ausführungspläne (Abschnitt 3.2.2). Danach werden die für die Ausführung von Anfrageplänen verfügbaren Algorithmen vorgestellt (Abschnitt 3.2.3), bevor im Anschluss die Erzeugung (Abschnitt 3.2.4) und die Abarbeitung (Abschnitt 3.2.5) objektbasierter Ausführungspläne näher betrachtet werden. Zum Abschluss illustrieren einige Beispiele die genaue Funktionsweise der Ausführungsebene in GOODAC (Abschnitt 3.2.6).

Einige Resultate, die in diesem Abschnitt dargestellt werden, wurden ansatzweise von Ditt [Dit] entwickelt; eine erste Übersicht über entscheidende Konzepte liefert bereits Ringena [Rin02].

3.2.1 Textuelle Ausführungspläne

Textuelle Ausführungspläne stellen die Grundlage für die Ausführung von Anfragen in GOODAC dar. Auch ohne eine Anfrage in OOGQL (siehe auch Abschnitt 2.2) zu formulieren, können versierte Datenbankbenutzer – insbesondere Systemprogrammierer zur Evaluation hinzugefügter Knotentypen auf der Ausführungsebene oder zum Vergleich unterschiedlicher Resultate der Anfrageoptimierung – durch die Angabe eines textuellen Ausführungsplans noch zur Laufzeit eines Programms bestimmen, welche Daten aus der Datenbank extrahiert werden sollen. Durch textuelle Ausführungspläne steht ihnen die gleiche Funktionalität zur Verfügung, die auch während der Beantwortung einer in OOGQL formulierten Anfrage zum Einsatz kommt.

Die grundlegende Struktur textueller Ausführungspläne in GOODAC findet sich in Form von Produktionen einer kontextfreien Grammatik [Sch97] in Abbildung 3.1 wieder. Im Wesentlichen bestehen textuelle Ausführungspläne demnach aus Paaren von Bezeichnern und Konstruktoraufrufen. Bei der Analyse und dem Einlesen eines textuellen Ausführungsplans (siehe auch Abschnitt 3.2.4) werden Objekte über die entsprechenden Konstruktoraufrufe – repräsentiert durch einen Klassennamen `CLASSNAME` mit einer optionalen Argumentliste – erzeugt und mit dem übergebenen Bezeichner als Schlüssel in einem Dictionary [OW96, 1.6] für die aktuelle Anfrage gesammelt.

Entscheidend ist an dieser Stelle, dass für die Parameterwerte der genannten Konstruktoraufrufe neben einfachen Werten und weiteren Konstruktoraufrufen auch im aktuellen textuellen Ausführungsplan definierte Bezeichner zugelassen sind. Durch die Übergabe von Bezeichnern als Parameterwerte in Konstruktoraufrufen werden zirkuläre Assoziationen zwischen mehreren zu erzeugenden Objekten ermöglicht. Dadurch lassen sich beliebige ausführbare Anfragegraphen und nicht nur solche, die eine baumartige Struktur besitzen, auf textuelle Ausführungspläne abbilden. Nähere Informationen zu diesem Aspekt liefert Abschnitt 3.2.4, insbesondere bei der Beschreibung der Objektregistrierung auf Seite 39.

Abbildung 3.1 Produktionen für den Aufbau textueller Ausführungspläne in EBNF

‹variableObjectPairList›	::=	‹variableObjectPair› |
		‹variableObjectPair› ‹variableObjectPairList›
‹variableObjectPair›	::=	`VARIABLE` ‹object› ';'
‹object›	::=	`CLASSNAME` '(' [‹argumentList›] ')'
‹argumentList›	::=	‹argument› |
		‹argument› ',' ‹argumentList›
‹argument›	::=	`VALUE` |
		`VARIABLE` |
		‹object›

Beispiel 3.1 zeigt einen syntaktisch korrekten textuellen Ausführungsplan. Zuerst wird durch einen Konstruktoraufruf ein Objekt vom Typ `OOGDMInt` erzeugt und an den Bezeichner `myInt` gebunden. Anschließend wird ein Objekt vom Typ `OOGDMDouble` erzeugt, das den Namen `myDouble` zugewiesen erhält. An dieser Stelle wird insbesondere ein Bezeichner (`myShort`) als Parameterwert eines Konstruktoraufrufs verwendet. Das zugehörige Objekt wird erst in der abschließenden Zeile des vorliegenden textuellen Ausführungsplans definiert und an den entsprechenden Bezeichner gebunden. Der in Beispiel 3.1 gezeigte textuelle Ausführungsplan erzeugt also bei seiner Abarbeitung (siehe auch Abschnitt 3.2.4) drei einfache Objekte.

Die hier verwendeten Klassen mit Präfix `OOGDM` sind feste Bestandteile von GOODAC, weil sie Teile des zugrunde liegenden Datenmodells OOGDM implementieren. Ihre Bedeutung ist für das weitere Verständnis dieser Arbeit nicht von entscheidender Bedeutung, daher wird an dieser Stelle auf eine genauere Beschreibung der verfügbaren Klassen verzichtet. Eine Übersicht aller zur Implementation des Datenmodells OOGDM in GOODAC bereitgestellten Klassen findet sich zum Beispiel in den Arbeiten von Melnikov [Mel02], Ringena [Rin02] und Ziemann [Zie02].

Beispiel 3.1 Ein einfacher, syntaktisch korrekter textueller Ausführungsplan

```
myInt       OOGDMInt("42");
myDouble    OOGDMDouble(myShort);
myShort     OOGDMShort(myInt);
```

Textuelle Ausführungspläne, die auf in der Datenbank gespeicherte Objekte zugreifen, erfordern dafür geeignete Algorithmen und entsprechende Knoten eines ausführbaren Anfragegraphen, die in Abschnitt 3.2.3 vorgestellt werden. Aus diesem Grund finden sich erst dort Beispiele für entsprechende textuelle Ausführungspläne. Es sei aber bereits hier darauf hingewiesen, dass textuelle Ausführungspläne als eine Repräsentationsform ausführbarer Anfragegraphen aufgefasst werden können und somit bereits alle Informationen enthalten, die zur Abarbeitung eines Ausführungsplans erforderlich sind.

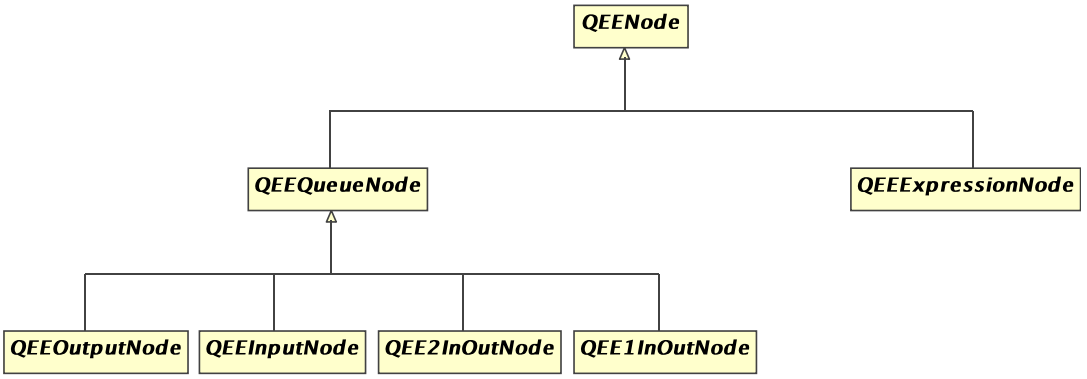
3.2.2 Objektbasierte Ausführungspläne

Objektbasierte Ausführungspläne stellen in GOODAC eine interne Repräsentation textueller Ausführungspläne dar. Sie lassen sich daher ebenfalls als ausführbare Anfragegraphen auffassen. Im Gegensatz zu textuellen Ausführungsplänen enthalten sie jedoch nicht nur die erforderlichen Informationen zu Abarbeitung eines Ausführungsplans, sondern zusätzlich die benötigte Funktionalität in Form der zum Einsatz kommenden Algorithmen.

Jeder Knoten eines ausführbaren Anfragegraphen wird dazu durch ein Objekt in einem objektbasierten Ausführungsplan (ein so genanntes *Knotenobjekt*) repräsentiert. Die Funktionalität dieser Knotenobjekte gewährleistet eine korrekte Abarbeitung des Ausführungsplans. So ist beispielsweise ein Objekt des Typs `QEEProjectionNode` [Rin02] in der Lage, eine Projektion zu erstellen, während eine Instanz der Klasse `QEESortNode` [Bre00] eine Sortierung durchführen kann.

Die Kanten zwischen zwei auf Objektströmen arbeitenden Knotenobjekten eines ausführbaren Anfragegraphen – also aus Implementationssicht die Pipelines – werden durch spezielle Warteschlangenobjekte (so genannte *Kantenobjekte*) dargestellt. Damit besitzen diejenigen auf Objektströmen arbeitenden Knotenobjekte eines objektbasierten Ausführungsplans, die adjazente Knoten eines ausführbaren Anfragegraphen repräsentieren, auch im objektorientierten Ausführungsplan eine direkte Verbindung untereinander: das assoziierte Warteschlangen- oder Kantenobjekt.

Abbildung 3.2 Hierarchie allgemeiner Knotentypen in ausführbaren Anfragegraphen



Dadurch, dass zusätzlich Instanzen bestimmter Klassen zur Repräsentation von Knotentypen ausführbarer Anfragegraphen in GOODAC – siehe auch Abschnitt 3.2.3 und dort insbesondere die Beschreibung der Unterklassen zu `QEEIndexNode` auf Seite 30 – Daten aus der Datenbank extrahieren können, ist der Datenfluss durch den objektbasierten Ausführungsplan gewährleistet. Nach der Extraktion aus der Datenbank werden die Daten über Warteschlangenobjekte zu Knotenobjekten weitergereicht, die bestimmte Operationen auf den Daten ausführen, beispielsweise eine Projektion oder eine Sortierung der Daten. Dieser Prozess setzt sich rekursiv fort, bevor zum Ende der Abarbeitung eines objektbasierten Ausführungsplans dann wiederum ein besonderes Objekt – zum Beispiel eine Instanz der auf Seite 31 beschriebenen Klasse `QEEPrintNode` – dafür sorgen muss, dass die zurückgelieferten Daten dem Benutzer zur Verfügung gestellt werden.

Schließlich treten in objektbasierten Ausführungsplänen ebenso wie in textuellen Ausführungsplänen Konstanten auf, die ebenfalls durch spezielle Knotenobjekte repräsentiert werden, die nicht auf Objektströmen arbeiten. So können beispielsweise einfache Selektionsprädikate realisiert werden, indem die unterschiedlichen Werte eines Attributs aller zu untersuchenden Objekte mit einer entsprechenden Konstanten verglichen werden.

Die nächsten Abschnitte gehen näher auf die zur Verfügung stehenden Knotentypen (Abschnitt 3.2.3) ein, bevor die Erzeugung objektbasierter Ausführungspläne (Abschnitt 3.2.4) thematisiert wird. Danach wird die Abarbeitung objektbasierter Ausführungspläne (Abschnitt 3.2.5) beschrieben. Zum Abschluss werden die bis dahin präsentierten Konzepte und die im weiteren Verlauf vorgestellte Funktionalität durch Beispiele dargestellt (Abschnitt 3.2.6).

3.2.3 Knotentypen in ausführbaren Anfragegraphen

In diesem Abschnitt werden zunächst diejenigen Knotentypen aufgeführt und beschrieben, die derzeit in ausführbaren Anfragegraphen zugelassen sind. Anschließend wird die konkrete Implementation der Knotentypen in GOODAC näher betrachtet. Hier wird insbesondere dargestellt, welche besonderen Eigenschaften die entsprechenden Klassen besitzen und welche Angaben erforderlich sind, um Objekte des entsprechenden Typs zu erzeugen.

Abbildung 3.2 zeigt, dass sich Knoten im Wesentlichen in fünf Kategorien einteilen lassen. In der Implementation werden einzelne Knotenobjekte durch Instanzen entsprechender Unterklassen der abstrakten Klasse `QEENode` dargestellt. Dabei weist die Abkürzung *QEE* in den Klassennamen darauf hin, dass die zugehörigen Objekte in der Auswertungskomponente für Anfragen (**Q**uery **E**valuation **E**ngine, siehe auch Silberschatz *et. al.* [SKS02, 13.1]) Verwendung finden.

Alle Knotentypen eines ausführbaren Anfragegraphen, die keine Objektströme verarbeiten, sondern Ausdrücke darstellen, die zur Abarbeitung eines Ausführungsplans erforderlich sind, also beispielsweise Selektionsprädikate oder Konstanten, werden intern durch Unterklassen zur Klasse `QEEExpressionNode` abgebildet. Alle anderen Knotentypen eines ausführbaren Anfragegraphen arbeiten auf Strömen von Objekten. Sie werden daher als Unterklassen zur Klasse `QEEQueueNode` dargestellt. Eine weitere Unterteilung lässt sich leicht finden, indem die Anzahl der benötigten Ein- und Ausgabeströme betrachtet wird. Die abstrakten Klassen `QEEOutputNode`, `QEEIntputNode`, `QEE2InOutNode` und `QEE1InOutNode` repräsentieren Knotentypen, die nur einen ausgehenden Strom – etwa bedingt durch das Auslesen von Objekten aus einer Datenbank –, nur einen eingehenden Strom – beispielsweise Knoten zur textuellen Ausgabe der Elemente eines eingehenden Objektstroms–, zwei eingehende und einen ausgehenden Strom – zum Beipiel bei der

Abbildung 3.3 Hierarchie zugelassener Knotentypen in ausführbaren Anfragegraphen

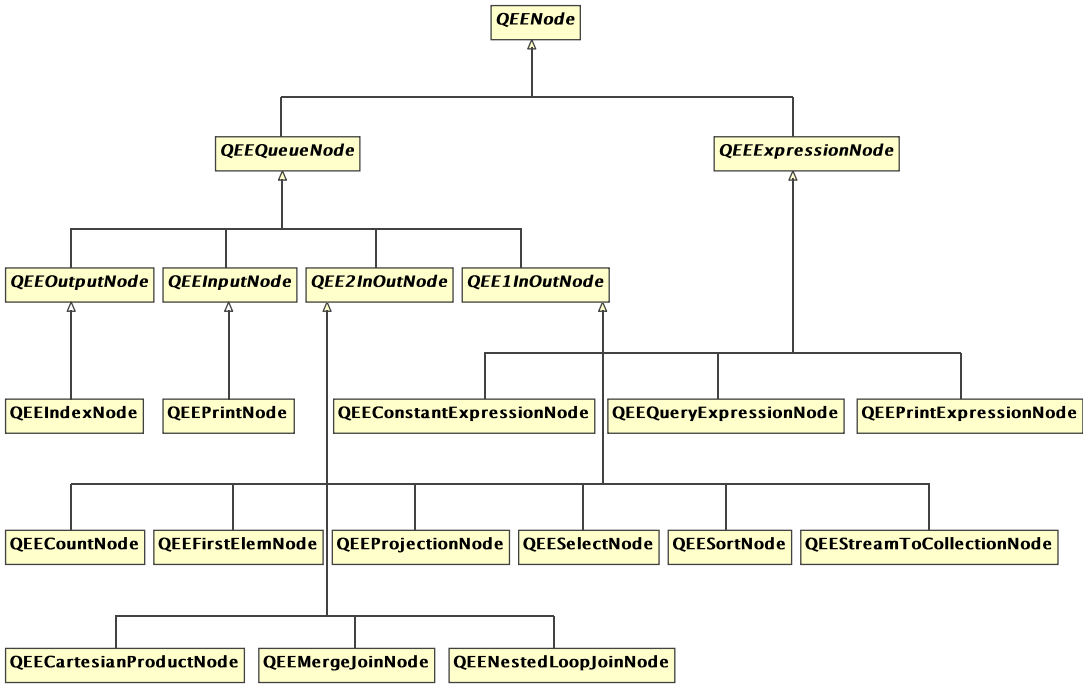


Tabelle 3.1 Funktionalität aller zugelassener Knotentypen in ausführbaren Anfragegraphen

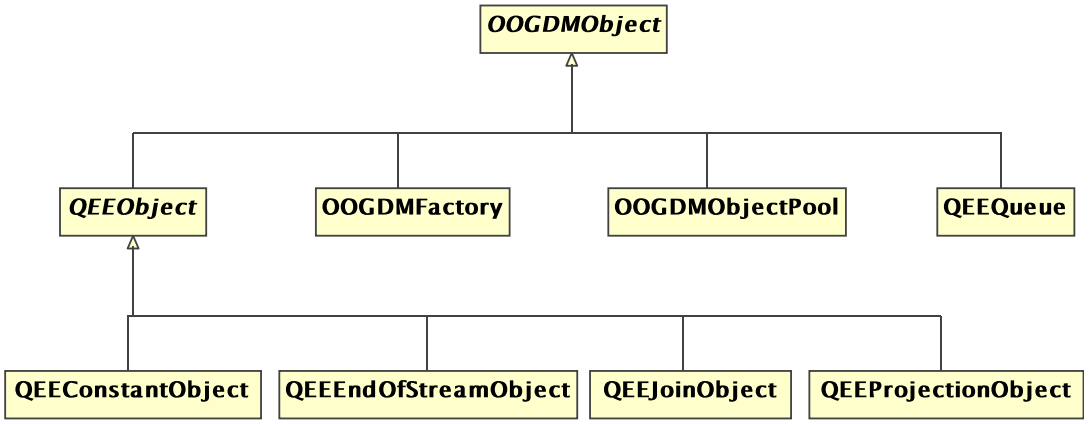
Knotentyp	Funktionalität
QEEConstantExpressionNode	Interne Darstellung von Konstanten
QEEQueryExpressionNode	Interne Darstellung von Prädikaten und Methodenaufrufen
QEEPrintExpressionNode	Weiterleitung eines Objekts in einen C++-Ausgabestrom, etwa zur Ausgabe eines Anfrageergebnisses
QEEIndexNode	Extraktion von Objekten aus der Datenbank und Umleitung in einen Ausgabestrom
QEEPrintNode	Ausgabe aller Objekte des Eingabestroms
QEECartesianProductNode	Erzeugung des kartesischen Produkts zweier Eingabeströme
QEEMergeJoinNode	Berechnung der Verbundoperation nach dem Merge-Join-Algorithmus für zwei Eingabeströme
QEENestedLoopJoinNode	Berechnung der Verbundoperation nach dem Nested-Loop-Join-Algorithmus für zwei Eingabeströme
QEECountNode	Bestimmung der Anzahl der Objekte eines Eingabestroms
QEEFirstElemNode	Abspaltung und Weiterleitung des ersten Objekts eines Eingabestroms
QEEProjectionNode	Durchführung einer Projektion für die Objekte eines Eingabestroms
QEESelectNode	Auswertung einer Selektionsbedingung für die Objekte eines Eingabestroms
QEESortNode	Sortierung und Weiterleitung der Objekte eines Eingabestroms
QEEStreamToCollectionNode	Einfügen der Objekte eines Eingabestroms in eine Kollektion

Berechnung eines Kreuzprodukts – sowie einen eingehenden und einen ausgehenden Strom – beispielsweise bei der Durchführung einer Selektion – besitzen. Alle bisher in ausführbaren Anfragegraphen zugelassenen Knotentypen lassen sich in diese Klassenhierarchie einordnen.

In Abbildung 3.3 werden alle derzeit zugelassenen Knotentypen gezeigt. Eine kurze Übersicht über ihre Funktionalität liefert Tabelle 3.1, während eine ausführlichere Beschreibung im weiteren Verlauf dieses Abschnitts erfolgt. Die Reihenfolge der in dieser Tabelle gezeigten Klassen entspricht aus Konsistenzgründen der in Abbildung 3.3 gewählten Anordnung, wobei abstrakte Oberklassen in der Tabelle nicht mehr aufgeführt werden.

Im Folgenden wird eine detaillierte Beschreibung dieser Knotentypen im Kontext textueller und ob-

Abbildung 3.4 Klassen für Hilfsaufgaben in objektbasierten Ausführungsplänen



jektbasierter Ausführungspläne gegeben. Dazu werden neben den benötigten Klassen für Hilfsaufgaben innerhalb objektbasierter Ausführungspläne insbesondere die erforderlichen Parameter zur Erzeugung der einzelnen Knotenobjekte in objektbasierten Ausführungsplänen und die genaue Funktionsweise der einzelnen Knotenobjekte vorgestellt. Wie bereits zu Beginn von Abschnitt 3.2 dargestellt, können objektbasierte Ausführungspläne auch als ausführbare Anfragegraphen aufgefasst werden. Aus diesem Grund können in der weiteren Beschreibung der Funktionalität einzelner Bestandteile objektbasierter Ausführungspläne eine auftretende Klasse oder der zugehörige Typ eines Objekts in objektbasierten Ausführungsplänen auch jeweils als Knotentyp verstanden werden.

Klassen für Hilfsaufgaben in objektbasierten Ausführungsplänen

Abbildung 3.4 zeigt alle Klassen, die neben den eigentlichen Knotenobjekten für die korrekte Funktionalität dieser Knotenobjekte in objektbasierten Ausführungsplänen erforderlich sind. Vier dieser Klassen finden sich dazu unterhalb der abstrakten Oberklasse `QEEObject` wieder, um eine einheitliche und einfache Behandlung im Rahmen der Abarbeitung eines objektbasierten Ausführungsplans gewährleisten zu können. Instanzen dieser Unterklassen werden nämlich – wie weiter unten beschrieben – als Elemente eines Objektstroms im Rahmen der Anfragebearbeitung verwendet. Die Klasse `QEEObject` wird zudem als Unterklasse zu `OOGDMObject` realisiert, sodass auch alle Objekte im Ergebnis von Anfragen wieder direkt in die Datenbank eingefügt werden können. Also lassen sich insbesondere alle Instanzen mit zugehöriger Oberklasse `QEEObject` auch als aus der Datenbank stammende Objekte aufgefassen.

Mit Ausnahme der Klasse `QEEEndOfStreamObject` sind die Unterklassen zu `QEEObject` als Hüllen (so genannte *Wrapper*) für die eigentlichen Datenbankobjekte erforderlich, um einen einheitlichen Zugriff auf Objekte im Rahmen der Anfragebearbeitung zu gewährleisten. Weiterhin können Methodenaufrufe an allen Stellen eines Anfragegraphen auftreten, sodass diese Hüllen vonnöten sind, um die vollständige Funktionalität der ursprünglichen Datenbankobjekte zu bewahren. Dadurch wird insbesondere spätes Binden (so genanntes *late binding*) [Oes01, 2.12] unterstützt. Schließlich stellt auch das Anfrageergebnis die zuvor aus der Datenbank ausgelesenen Objekte – etwa zur erneuten Speicherung in der Datenbank oder zur Weiterverarbeitung durch ein Anwendungsprogramm – bereit, sodass während der Abarbeitung eines Ausführungsplans keine neuen Objekte zur Repräsentation der Datenbankinhalte erzeugt werden. Weiterhin ermöglicht die Verwendung einer derartigen objekterhaltenden Anfragebearbeitung [Heu97, 6.3.5] unter anderem den Erhalt der ursprünglichen Objektidentität, sodass einfache Vergleiche zwischen dem Anfrageergebnis und dem urspünglichen Datenbankzustand – etwa zur Untersuchung der Veränderungen des Objektzustands – ermöglicht werden.

Ein Objekt vom Typ `QQEConstantObject` assoziiert ein Objekt aus der Datenbank oder ein über den textuellen Ausführungsplan definiertes konstantes Objekt. In jedem Fall wird ein Objekt mit zugehöriger Oberklasse `OOGDMObject` gekapselt.

Objekte vom Typ `QEEEndOfStreamObject` dienen der Kennzeichnung des Endes eines Stroms von Objekten. Dadurch können die weiter unten beschriebenden Knotenobjekte, beispielsweise ein Objekt vom Typ `QEEProjectionNode`, feststellen, ob ein zugrunde liegender Eingabestrom versiegt ist, indem sie überprüfen, ob dieser Objektstrom nur noch Instanzen der Klasse `QEEEndOfStreamObject` enthält. Die Verwendung von Objekten dieses Typs wird zum Beispiel bei der Darstellung der Abarbeitung objektbasierter

Ausführungspläne ab Seite 39 deutlich.

Durch ein Objekt vom Typ `QEEJoinObject` werden Referenzen auf zwei Objekte gekapselt. Dadurch ist es möglich, auch nach der Berechnung des kartesischen Produkts oder der Verbundoperation für zwei Ströme von Objekten ungehindert auf die Attribute und Methoden jedes der ursprünglichen Objekte zuzugreifen. Im Gegensatz zu relationalen Datenbanksystemen, in denen nur Attributwerte für Datensätze bei der Berechnung von Anfragen zu berücksichtigen sind, müssen in objektrelationalen [SKS02, 9] und objektorientierten [SKS02, 8] Datenbanksystemen auch die durch Methoden bereitgestellte Funktionalität nach der Verknüpfung zweier Objekte erhalten bleiben. Durch Verwendung mehrerer Instanzen der Klasse `QEEJoinObject` ist eine Kapselung von mehr als zwei Objekten möglich, wodurch beispielsweise die Hintereinanderausführung mehrerer Verbundoperationen oder die Verknüpfung von mehr als zwei Objektströmen zu ihrem kartesischen Produkt ermöglicht werden.

Eine Instanz der Klasse `QEEProjectionObject` stellt die Möglichkeit bereit, ein Objekt auf bestimmte Attributwerte und Rückgaben von Methodenaufrufen zu reduzieren [Rin02, 5.3]. Auch hier zeigt sich deutlich der Unterschied zu relationalen Datenbanksystemen; in objektrelationalen [SKS02, 9] und objektorientierten [SKS02, 8] Datenbanksystemen können im Resultat einer Projektion im Gegensatz zu relationalen Datenbanksystemen Rückgabewerte von Methodenaufrufen enthalten sein, sodass das zu projizierende Objekt nicht einfach um bestimmte Attributwerte beschnitten werden darf [Rin02, 5.3].

Jede Instanz der Klasse `QEEQueue` stellt ein Warteschlangenobjekt dar. Ein derartiges Objekt repräsentiert eine Kante zwischen zwei Knotenobjekten und realisiert einen Objektstrom, indem es Objekte von einem assoziierten Knotenobjekt anfordert und bei Bedarf an ein weiteres Knotenobjekt weiterleitet. Warteschlangenobjekte unterstützen das Durchleiten von Zwischenergebnissen (siehe auch Abschnitt 3.1) [CB02, 20.5], um unnötige Zugriffe auf den Sekundärspeicher zur Speicherung von Zwischenresultaten zu vermeiden. Eine genauere Beschreibung der Funktionalität dieser Warteschlangenobjekte findet sich in Abschnitt 3.2.5.

Zu den Klassen `OOGDMFactory` und `OOGDMObjectPool` werden keine Instanzen gebildet, stattdessen dienen sie als globale Einrichtungen im Rahmen der Ausführungsebene in GOODAC. Die Klasse `OOGDMFactory` stellt Funktionalität bereit, die erforderlich ist, um neue Objekte aus einer textuellen Beschreibung zu erzeugen. Die Klasse `OOGDMObjectPool` hält jede auf der Grundlage eines textuellen Ausführungsplans erzeugte Instanz einer Klasse unter einem ebenfalls innerhalb des textuellen Ausführungsplans spezifizierten Bezeichner vor. Damit kann während der Abarbeitung des erzeugten objektbasierten Ausführungsplans über diesen Bezeichner auf das entsprechende Objekt zugegriffen werden (siehe auch Abschnitte 3.2.4 und 3.2.5).

Untertypen von `QEEExpressionNode`

Wie bereits oben beschrieben, arbeiten Instanzen aller Unterklassen zu `QEEExpressionNode` nicht auf Strömen, sondern nur auf einzelnen Objekten. Ihre genaue Einbindung in die Anfragebearbeitung wird zum einen in der Beschreibung der entsprechenden Knotentypen, die Ströme von Objekten verarbeiten, als auch in den Erläuterungen zur Erzeugung (Abschnitt 3.2.4) und zur Abarbeitung (Abschnitt 3.2.5) objektbasierter Ausführungspläne näher vorgestellt.

Objekte der Klasse `QEEConstantExpressionNode` binden ein übergebenes Objekt an eine Instanz der Klasse `QEEConstantObject`. Der Konstruktor der Klasse `QEEConstantExpressionNode` erwartet dazu ein beliebiges Objekt aus der Datenbank, also eine Instanz einer Unterklasse von `OOGDMObject`. Ein Knoten dieses Typs liefert somit auf Anfrage ein Objekt vom Typ `QEEConstantObject` zurück.

In textuellen Ausführungsplänen könnten entsprechende Einträge zur Erzeugung einer Instanz der Klasse `QEEConstantExpressionNode` – wie in Beispiel 3.2 gezeigt – aufgebaut sein. Auf Anfrage würde also etwa das an den Bezeichner `myFirstConstantExpressionNode` gebundene Objekt das im Konstruktor übergebene Objekt vom Typ `OOGDMInt` in einer Instanz der Klasse `QEEConstantObject` kapseln und diese zurückliefern.

Beispiel 3.2 Konstruktoraufrufe für `QEEConstantExpressionNode` in textuellen Ausführungsplänen

myFirstConstantExpressionNode	QEEConstantExpressionNode(OOGDMInt(“42”));
mySecondConstantExpressionNode	QEEConstantExpressionNode(OOGDMString(“constant”));

Instanzen der Klasse `QEEQueryExpressionNode` ermöglichen die Darstellung von Methodenaufrufen in objektbasierten Ausführungsplänen. Somit lassen sich neben einzelnen Methodenaufrufen – etwa im

Zusammenhang mit einer Projektion – insbesondere Prädikate einer Selektion oder einer Verbundoperation über die Verknüpfung mehrerer Instanzen dieser Klasse verwirklichen. Dazu erwartet der Konstruktor als erstes Argument den Namen eines Objekts mit Obertyp `QEENode`. Dieses ist dafür verantwortlich, auf Anforderung jeweils ein konkretes Objekt zurückzuliefern.

Sollte es sich beim ersten Argument um eine Instanz zur Oberklasse `QEEQueueNode` handeln, wird das aktuelle Element des eingehenden Objektstroms angefordert. Wird hingehen eine Instanz mit Oberklasse `QEEExpressionNode` als erstes Argument des Konstruktoraufrufs angegeben, liefert diese abhängig vom konkreten Typ folgende Rückgabe:

`QEEConstantExpressionNode`: Es wird das gekapselte Objekt zurückgeliefert.

`QEEQueryExpressionNode`: Hier stellt das Ergebnis des realisierten Methodenaufrufs die Rückgabe dar.

`QEEPrintExpressionNode`: Konkrete Instanzen dieser Klasse geben das auszugebende Objekt zurück.

In jedem Fall sorgt also das über den ersten Parameterwert des Konstruktors angegebene Objekt dafür, dass ein Objekt zurückgeliefert wird, das den durch die weiteren Argumente näher beschriebenen Methodenaufruf ausführen kann.

Da es sich bei diesem zurückgegebenen Objekt auch um eine Instanz der Klasse `QEEJoinObject` handeln kann, wird als zweiter Parameterwert des Konstruktors der Klasse `QEEQueryExpressionNode` ein Objekt vom Typ `OOGDMUShort` erwartet, um gegebenenfalls eines der gekapselten Objekte zu identifizieren. Eine entsprechende Wahl dieses Parameterwerts sorgt dafür, dass auch eine Berücksichtigung geschachtelter Instanzen der Klasse `QEEJoinObject` ermöglicht wird. Der dritte Parameterwert des Konstruktoraufrufs ist der entsprechende Methodenname; alle weiteren Parameterwerte bilden die unter Umständen leere Argumentliste. Ein Knoten des Typs `QEEQueryExpressionNode` liefert demzufolge jeweils das Ergebnis des zugrunde liegenden Methodenaufrufs zurück.

Beispiel 3.3 zeigt drei Konstruktoraufrufe für die Klasse `QEEQueryExpressionNode`. Im ersten Aufruf wird ein Objekt erzeugt, das auf Anfrage zuerst das aktuelle Objekt des von `aQueueNode` referenzierten Objektstroms anfordert. Sollte es sich dabei um eine Instanz von `QEEJoinObject` handeln, wird das gekapselte Objekt mit der Nummer `1` ausgewählt, andernfalls wird diese Angabe ignoriert. Schließlich erfolgt für dieses Objekt der Aufruf der Methode `getName`. Der zweite Konstruktoraufruf erzeugt ein Objekt, das auf Anfrage erneut das aktuelle Objekt des über `aQueueNode` referenzierten Objektstroms anfordert. Auf dieses Objekt – im Fall eines Objekts vom Typ `QEEJoinObject` auf die unter `0` gekapselte Instanz – wird anschließend die Methode `operator<` mit der durch das Objekt `aConstantExpressionNode` repräsentierten Konstanten als einzigem Argument angewendet. Die durch den dritten in Beispiel 3.3 gezeigten Konstruktoraufruf erzeugte Instanz der Klasse `QEEQueryExpressionNode` sorgt dafür, dass zuerst beide Objekte vom Typ `QEEQueryExpressionNode` – also die Objekte mit den Bezeichnern `aQueryExpressionNode1` und `aQueryExpressionNode2` – ausgewertet werden. Danach wird für das zurückgegebene Objekt des ersten Objekts (`aQueryExpressionNode1`) die Methode `operator==` aufgerufen, wobei das Ergebnis der Auswertung des zweiten Objekts (`aQueryExpressionNode2`) als einziges Argument dient. Das Resultat dieses Methodenaufrufs stellt abschließend das Rückgabeobjekt bei der Auswertung des durch den dritten Konstruktoraufruf erzeugten Objekts `myThirdQueryExpressionNode` dar.

Beispiel 3.3 Konstruktoraufrufe für `QEEQueryExpressionNode` in textuellen Ausführungsplänen

```
myFirstQueryExpressionNode      QEEQueryExpressionNode(aQueueNode,
                                                       OOGDMUShort("1"),
                                                       "getName");
mySecondQueryExpressionNode     QEEQueryExpressionNode(aQueueNode,
                                                       OOGDMUShort("0"),
                                                       "operator<",
                                                       aConstantExpressionNode);
myThirdQueryExpressionNode      QEEQueryExpressionNode(aQueryExpressionNode1,
                                                       OOGDMUShort("0"),
                                                       "operator==",
                                                       aQueryExpressionNode2);
```

Die Klasse `QEEPrintExpressionNode` stellt die Funktionalität zur Ausgabe von einzelnen Objekten bereit. Jede Instanz dieser Klasse wird durch die Angabe des Namens eines Objekts einer Unterklasse

von QEEQueueNode sowie eines C++-Ausgabestroms erzeugt. Bei der Angabe nur des ersten Arguments im Konstruktor wird cout als C++-Ausgabestrom gesetzt. Jede Instanz fordert bei ihrer Aktivierung ein Objekt des aktuellen Stroms vom assoziierten Objekt des Typs QEEQueueNode an und leitet dieses an den angegebenen Ausgabestrom weiter.

Das über den in Beispiel 3.4 gezeigten Konstruktoraufruf erzeugte Objekt fordert somit jeweils das aktuelle Objekt des von myPrintNode referenzierten eingehenden Objektstroms an und gibt es auf den C++-Ausgabestrom cout aus.

Beispiel 3.4 Konstruktoraufruf für QEEPrintExpressionNode in textuellen Ausführungsplänen

```
myPrintExpressionNode    QEEPrintExpressionNode(myPrintNode);
```

Objekte vom Typ QEEQueryExpressionNode oder QEEPrintExpressionNode verweisen wie zuvor beschrieben unter Umständen auf eine Instanz mit Oberklasse QEEQueueNode. Im Rahmen der Beschreibung einiger der folgenden Knotentypen – etwa QEESelectNode auf Seite 35 oder QEEPrintNode auf Seite 31 – wird zusätzlich deutlich, dass diese wiederum Objekte vom Typ QEEQueryExpressionNode oder QEEPrintExpressionNode referenzieren. Durch diese gegenseitigen Verweise entstehen Zyklen im ausführbaren Anfragegraphen, weil beispielsweise in diesem Anfragegraphen sowohl eine gerichtete Kante von einer Instanz der Klasse QEEPrintExpressionNode zu einem Objekt vom Typ QEEPrintNode als auch eine Kante von eben diesem Objekt zur erwähnten Instanz der Klasse QEEPrintExpressionNode existiert. Deswegen kann dieser Anfragegraph im Kontext der Anfragebearbeitung in GOODAC nicht als Anfragebaum – also als ausführbarer Anfragegraph mit einer im Wesentlichen baumartigen Struktur – aufgefasst werden. Weitere Beispiele für Zyklen in ausführbaren Anfragegraphen finden sich in Abschnitt 3.2.6 – beispielsweise bei der Darstellung der Berechnung einer Selektion in Abbildung 3.12 auf Seite 52.

Untertypen von QEEOutputNode

In GOODAC werden Objekte in der Datenbank intern durch einen Index, also durch eine Instanz einer Unterklasse zur abstrakten Klasse OOGDMIndex, verwaltet, die wiederum nur eine transiente Hülle darstellt, um unabhängig vom zugrunde liegenden Datenbanksystem ObjectStore auf die in der Datenbank gespeicherten Elemente zugreifen zu können. Eine genauere Übersicht über dieses Konzept liefert bereits Voigtmann [Voi97, 7.3.4 f], während Steinberg [Ste96, 3.4] einige Implementationsdetails beschreibt. Daher sollen die detaillierte Funktionsweise und die konkrete Realisierung an dieser Stelle nicht vorgestellt werden. Im Rahmen der Abarbeitung objektbasierter Ausführungspläne ist allein der Zugriff auf konkrete Datenbankobjekte von Interesse, um diese aus der Datenbank auslesen zu können. Wurde – beispielsweise vom Anfrageoptimierer – ein bestimmter Index für den Zugriff auf die Datenobjekte bestimmt, so sorgt ein Objekt mit Oberklasse OOGDMIndexScan dafür, dass die benötigten Datenbankobjekte sequentiell über diesen Index ausgelesen werden können [Mel02, 4.4].

Die Klasse QEEIndexNode, die bisher einzige Unterklasse von QEEOutputNode, stellt nun Objekte bereit, die in objektbasierten Ausführungsgraphen dafür sorgen, dass ein Strom von Datenbankobjekten erzeugt wird. Dazu wird eine konkrete Instanz der Klasse QEEIndexNode durch einen Konstruktoraufruf mit einem Objekt mit zugehöriger Oberklasse OOGDMIndexScan als einzigem Parameter erzeugt. Daraufhin werden alle Objekte, die die assoziierte Instanz einer Unterklasse von OOGDMIndexScan bereitstellt, in einen neuen Strom eingefügt. Abschließend wird ein neues Objekt vom Typ QEEEndOfStreamObject in den Strom eingefügt, um das Ende dieses Stroms zu kennzeichnen.

In Beispiel 3.5 findet sich ein entsprechender Konstruktoraufruf zur Erzeugung eines Objekts vom Typ QEEIndexNode. Als Parameter wird eine Instanz der Klasse OOGDMListIndexScan verwendet, die wiederum einen listenbasierten Index für alle Datenbankobjekte vom Typ City kapselt. Bei einem listenbasierten Index handelt es sich im Wesentlichen um eine lineare Liste, die alle Instanzen einer Klasse referenziert und für diese ausschließlich einen sequentiellen Zugriff ausgehend vom ersten Listenelement ermöglicht. Insbesondere unterstützt ein listenbasierter Index keine Suchschlüssel und eignet sich daher vor allem für komplexe Objekte, für die sich keine totale Ordnung bestimmen lässt. Das zuvor beschriebene Objekt vom Typ QEEIndexNode erzeugt also einen Strom, der aus allen in der Datenbank vorhandenen Instanzen zur Klasse City besteht (siehe auch Melnikov [Mel02, 4.4] sowie Abschnitt 3.2.6).

Beispiel 3.5 Konstruktoraufruf für QEEIndexNode in textuellen Ausführungsplänen

```
myIndexNode    QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
```

Untertypen von QEEInputNode

Unterklassen zu QEEInputNode stellen jeweils Objekte bereit, die nur über einen eingehenden Objektstrom, jedoch über keinen ausgehenden Strom verfügen. Benötigt werden derartige Objekte beispielsweise zum Ende eines objektbasierten Ausführungsplans, wenn die Objekte im Resultat der Anfrage ausgegeben oder an anderer Stelle in der Datenbank gespeichert werden sollen.

Bisher wird für objektbasierte Ausführungspläne nur die Möglichkeit realisiert, Objekte auszugeben. Dazu dient die Klasse QEEPrintNode, deren Konstruktor zur Erzeugung von Instanzen den Namen eines Warteschlangenobjekts für den eingehenden Objektstrom sowie als zweiten Parameterwert den Namen eines Objekts vom Typ QEEPrintExpressionNode erwartet. Bei der Ausführung wird dann das assoziierte QEEPrintExpressionNode-Objekt aufgefordert, das aktuelle Objekt des eingehenden Stroms zu verarbeiten.

Der nachfolgend in Beispiel 3.6 gezeigte Konstruktoraufruf erzeugt also ein entsprechendes Objekt der Klasse QEEPrintNode, das alle Objekte des durch das Warteschlangenobjekt aQueueNode eingehenden Stroms über den assoziierten Knoten vom Typ myPrintExpressionNode ausgibt.

Beispiel 3.6 Konstruktoraufruf für QEEPrintNode in textuellen Ausführungsplänen

```
myPrintNode    QEEPrintNode(aQueueNode, myPrintExpressionNode);
```

Beispiel 3.7 zeigt eine erste Anwendung von Knoten des Typs QEEIndexNode und QEEPrintNode sowie von Warteschlangenobjekten – also Instanzen der Klasse QEEQueue. Das Objekt namens myIndexNode entsteht durch einen Konstruktoraufruf, der dem in Beispiel 3.5 dargestellten entspricht. Es sorgt also für das Auslesen aller Instanzen des Typs City aus der Datenbank und fügt diese in seinen ausgehenden Objektstrom ein. Dieser Objektstrom wird über das Warteschlangenobjekt myQueueCity repräsentiert, das von einer Instanz der Klasse QEEPrintNode – nämlich myPrintNode – ausgelesen wird. Dieser Knoten zur Ausgabe aller eingehenden Objekte aktiviert für jedes Element des eingehenden Objektstroms sein assoziiertes Objekt myPrintExp, das schließlich für eine Ausgabe dieser Elemente auf den C++-Ausgabestrom cout sorgt, weil kein anderer C++-Ausgabestrom als Parameter angegeben ist.

Beispiel 3.7 Textueller Ausführungsplan zur Ausgabe aller Instanzen der Klasse City

```
myIndexNode    QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myQueueCity    QEEQueue(myIndexNode);
myPrintNode    QEEPrintNode(myQueueCity, myPrintExp);
myPrintExp     QEEPrintExpressionNode(myPrintNode);
```

Untertypen von QEE2InOutNode

Die Unterklassen zur Klasse QEE2InOutNode stellen jeweils Instanzen bereit, die bestimmte Objekte zweier eingehender Objektströme unter gegebenenfalls näher spezifizierten Bedingungen miteinander verknüpfen, in einem Objekt vom Typ QEEJoinObject kapseln und in einen Ausgabestrom einfügen. Bisher werden in GOODAC die drei aus Abbildung 3.3 ersichtlichen Knotentypen bereitgestellt.

Ein Objekt vom Typ QEECartesianProductNode bildet das kartesische Produkt der Objekte der beiden eingehenden Ströme und fügt die erzeugten Objekt vom Typ QEEJoinObject in einen ausgehenden Objektstrom ein. Zur Erzeugung einer Instanz der Klasse QEECartesianProductNode erwartet der entsprechende Konstruktor nur die Namen zweier Eingabeströme, also zweier Kantenobjekte, weil keine Bedingungen an die eingehenden Objekte gestellt werden.

Durch den im nachfolgenden Beispiel 3.8 dargestellten Konstruktoraufruf wird ein neues Objekt vom Typ QEECartesianProductNode erzeugt, das das kartesische Produkt derjenigen Objekte bildet, die von den durch die Warteschlangenobjekte mit den Bezeichnern aQueueNode1 und aQueueNode2 repräsentierten Objektströmen geliefert werden.

Beispiel 3.8 Konstruktoraufruf für QEECartesianProductNode in textuellen Ausführungsplänen

```
myCartesianProductNode    QEECartesianProductNode(aQueueNode1, aQueueNode2);
```

Die beiden verbleibenden Klassen mit Oberklasse QEE2InOutNode ermöglichen die Berechnung der Verbundoperation für zwei Eingabeströme nach unterschiedlichen Verfahren. Eine ausführliche Beschrei-

bung der Berechnung von Verbundoperationen geben beispielsweise Breimann [Bre00] sowie Mishra und Eich [ME92].

Ein Objekt vom Typ `QEEMergeJoinNode` [Bre00, 4.2.2] ist in der Lage, für die Objekte zweier Eingabeströme die Verbundoperation gemäß eines angegebenen Prädikats zu berechnen. Der hierbei verwendete *Merge-Join-Algorithmus* [SKS02, 13.5.4] setzt zwingend voraus, dass die Eingabeströme nach einer dem Prädikat der Verbundoperation entsprechenden totalen Ordnung sortiert [Knu97, 5] sind. Werden beispielsweise zwei Attributwerte der beiden Eingabeströme im Prädikat der Verbundoperation verglichen, so müssen die Eingabeströme bezüglich einer totalen Ordnung auf dem gemeinsamen Wertebereich dieser Attribute sortiert vorliegen. Wird bereits innerhalb dieses Prädikats eine totale Ordnung vorausgesetzt, so muss diese zwingend für die vorhergehende Sortierung verwendet worden sein. Die hier verwendete Implementation ermöglicht im Gegensatz zu der häufig in Lehrbüchern – siehe hierzu Connolly und Begg [CB02, 20.4.3], Date [Dat00, 17.7], Elmasri und Navathe [EN00, 18.2.3], Garcia-Molina *et. al.* [GMUW02, 15.4.7], Kemper und Eickler [KE01, 8.2.2], Lewis *et. al.* [LBK02, 13.4.2], Riccardi [Ric01, 13.3.4], Silberschatz *et. al.* [SKS02, 13.5.4] sowie Vossen [Vos99, 15.4.2] – oder auch in der ursprünglichen Präsentation dieses Verfahrens [BE77] zu findenden Darstellung auch Prädikate, die nicht auf der Gleichheit von Attributwerten beruhen [Bre00, 4.2.2].

Um ein Objekt vom Typ `QEEMergeJoinNode` zu erzeugen, erwartet der Konstruktor vier Parameter. Zuerst werden wie bereits beim Konstruktor der Klasse `QEECartesianProductNode` die Namen zweier Warteschlangenobjekte angegeben, um die in die Verbundoperation involvierten Eingabeströme zu bestimmen. Weiterhin wird der Name einer Instanz der Klasse `QEEQueryExpressionNode` erwartet, die das Prädikat der Verbundoperation repräsentiert. Der zugrunde liegende Methodenaufruf dieses Prädikats wird jeweils für Paare von Objekten aus beiden Eingabeströmen ausgeführt und muss demzufolge insbesondere einen Wert vom Typ `bool` zurückliefern. Komplexe Prädikate lassen sich in diesem Zusammenhang durch geschachtelte Methodenaufrufe – von denen beispielsweise manche boolesche Operationen darstellen – und eine entsprechende Anzahl von Instanzen der Klasse `QEEQueryExpressionNode` – von denen zum Beispiel einige wiederum als Parameter für den Konstruktoraufruf weiterer Objekte dieses Typs dienen – realisieren. Ein entsprechendes Prädikat zeigt etwa der textuelles Ausführungsplan in Beispiel 3.19 auf Seite 43. Beim vierten Parameterwert des Konstruktoraufrufs für Instanzen der Klasse `QEEMergeJoinNode` handelt es sich ebenfalls um den Namen eines Objekts vom Typ `QEEQueryExpressionNode`. Dieses stellt die totale Ordnung eines Wertebereichs dar, nach der beide Eingabeströme sortiert sind. Mit Hilfe dieses Parameters kann das erzeugte Objekt vom Typ `QEEMergeJoinNode` durch die Auswertung des repräsentierten Methodenaufrufs bei Durchführung des Merge-Join-Algorithmus entscheiden, ob für ein Objekt eines Eingabestroms noch weitere Objekte des anderen Eingabestroms das Prädikat der Verbundoperation erfüllen können. Sobald zwei Objekte der beiden Eingabeströme das Prädikat der Verbundoperation erfüllen, werden sie in einer Instanz der Klasse `QEEJoinObject` gekapselt und in den Ausgabestrom eingefügt. Eine genauere Beschreibung der Funktionsweise dieser Klasse, die zudem mit Beispielen illustriert ist, liefert schon Breimann [Bre00, 4.2.2].

In Beispiel 3.9 wird ein Objekt vom Typ `QEEMergeJoinNode` über einen Konstruktoraufruf erzeugt, das Objekte aus den durch die Warteschlangenobjekte `aQueueNode1` und `aQueueNode2` realisierten Eingabeströmen gemäß eines Prädikats der Verbundoperation unterzieht. Dieses Prädikat wird durch den dritten Parameterwert des Konstruktoraufrufs, also das an den Bezeichner `aQueryExpressionNode1` gebundene Objekt vom Typ `QEEQueryExpressionNode`, repräsentiert und nun für alle in Frage kommenden Paare von Objekten der beiden Eingabeströme ausgewertet. Die Menge der zu untersuchenden Paare lässt sich im Allgemeinen durch Berücksichtigung der totalen Ordnung, nach denen die Eingabeströme sortiert sind und die durch das an den Bezeichner `aQueryExpressionNode2` gebundene und den vierten Parameterwert des Konstruktoraufrufs darstellende Objekt vom Typ `QEEQueryExpressionNode` repräsentiert wird, reduzieren. Das ist möglich, indem die Objekte der beiden Eingabeströme gemäß dieser Ordnung miteinander verglichen werden [Bre00, 4.2.2]. Als Resultat wird ein Objektstrom erzeugt, der alle in Instanzen der Klasse `QEEJoinObjekt` eingeschlossenen Paare von Objekten, die das Prädikat der Verbundoperation erfüllen, enthält.

Beispiel 3.9 Konstruktoraufruf für `QEEMergeJoinNode` in textuellen Ausführungsplänen

```
myMergeJoinNode    QEEMergeJoinNode(aQueueNode1,
                                    aQueueNode2,
                                    aQueryExpressionNode1,
                                    aQueryExpressionNode2);
```

Eine Instanz der Klasse `QEENestedLoopJoinNode` [Rin02, 5.2] ermöglicht im Gegensatz zu den zuvor vorgestellten Objekten des Typs `QEEMergeJoinNode` die Berechung der Verbundoperation zwischen den Elementen zweier Eingabeströme nach dem *Nested-Loop-Join-Algorithmus* [SKS02, 13.5.1]. Im Unterschied zum zuvor beschriebenen Knotentyp `QEEMergeJoinNode` wird keine besondere Bedingung wie beispielsweise eine vorliegende Sortierung oder die Existenz einer totalen Ordnung auf dem Wertebereich der Attribute im Prädikat der Verbundoperation an die Eingabeströme gestellt. Daher ermöglichen die Instanzen der Klasse `QEENestedLoopJoinNode` die Verwendung aller für eine Verbundoperation erlaubten Prädikate. Zur Erzeugung eines konkreten Objekts erwartet der Konstruktor analog zu den Konstruktoren der oben beschriebenen anderen beiden Unterklassen zur Klasse `QEE2InOutNode` zuerst den Namen zweier Warteschlangenobjekte, die die beiden Eingabeströme darstellen. Zusätzlich wird als dritter Parameterwert noch der Name eines Objekts vom Typ `QEEQueryExpressionNode` erwartet, das wie auch bei Instanzen der zuvor beschriebenen Klasse `QEEMergeJoinNode` das Prädikat der Verbundoperation repräsentiert. Alle Paare von Objekten der beiden Eingabeströme, für die die Auswertung des Prädikats der Verbundoperation ein positives Ergebnis liefert, werden in einer neuen Instanz der Klasse `QEEJoinObject` gekapselt und in den ausgehenden Objektstrom eingefügt.

Der in Beispiel 3.10 gezeigte Ausschnitt eines textuellen Ausführungsplans führt zur Erzeugung eines neuen Objekts vom Typ `QEENestedLoopJoinNode`, das an den Bezeichner `myNestedJoinNode` gebunden wird. Dieses Objekt untersucht für jedes Paar von Objekten in den beiden Eingabeströmen der durch die Bezeichner `aQueueNode1` und `aQueueNode2` bestimmten Warteschlangenobjekte, ob sie das durch das unter dem Bezeichner `aQueryExpressionNode` verfügbare Objekt vom Typ `QEEQueryExpressionNode` repräsentierte Prädikat der Verbundoperationen erfüllen. Verläuft dieser Test positiv, liefert also der durch das Objekt mit dem Namen `aQueryExpressionNode` gekapselte Methodenaufruf den Wert `true` zurück, so wird das untersuchte Paar von Objekten in einem neuen Objekt vom TypQEEJoinObject gekapselt und dem ausgehenden Objektstrom hinzugefügt.

Beispiel 3.10 Konstruktoraufruf für `QEENestedLoopJoinNode` in textuellen Ausführungsplänen

```
myNestedJoinNode    QEENestedLoopJoinNode(aQueueNode1,
                                          aQueueNode2,
                                          aQueryExpressionNode);
```

Untertypen von `QEE1InOutNode`

Als letzte Gruppe von Knotentypen sollen im Folgenden die Untertypen zur Klasse `QEE1InOutNode` vorgestellt werden. Ihnen ist gemein, dass sie genau einen eingehenden und genau einen ausgehenden Objektstrom besitzen. Bisher werden von GOODAC die bereits in Abbildung 3.3 gezeigten Knotentypen bereitgestellt.

Die Instanzen der Klasse `QEECountNode` zählen die Objekte ihres assoziierten Eingabestroms und liefern die Anzahl in Form eines Objekts vom Typ `OOGDMInt` – gekapselt in einer Instanz der Klasse `QEEConstantObject` – zurück. Der Konstruktor besitzt demzufolge nur einen Parameter, nämlich den Namen des Warteschlangenobjekts, das den Eingabestrom darstellt.

In Beispiel 3.11 wird also eine Instanz der Klasse `QEECountNode` erzeugt, die die Anzahl der Objekte im durch das Warteschlangenobjekt mit dem Bezeichner `aQueueNode` realisierten Eingabestrom bestimmt und als einziges Objekt eine Instanz der Klasse `OOGDMInt`, die eben diese Anzahl angibt, mit einem umschließenden Objekt vom Typ `QEEConstantObject` in ihren Ausgabestrom einfügt.

Beispiel 3.11 Konstruktoraufruf für `QEECountNode` in textuellen Ausführungsplänen

```
myCountNode    QEECountNode(aQueueNode);
```

Beispiel 3.12 zeigt einen etwas umfangreicheren textuellen Ausführungsplan, der zu einer Bestimmung der Anzahl der Objekte im Ergebnis einer Verbundoperation führt. Diese Verbundoperation bestimmt diejenigen in der Datenbank gespeicherten Städte, die die gleiche Einwohnerzahl besitzen.

Die ersten vier Zeilen dieses textuellen Ausführungsplans ähneln dem Beginn des in Beispiel 3.7 dargestellten textuellen Ausführungsplans. Sie führen zum Auslesen aller Objekte des Typs `City` aus der Datenbank und dem Einfügen in die durch die Warteschlangenobjekte `myQueueCity1` und `myQueueCity2` repräsentierten Objektströme. Diese beiden Objektströme dienen nun als Basis der durch `myNestedLoopJoin`

dargestellten Berechnung der Verbundoperation. Die Repräsentation des Prädikats dieser Verbundoperation erfolgt durch die an die Bezeichner `myQuery1`, `myQuery2` und `myQuery3` gebundenen Knoten des Typs `QEEQueryExpressionNode`. Der Knoten `myNestedLoopJoin` wertet dazu für jedes Paar von Objekten der eingehenden Ströme den assoziierten Knoten `myQuery1` aus. Dieser wiederum ruft die Methode `operator==` des durch `myQuery2` zurückgelieferten Objekts mit dem Resultat von `myQuery3` als Argument auf. Die Knoten `myQuery2` und `myQuery3` fordern schließlich beim ursprünglichen Knoten `myNestedLoopJoin` das aktuelle Objekt des ersten – für `myQuery2` – und zweiten – für `myQuery3` – Eingabestroms an und wenden dessen Methode `getInhabitants` an, um die Einwohnerzahl der entsprechenden Stadt zu bestimmen. Diese wird anschließend an den Knoten `myQuery1` zurückgegeben, sodass dieser die beiden Einwohnerzahlen auf Gleichheit testen und das ermittelte boolesche Resultat für den die Verbundoperation repräsentierenden Knoten `myNestedLoopJoin` bereitstellen kann. Eine weitere ausführliche Darstellung eines ähnlichen Ablaufs findet sich bei der Beschreibung der Erzeugung und Abarbeitung objektbasierter Ausführungspläne in Abschnitt 3.2.6. Im nachfolgenden Beispiel wird das Ergebnis der Verbundoperation an das Warteschlangenobjekt `myQueueJoinResult` weitergegeben. Dieses repräsentiert den eingehenden Objektstrom für den Knoten `myCountNode`, dessen Aufgabe darin besteht, die Anzahl der Elemente in diesem eingehenden Objektstrom zu zählen.

Beispiel 3.12 Textueller Ausführungsplan zur Bestimmung der Elementzahl nach einer Verbundoperation

```
myIndexNode1      QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myIndexNode2      QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myQueueCity1      QEEQueue(myIndexNode1);
myQueueCity2      QEEQueue(myIndexNode2);
myNestedLoopJoin  QEENestedLoopJoinNode(myQueueCity1, myQueueCity2, myQuery1);
myQuery1          QEEQueryExpressionNode(myQuery2,
                                         OOGDMUShort("0"),
                                         "operator==",
                                         myQuery3);
myQuery2          QEEQueryExpressionNode(myNestedLoopJoin,
                                         OOGDMUShort("0"),
                                         "getInhabitants");
myQuery3          QEEQueryExpressionNode(myNestedLoopJoin,
                                         OOGDMUShort("1"),
                                         "getInhabitants");
myQueueJoinResult QEEQueue(myNestedLoopJoin);
myCountNode       QEECountNode(myQueueJoinResult);
```

Ein Objekt vom Typ `QEEFirstElemNode` liefert nur das erste Objekt seines Eingabestroms zurück und fügt es in seinen ausgehenden Objektstrom ein. In Verbindung mit einem Sortierknoten lässt sich so beispielsweise das Maximum bezüglich einer bestimmten Ordnung auf den Objekten eines Stroms bestimmen. Der Konstruktor dieser Klasse erwartet wiederum nur den Namen eines Warteschlangenobjekts als einzigen Parameterwert, damit jede erzeugte Instanz der Klasse `QEEFirstElemNode` auf ihren jeweils eingehenden Objektstrom zugreifen kann.

Der in Beispiel 3.13 dargestellte Konstruktoraufruf erzeugt demzufolge bei der Abarbeitung des hier nur auszugsweise gezeigten textuellen Ausführungsplans eine Instanz der Klasse `QEEFirstElemNode`, die ausschließlich das erste Element des durch `aQueueNode` bestimmten Eingabestroms an ihren Ausgabestrom weiterleitet. Die weiteren Elemente des eingehenden Objektstroms finden also keine Beachtung.

Beispiel 3.13 Konstruktoraufruf für `QEEFirstElemNode` in textuellen Ausführungsplänen

```
myFirstElemNode  QEEFirstElemNode(aQueueNode);
```

Objekte vom Typ `QEEProjectionNode` [Rin02, 5.3] ermöglichen die Durchführung von Projektionen für alle Objekte des eingehenden Objektstroms. Zur Erzeugung einer neuen Instanz erwartet der Konstruktor neben dem obligatorischen textuellen Verweis auf das den Eingabestrom darstellende Warteschlangenobjekt zwei weitere Argumente, die beide vom Typ `OOGDMList` sein müssen. Die erste dieser Listen enthält beliebig viele Zeichenketten. Diese geben die jeweiligen Bezeichner an, unter denen die einzelnen projizierten

Werte ab sofort verfügbar sind. Diese projizierten Werte werden durch die Elemente der zweiten übergebenen Instanz der Klasse `OOGDMList` bestimmt. Diese zweite Liste muss nämlich ebenso viele Objekte vom Typ `QEEQueryExpressionNode` enthalten wie die erste Liste Zeichenketten enthält. Ein auf diese Art und Weise erzeugtes Objekt vom Typ `QEEProjectionNode` legt nun für jedes Objekt des eingehenden Stroms eine neue Instanz der Klasse `QEEProjectionObject` (siehe auch Seite 28) an. In dieser Instanz speichert ein Objekt vom Typ `QEEProjectionNode` nun jeweils Paare von Zeichenketten und Objekten, wobei die Zeichenketten aus der ersten übergebenen Liste stammen und die Objekte als Ergebnis der durch die Instanzen der Klasse `QEEQueryExpressionNode` repräsentierten Methodenaufrufe zurückgeliefert werden. Sobald alle zu projizierenden Werte in die Instanz der Klasse `QEEProjectionObject` eingefügt worden sind, wird diese dem ausgehenden Objektstrom hinzugefügt. Anschließend wird das nächste Objekt des eingehenden Stroms auf die gleiche Art und Weise behandelt.

Der in Beispiel 3.14 gezeigte Auszug eines textuellen Ausführungsplans führt zur Erzeugung eines recht einfach aufgebauten Objekts vom Typ `QEEProjectionNode`. Die Projektion wird für alle Elemente des über das Warteschlangenobjekt mit dem Bezeichner `aQueueNode` eingehenden Objektstroms durchgeführt. Dazu wird für jedes eingehende Objekt eine neue Instanz der Klasse `QEEProjectionObject` angelegt, in die als einziger Eintrag ein Paar bestehend aus der Zeichenkette `''aName''` und dem Resultat des durch `aQueryExpressionNode`, ein Objekt vom Typ `QEEQueryExpressionNode`, dargestellten Methodenaufrufs eingefügt wird. Im Anschluss wird diese Instanz der Klasse `QEEProjectionObject` in den ausgehenden Objektstrom eingefügt. Alle weiteren Operationen, die diesen ausgehenden Objektstrom weiterverarbeiten, können also nur noch auf das unter dem Bezeichner `''aName''` gespeicherte Objekt zugreifen. Alle anderen Attribute und sämtliche Methoden der eingehenden Objekte sind somit während der weiteren Abarbeitung des vorliegenden Ausführungsplans nicht mehr verfügbar.

Beispiel 3.14 Konstruktoraufruf für `QEEProjectionNode` in textuellen Ausführungsplänen

```
myProjectionNode    QEEProjectionNode(aQueueNode,
                                      OOGDMList("aName"),
                                      OOGDMList(aQueryExpressionNode));
```

Eine Instanz der Klasse `QEESelectNode` ist in der Lage, Selektionen durchzuführen, also für jedes Objekt eines eingehenden Stroms zu entscheiden, ob es ein gegebenes Prädikat erfüllt. Nur in diesem Fall wird das eingehende Objekt in den ausgehenden Strom eingefügt, andernfalls wird es verworfen. Zur Erzeugung einer konkreten Instanz erwartet der Konstruktor der Klasse `QEESelectNode` zwei Elemente, und zwar den Namen eines Warteschlangenobjekts, das den eingehenden Objektstrom repräsentiert, sowie den Namen eines Objekts vom Typ `QEEQueryExpressionNode`, das das auszuwertende Prädikat darstellt.

Der in Beispiel 3.15 auszugsweise dargestellte textuelle Ausführungsplan führt also zur Erzeugung eines Objekts vom Typ `QEESelectNode`, das für jedes Element des durch das Warteschlangenobjekt `aQueueNode` realisierten eingehenden Objektstroms ein Prädikat auswertet. Dieses Prädikat wird durch die unter dem Bezeichner `aQueryExpressionNode` verfügbare Instanz der Klasse `QEEQueryExpressionNode` repräsentiert. Eine weitergehende Beschreibung der Rolle von Objekten des Typs `QEESelectNode` in objektbasierten Ausführungsplänen sowie ihrer Funktionsweise im Rahmen komplexerer Beispiele findet sich in Abschnitt 3.2.6.

Beispiel 3.15 Konstruktoraufruf für `QEESelectNode` in textuellen Ausführungsplänen

```
mySelectNode    QEESelectNode(aQueueNode, aQueryExpressionNode);
```

Die Klasse `QEESortNode` [Bre00, 4.2.1] stellt Instanzen bereit, die es ermöglichen, einen eingehenden Objektstrom gemäß einer totalen Ordnung zu sortieren. Als konkretes Verfahren kommt eine bereits von Breimann [Bre00, 4.2.1] beschriebene Kombination aus einem modifizierten Zwei-Wege-Mergesort-Algorithmus [OW96, 2.4.1] und einem Quicksort-Verfahren [CLR97, 8] zum Einsatz. Dabei werden mit Hilfe des Quicksort-Algorithmus sortierte Teilfolgen einer bestimmten Länge gebildet, die anschließend durch das Zwei-Wege-Mergesort-Verfahren miteinander verschmolzen werden. Die Länge dieser vorsortierten Teilfolgen wird durch einen übergebenen Parameterwert beim Start des Sortiervorgangs angegeben [Bre00, 4.2.1]. Um eine konkrete Instanz dieser Klasse zu erzeugen, erwartet der Konstuktor wiederum den Namen eines Warteschlangenobjekts zur Repräsentation des eingehenden Objektstroms. Zusätzlich müssen der Name eines die Sortierbedingung darstellenden Objekts vom Typ `QEEQueryExpressionNode` sowie

eine Referenz auf eine Instanz der Klasse `OOGDMInt` – zur Bestimmung der Länge der durch den Quicksort-Algorithmus gebildeten sortierten Teilfolgen – übergeben werden. Dadurch, dass sich die Länge der vorsortierten Teilfolgen für jeden einzelnen Knoten vom Typ `QEESortNode` in jedem Ausführungsplan getrennt angeben lässt, wird zudem gewährleistet, dass der zur Verfügung stehende Hauptspeicher durch diese Operationen optimal genutzt werden kann.

Der in Beispiel 3.16 gezeigte Konstruktoraufruf führt somit zu einer Instanz der Klasse `QEESortNode`, die alle Objekte des vom Warteschlangenobjekt `aQueueNode` realisierten eingehenden Stroms unter der an den Bezeichner `aQueryExpressionNode` gebundenen Bedingung sortiert. Dabei besitzen alle durch den Quicksort-Algorithmus vorsortierten Teilfolgen jeweils die Länge `32`. Dieses erzeugte Objekt wird an den Bezeichner `mySortNode` gebunden.

Beispiel 3.16 Konstruktoraufruf für `QEESortNode` in textuellen Ausführungsplänen

```
mySortNode    QEESortNode(aQueueNode, aQueryExpressionNode,OOGDMInt("32"));
```

Die Instanzen der Klasse `QEEStreamToCollectionNode` sind schließlich in der Lage, die Objekte eines eingehenden Stroms in eine Kollektion einzufügen und dieses Kollektionsobjekt als einziges Element – gekapselt in einer Instanz der Klasse `QEEConstantObject` – an ihren ausgehenden Strom weiterzuleiten. Hierdurch wird vor allem die weitere Verarbeitung des Anfrageergebnisses erleichert, so kann dieses Kollektionsobjekt beispielsweise erneut als Datenbankwurzel [Pro03, 4] in der Datenbank gespeichert werden, um zum Beispiel einen späteren Zugriff auf die Elemente des Resultats der vorliegenden Anfrage zu vereinfachen. Als ersten Parameterwert zur Erzeugung einer neuen Instanz erwartet der Konstruktor der Klasse `QEEStreamToCollectionNode` daher den Namen eines Kantenobjekts, während das zweite Argument ein Kollektionsobjekt mit Oberklasse `OOGDMCollection` [Mel02, 4.1.2] darstellen muss.

Beispiel 3.17 präsentiert somit einen exemplarischen Konstruktoraufruf zur Erzeugung eines an den Namen `myStreamToCollNode` gebundenen Objekts vom Typ `QEEStreamToCollectionNode`. Dieses Objekt fügt auf Anfrage alle Elemente des durch das Kantenobjekt `aQueueNode` dargestellten eingehenden Objektstroms in eine neu erzeugte Instanz der Klasse `OOGDMBag` ein, bevor es dieses Objekt vom Typ `OOGDMBag` in einer neuen Instanz der Klasse `QEEConstantObject` kapselt und als einziges Element an das ausgehende Kantenobjekt weiterleitet.

Beispiel 3.17 Konstruktoraufruf für `QEEStreamToCollectionNode` in textuellen Ausführungsplänen

```
myStreamToCollNode    QEEStreamToCollectionNode(aQueueNode,OOGDMBag());
```

3.2.4 Erzeugung objektbasierter Ausführungspläne

Nachdem bisher nur die Struktur textueller (Abschnitt 3.2.1) und objektbasierter (Abschnitt 3.2.2) Ausführungspläne sowie die Erzeugung und Funktionsweise konkreter Knotenobjekte (Abschnitt 3.2.3) thematisiert worden sind, beschreibt dieser Abschnitt die Erzeugung objektbasierter Ausführungspläne aus gegebenen textuellen Ausführungsplänen.

Die Erzeugung objektbasierter Ausführungspläne erfolgt im Wesentlichen in vier Schritten:

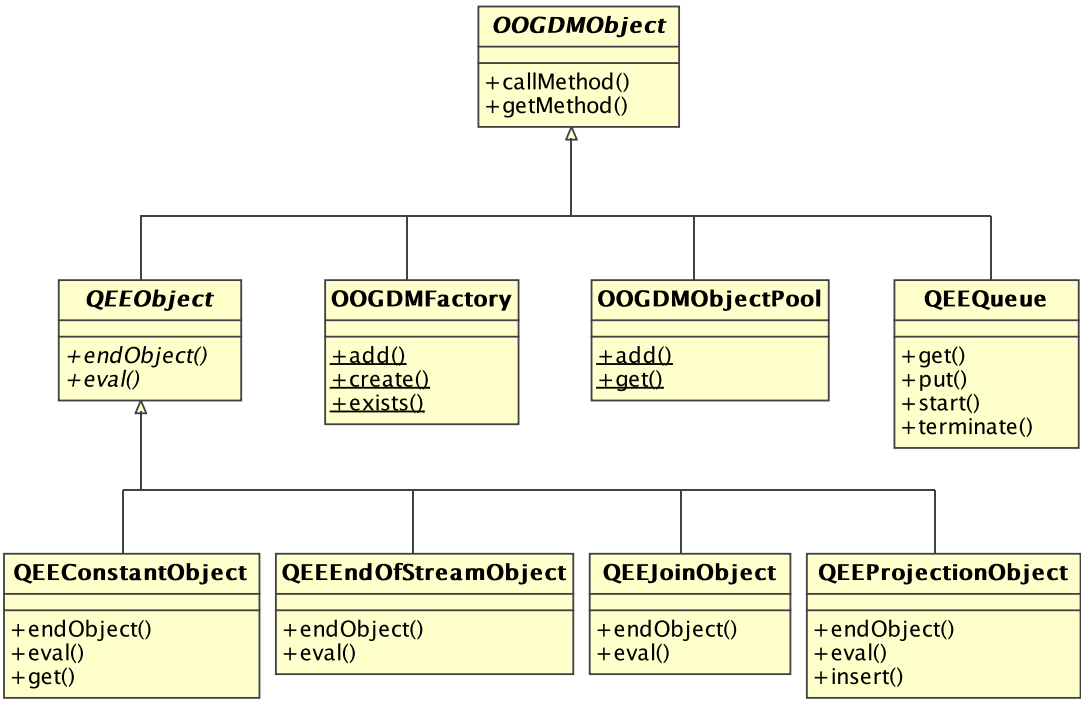
Lexikalische Analyse: Zu Beginn wird ein vorliegender textueller Ausführungsplan einer lexikalischen Analyse [Pax95] unterzogen. Dabei werden so genannte *Token* bestimmt, aus denen sich der textuelle Ausführungsplan zusammensetzt.

Syntaktische Analyse: In diesem zweiten Schritt werden die während der lexikalischen Analyse ermittelten Token hinsichtlich ihres Auftretens im textuellen Ausführungsplan auf syntaktische Korrektheit überprüft [DS02].

Objekterzeugung: Nachdem ein gültiger Konstruktoraufruf erkannt worden ist, wird ein neues Objekt aus der vorliegenden textuellen Beschreibung erzeugt. Die erforderlichen Argumente für Konstruktoraufrufe einzelner Klassen finden sich schon in Abschnitt 3.2.3; daher werden weiter unten nur die übrigen Voraussetzungen für die Objekterzeugung genannt.

Objektregistrierung: Nach der Erzeugung wird schließlich jedes Objekt bei einer zentralen Stelle, und zwar der Klasse `OOGDMObjectPool`, unter dem ebenfalls im Ausführungsplan angegebenen Bezeichner registriert. Dadurch wird es möglich, als Parameterwerte für einen Konstruktoraufruf ebenfalls Bezeichner anstelle weiterer Konstruktoraufrufe anzugeben.

Abbildung 3.5 Klassen für Hilfsaufgaben in objektbasierten Ausführungsplänen mit Methoden



Im Folgenden werden diese vier Schritte genauer beschrieben, zuvor sollen jedoch noch einige wesentliche Voraussetzungen aufgeführt werden.

Voraussetzungen zur Erzeugung objektbasierter Ausführungspläne

Bei jedem Start von GOODAC sorgen alle Klassen, die während der Abarbeitung objektbasierter Ausführungspläne eine Rolle spielen – nämlich alle Unterklassen von `OOGDMObject`, also sämtliche Klassen, deren Instanzen in der Datenbank gespeichert werden können, und damit insbesondere alle Klassen zur Erzeugung von Knoten- und Warteschlangenobjekten (siehe auch Abschnitt 3.2.2 und Abschnitt 3.2.3) – dafür, dass sie sich selbst bei der Klasse `OOGDMFactory` über deren Klassenmethode `add` (siehe auch Abbildung 3.5) registrieren. Dadurch wird sichergestellt, dass bereits während der lexikalischen Analyse eines textuellen Ausführungsplans sämtliche Klassennamen bekannt sind, somit wird also eine korrekte Unterscheidung zwischen Bezeichnern und Klassennamen gewährleistet.

Zudem muss jede der oben genannten Klassen eine Methode `tcreate` – diese Abkürzung steht für *textuelles Erzeugen* oder *textual create* – bereithalten, die eine Erzeugung neuer Objekte dieser Klasse auf Grundlage einer übergebenen Argumentliste – also einem Objekt des Typs `OOGDMArguments` – ermöglicht. Eine derartige Argumentliste kann etwa aus den Parametern eines Konstruktoraufrufs im Rahmen textueller Ausführungspläne bestehen. Auch diese Methode wird gemeinsam mit dem Klassennamen als Schlüssel bei der Klasse `OOGDMFactory` registriert und im Falle der Anforderung eines neuen Objekts durch die statische Methode `create` der Klasse `OOGDMFactory` aufgerufen. Dabei führt der als Parameterwert an `create` übergebene Klassenname dazu, dass die Methode `tcreate` zur Erzeugung neuer Instanzen der Klasse diesen Namens mit den ebenfalls an `create` übergebenen Argumenten aufgerufen wird. Als Ergebnis wird dadurch ein neues Objekt mit Obertyp `OOGDMObject` erzeugt.

So besitzen etwa die in Beispiel 3.16 auftretenden Klassen `QEESortNode` und `OOGDMInt` jeweils eine Methode `tcreate`, die unter dem jeweiligen Klassennamen bei der Klasse `OOGDMFactory` registriert ist. Die Methode `OOGDMInt::tcreate` erwartet eine Zeichenkette als Argument, die sie in eine ganze Zahl umwandelt und als Parameter für einen Konstruktoraufruf der Klasse `OOGDMInt` einsetzt. Diese Methode liefert also eine neue Instanz der Klasse `OOGDMInt` zurück. Ebenso erzeugt die Methode `QEESortNode::tcreate` ausgehend von zwei Objekten des Typs `OOGDMString` und einem Objekt des Typs `OOGDMInt` eine neue Instanz der Klasse `QEESortNode`.

Die Klasse `OOGDMFactory` stellt in diesem Kontext also eine parametrisierte Methode `create` zur Erzeugung von Objekten bereit, die in ihrem Verhalten im Wesentlichen der von Gamma *et. al.* vorgestellten

parametrisierten Fabrikmethode (der so genannten *Factory Method* [GHJV95, 3]) folgt. Die übrigen in Abbildung 3.5 dargestellten Methoden werden im weiteren Verlauf dieses Kapitels näher beschrieben.

Lexikalische Analyse

Während der lexikalischen Analyse [Pax95] eines textuellen Ausführungsplans werden alle auftretenden Elemente im Wesentlichen in vier Gruppen unterteilt:

Argumentwerte: Alle in Anführungszeichen auftretenden alphanumerischen Folgen werden als Argumentwert identifiziert. Aus diesem Grund liefert die lexikalische Analyse das Token `VALUE` zurück.

Bezeichner: Alle mit einem Buchstaben beginnenden alphanumerischen Folgen werden als Bezeichner identifiziert, falls sie der Klasse `OOGDMFactory` nicht bekannt sind; der Aufruf der Methode `exists` also `false` ergibt. In diesem Fall wird das Token `VARIABLE` erzeugt.

Klassennamen: Diejenigen mit einem Buchstaben beginnenden alphanumerischen Folgen, die als Klassenname bei der Klasse `OOGDMFactory` registriert sind, können als solche identifiziert werden, sodass das Token `CLASSNAME` zurückgegeben wird.

Sonstige: Alle weiteren Zeichen wie beispielsweise öffnende und schließende Klammern sowie Kommata und Semikola werden ohne Erzeugung eines speziellen Tokens im Rahmen der lexikalischen Analyse weitergeleitet.

Während der lexikalischen Analyse werden nun unter Ignorierung sämtlicher Leerzeichen und Zeilenumbrüche die oben genannten Gruppen von Elementen identifiziert. Sollte ein unbekanntes Element auftreten, gilt der zu untersuchende textuelle Ausführungsplan als fehlerhaft und wird nicht weiter betrachtet. Andernfalls liefert die lexikalische Analyse jeweils die Elemente des textuellen Ausführungsplans gemeinsam mit den gegebenenfalls erzeugten Token zurück, sodass beide der nachfolgenden syntaktischen Analyse zur Verfügung stehen.

Syntaktische Analyse

Im Rahmen der syntaktischen Analyse [DS02] wird überprüft, ob die während der lexikalischen Analyse ermittelten Token und sonstigen Zeichen auf eine korrekte Art und Weise verwendet werden. Die erlaubten Abfolgen werden durch Produktionen einer kontextfreien Grammatik (Abbildung 3.1 auf Seite 24) definiert, die bereits bei der Beschreibung textueller Ausführungspläne in Abschnitt 3.2.1 näher vorgestellt wurden. Sollte sich die Abfolge der gefundenen Token und weiteren Zeichen eines textuellen Ausführungsplans nicht durch die angegebenen Produktionen herleiten lassen, so wird die Betrachtung des textuellen Ausführungsplans während der syntaktischen Analyse mit einem Fehler abgebrochen.

Objekterzeugung

Bevor ein konkretes Objekt erzeugt werden kann, müssen während der syntaktischen Analyse gegebenenfalls erst alle angegebenen Argumente in eine Parameterliste vom Typ `OOGDMArguments` eingefügt werden. Anschließend sorgt die Behandlung der Regel

‹object› ::= `CLASSNAME` ’(’ [‹argumentList›] ’)’

dafür, dass die statische Methode `OOGDMFactory::create` mit dem vorliegenden Klassennamen und der eventuell leeren Parameterliste aufgerufen wird. Intern ruft nun diese Methode `OOGDMFactory::create` die unter dem vorliegenden Klassennamen gespeicherte Methode `tcreate` zur Erzeugung eines neuen Objekts für die ebenfalls übergebene Parameterliste auf und liefert so schließlich eine neue Instanz der vorliegenden Klasse zurück.

Neben diesen explizit formulierten Konstruktoraufrufen werden implizit aus allen weiteren auftretenden Zeichenketten korrespondierende Objekte des Typs `OOGDMString` erzeugt, die diese Zeichenketten repräsentieren.

Objektregistrierung

Bereits bei der Vorstellung der einzelnen Knotentypen in Abschnitt 3.2.3 ist deutlich geworden, dass viele Konstruktoren unter anderem die Bezeichner anderer Objekte des vorliegenden textuellen Ausführungsplans als Parameter erwarten. Aus diesem Grund werden die jeweils neu erzeugten Objekte bei Behandlung der Regel

‹variableObjectPair› ::= ‹VARIABLE› ‹object› ';'

an den entsprechenden Bezeichner gebunden und anschließend über die statische Methode `add` in die Klasse `OOGDMObjectPool` (siehe auch Abbildung 3.5), die im Wesentlichen ein Dictionary [OW96, 1.6] repräsentiert, eingefügt. Diese Bezeichner müssen innerhalb eines textuellen Ausführungsplans eindeutig sein, weil sie beim späteren Auslesen des zugehörigen Objekts aus der Klasse `OOGDMObjectPool` als Suchschlüssel dienen. Somit können alle weiteren Objekte, die bei der Konstruktion nur den Namen eines assoziierten Objekts anstelle des Objekts selbst erhalten haben, diesen Namen später über die Klassenmethode `get`der Klasse `OOGDMObjectPool` zu einem konkreten Objekt auflösen. Insbesondere müssen im Konstruktoraufruf angegebene Bezeichner zum Zeitpunkt der Erzeugung eines neuen Objekts noch nicht in der Klasse `OOGDMObjectPool` existieren, solange gewährleitet ist, dass sie bei der weiteren Analyse des textuellen Ausführungsplans dort registriert werden. Dadurch kann zum einen die Reihenfolge der Bezeichner-Objekt-Paare in textuellen Ausführungsplänen vernachlässigt werden, während zum anderen auch zirkuläre Beziehungen zwischen einzelnen Objekten – beispielsweise zwischen einer Instanz der Klasse `QEEPrintNode` und einer Instanz der Klasse `QEEPrintExpressionNode` (siehe auch Beispiel 3.19 auf Seite 43) – besser unterstützt werden. Alle Instanzen dieser Klassen werden mit einem Bezeichner anstelle eines konkreten Objekts als Parameterwert gebildet und erst nach der Erzeugung aller erforderlichen Instanzen werden die Bezeichner im Rahmen der Ausführung objektorientierter Ausführungspläne in konkrete assoziierte Objekte umgesetzt.

3.2.5 Abarbeitung objektbasierter Ausführungspläne

Nachdem alle Objekte eines objektbasierten Ausführungsplans erzeugt und unter ihrem jeweiligen Bezeichner bei der Klasse `OOGDMObjectPool` registriert worden sind, kann die Abarbeitung dieses objektbasierten Ausführungsplans beginnen. Im Folgenden wird zuerst das allgemeine Konzept der Abarbeitung beschrieben, bevor der eigentliche Ablauf näher vorgestellt wird.

Allgemeines Konzept der Abarbeitung objektbasierter Ausführungspläne

Die einzelnen auf Objektströmen arbeitenden Knotenobjekte eines objektbasierten Ausführungsplans werden wie bereits in Abschnitt 3.2.2 dargestellt über Kantenobjekte miteinander verknüpft.

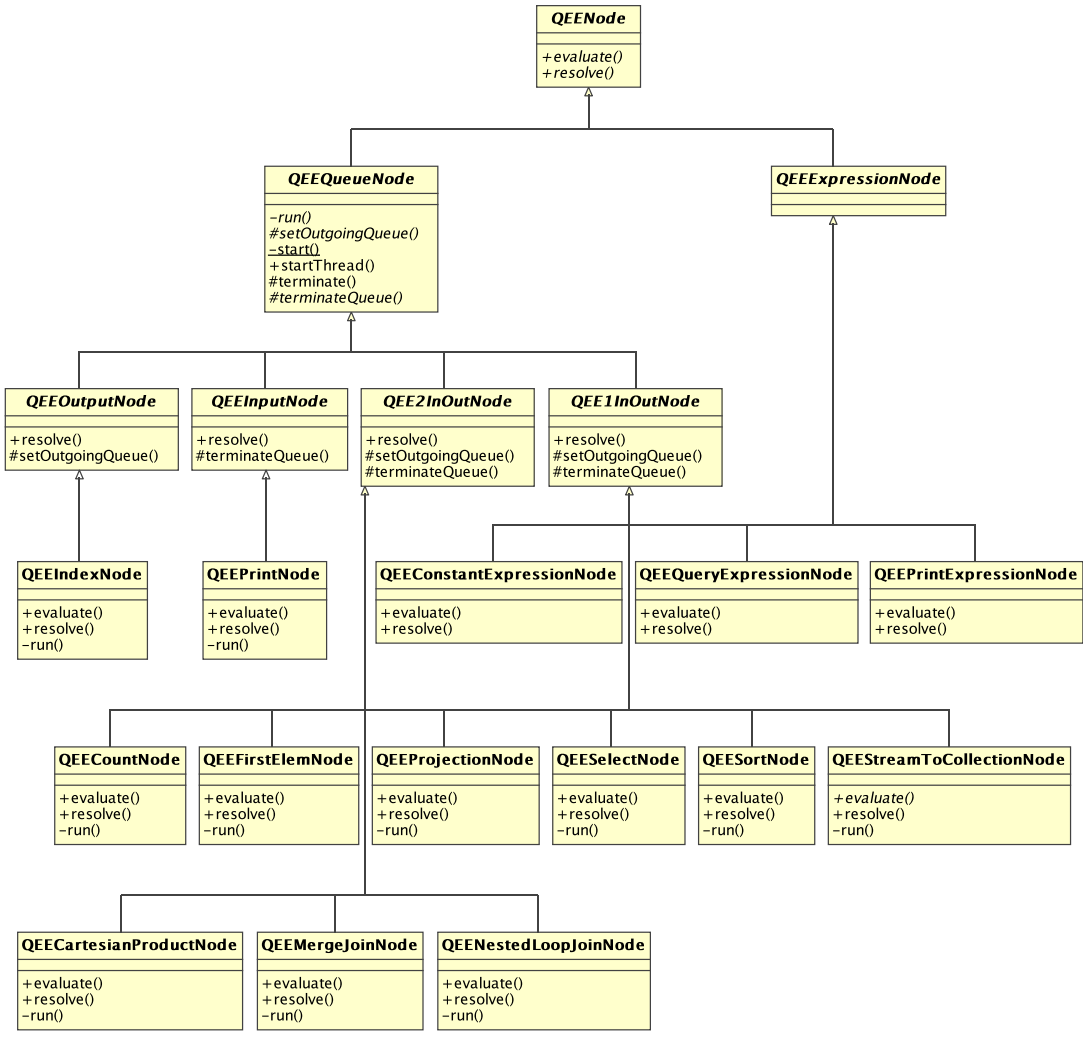
Dabei wird die Strategie eines Produzenten-getriebenen Durchleitens (siehe auch Abschnitt 3.1.1) verfolgt. Das bedeutet, dass jeder auf Objektströmen arbeitende Knoten solange Daten erzeugt und in sein ausgehendes Kantenobjekt einfügt, bis die Kapazität dieses Kantenobjekts erschöpft ist. In diesem Fall blockiert das Kantenobjekt den Prozess des Knotenobjekts, um aktives Warten (so genanntes *busy waiting* [SGG00, 7.5.2]) zu vermeiden. Erst wenn das Kantenobjekt wieder Platz für neue Objekte besitzt, reaktiviert es den Prozess des zuvor blockierten Knotenobjekts.

Damit möglichst viele auf Objektströmen arbeitende Knotenobjekte parallel aktiv sein können und damit überflüssige Kontextwechsel vermieden werden, erfolgt der Start dieser Knotenobjekte jeweils in einem eigenen leichtgewichtigen Prozess (einem so genannten *Thread* [SGG00, 5]). Dadurch kann beispielsweise ein Knotenobjekt seine ausgehende Warteschlange füllen, während ein zweites Knotenobjekt gleichzeitig Daten aus eben diesem gemeinsamen Kantenobjekt ausliest. Die in Abbildung 3.6 gezeigte Methode `startThread` eines auf Objektströmen arbeitenden Knotenobjekts – also einer Instanz mit Oberklasse `QEEQueueNode` – dient genau diesem Zweck, nämlich der Erzeugung eines neuen leichtgewichtigen Prozesses für die Arbeit dieses Objekts.

Ablauf der Abarbeitung objektbasierter Ausführungspläne

Um die Abarbeitung eines objektbasierten Ausführungsplans zu starten, muss für einen Knoten mit Obertyp `QEEQueueNode` die Methode `startThread` (siehe auch Abbildung 3.6) aufgerufen werden. Dazu wird das entsprechende Objekt – meistens vom Typ `QEEPrintNode` (siehe auch Seite 31) – über seinen Bezeichner aus der Klasse `OOGDMObjectPool` ausgelesen, bevor der Methodenaufruf erfolgt. Innerhalb dieser Methode

Abbildung 3.6 Hierarchie zugelassener Knotentypen in ausführbaren Anfragegraphen mit Methoden



wird über die private Methode `start` ein neuer leichtgewichtiger Prozess für dieses Knotenobjekt erzeugt, dessen eigentliche Arbeit schließlich durch einen Aufruf der Methode `run` gestartet wird.

Nun ist dieses Objekt dafür verantwortlich, die weitere Abarbeitung des objektbasierten Ausführungsplans in Gang zu setzen. Dazu müssen insbesondere alle bisher nur über ihren Bezeichner repräsentierten Verweise auf weitere Kanten- und Knotenobjekte durch konkrete Objektreferenzen ersetzt werden. Das geschieht durch die Methode `resolve` (siehe auch Abbildung 3.6), die die jeweils für die Knoten eines Typs vorliegenden Bezeichner in Objektreferenzen umsetzen kann. Das erste aus der Klasse `OOGDMObjectPool` ausgelesene Objekt setzt also seine Bezeichner mittels `resolve` um und aktiviert anschließend seine eingehenden Kantenobjekte, also Instanzen der Klasse `QEEQueue` (siehe auch Seite 28), über deren Methode `start` (siehe auch Abbildung 3.5). Ein Kantenobjekt wiederum besitzt einen Verweis auf das zweite auf Objektströmen arbeitende Knotenobjekt, das im Rahmen der Abarbeitung des objektbasierten Ausführungsplans die repräsentierte Warteschlange mit Elementen füllen wird. Daher macht ein Kantenobjekt sich selbst über die Methode `setOutgoingQueue` als ausgehendes Warteschlangenobjekt dieses assoziierten Knotenobjekts bekannt und ruft wiederum die Methode `startThread` des Knotenobjekts auf, sodass sich dieser Prozess rekursiv fortsetzt, bis alle Kanten- und Knotenobjekte aktiviert worden sind und die erforderlichen Assoziationen bestehen.

Diejenigen Knotenobjekte, die nicht auf Objektströmen arbeiten – also alle Instanzen zur Oberklasse `QEEExpressionNode` (siehe auch Abbildung 3.6) –, lösen ihre Bezeichner mittels `resolve` erst bei Bedarf auf, und zwar genau dann, wenn das erste Mal ihre Methode `evaluate` aufgerufen wird.

Sobald nun ein auf Objektströmen arbeitendes Knotenobjekt bereit ist, seine eigentliche Arbeit aufzunehmen, liest es die gegebenenfalls erforderlichen aktuellen Objekte seiner Eingabeströme über die Methode `get` der entsprechenden Kantenobjekte ein und verarbeitet sie. Als Ergebnis dieser Behandlung werden

die Resultatobjekte nacheinander über die Methode `put` in die ausgehende Warteschlange eingefügt.

Im Rahmen dieser Verarbeitung kann es erforderlich sein, bestimmte Aufgaben – etwa die Auswertung eines Selektionsprädikats im Fall einer Instanz der Klasse `QEESelectNode` (siehe auch Seite 35) oder den Vergleich zweier Objekte bezüglich einer Sortierbedingung im Fall einer Instanz der Klasse `QEESortNode` (siehe auch Seite 35) – an assoziierte Knotenobjekte mit Obertyp `QEEExpressionNode` zu delegieren. Dieses geschieht über den Aufruf der entsprechenden Methode `evaluate` (siehe auch Abbildung 3.6), die wiederum über die Methode `evaluate` des aufrufenden Knotenobjekts die zu bearbeitenden Elemente des aktuellen Objektstroms erhält.

Die Objekte mit zugehöriger Oberklasse `QEEExpressionNode` verarbeiten häufig Instanzen zu einer Unterklasse von `QEEObject` (siehe auch Seite 27), die weitere Objekte kapseln. Um einen einheitlichen Zugriff auf diese enthaltenen Instanzen zu gewährleisten, besitzen derartige Objekte mit Obertyp `QEEObject` die Methode `eval` (siehe auch Abbildung 3.5), die jeweils das angeforderte gekapselte Element zurückliefert – also beispielsweise im Fall einer Instanz der Klasse `QEEJoinObject` (siehe auch Seite 28) eines der beiden gekapselten Objekte und im Fall einer Instanz der Klasse `QEEProjectionObject` (siehe auch Seite 28) ein bestimmtes Ergebnis einer vorher durchgeführten Projektion.

Jedes auf Objektströmen arbeitende Knotenobjekt führt seine konkrete Aufgabe – also beispielsweise die Berechnung einer Verbundoperation nach dem Merge-Join-Algorithmus im Fall einer Instanz der Klasse `QEEMergeJoinNode` (siehe auch Seite 32) – solange durch, bis es entweder durch sein ausgehendes Kantenobjekt über den Aufruf der Methode `terminate` (siehe auch Abbildung 3.6) gestoppt wird oder aber seine Eingabeströme versiegt sind und daher nur noch Instanzen der Klasse `QEEEndOfStreamObject` enthalten, deren Methode `endObject` (siehe auch Abbildung 3.5) den Wert `true` zurückliefert. Da ein auf Objektströmen arbeitendes Knotenobjekt für jedes Element des eingehenden Objektstroms – eine Instanz mit Oberklasse `QEEObject` – den Rückgabewert der Methode `endObject` abfragt, kann es so erkennen, ob der Eingabestrom keine Elemente mehr liefern wird. Falls ein auf Objektströmen arbeitendes Knotenobjekt durch sein Kantenobjekt gestoppt wird oder feststellt, dass sein Eingabestrom versiegt ist, stellt es seine Arbeit ein und stoppt über seine Methode `terminateQueue` auch alle eingehenden Kantenobjekte durch Aufruf von `terminate` (siehe auch Abbildung 3.5), sodass rekursiv die Arbeit aller weiteren auf Objektströmen arbeitenden Knotenobjekte sowie der verbindenden Kantenobjekte beendet wird. Schließlich ist die Abarbeitung des gesamten objektbasierten Ausführungsplans beendet.

3.2.6 Beispiele für die Funktionsweise der Ausführungsebene in GOODAC

Nachdem im bisherigen Verlauf dieses Kapitels die allgemeine Funktionsweise der Ausführungsebene in GOODAC vorgestellt worden ist, wird in diesem Abschnitt ein ausführliches Beispiel gezeigt, um die zuvor getätigten generellen Aussagen zu veranschaulichen. Dazu wird eine Anfrage betrachtet, deren Ziel es ist, alle Paare von in der Datenbank gespeicherten Städten der Bundesländer *NRW* und *Bayern* zu bestimmen, die die gleiche Einwohnerzahl besitzen. Eine entsprechende OOGQL-Anfrage (siehe auch Abschnitt 2.2) könnte beispielsweise wie in Beispiel 3.18 dargestellt formuliert werden.

Beispiel 3.18 OOGQL-Anfrage zum durchgängigen Beispiel in Abschnitt 3.2.6

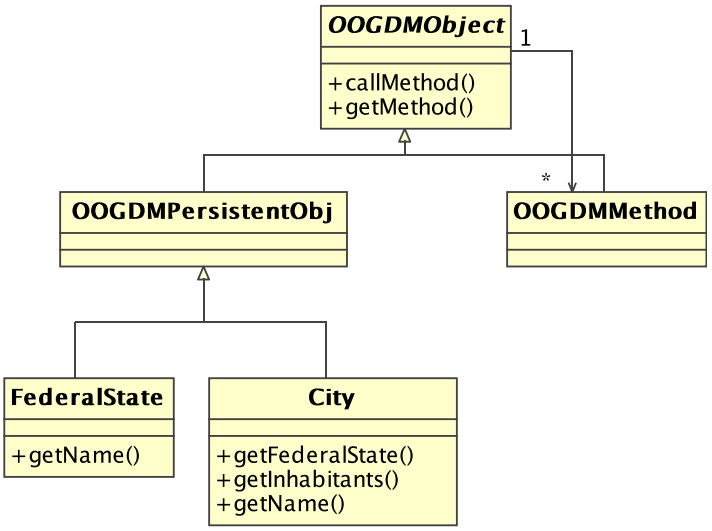
```
select *
from City c1, City c2
where ((c1.getFederalState().getName() == "NRW") and ((c2.getFederalState().getName() == "Bayern")
      and (c1.getInhabitants() == c2.getInhabitants())));
```

Die Gliederung der nachfolgenden Beschreibung folgt der bereits in den Abschnitten 3.2.4 und 3.2.5 erfolgten Aufteilung: Im Vorfeld werden wesentliche Voraussetzungen besprochen und die in der Anfrage vorkommenden Anwendungsklassen beschrieben. Anschließend wird die lexikalische Analyse thematisiert, bevor ausgehend von der syntaktischen Analyse die Erzeugung und die Registrierung von Objekten Erwähnung finden. Darauf aufbauend erfolgt abschließend eine ausführliche Erläuterung der Abarbeitung des dann erzeugten objektbasierten Ausführungsplans.

Voraussetzungen zur Erzeugung und Abarbeitung des benötigten objektbasierten Ausführungsplans

Neben den bereits in Abschnitt 3.2.4 erwähnten allgemeinen Voraussetzungen werden an dieser Stelle im Rahmen der Betrachtung eines speziellen Ausführungsplans weitere Anforderungen formuliert, die sich am

Abbildung 3.7 Verwendete Anwendungsklassen



besten an diesem konkreten Beispiel verdeutlichen lassen. Sie können allerdings leicht auf den allgemeinen Fall übertragen werden.

So werden im vorliegenden Anfrageplan Instanzen der Klasse `City` (siehe auch Abbildung 3.7) aus der Datenbank ausgelesen, um während der Abarbeitung des zu erzeugenden objektbasierten Ausführungsplans unter anderem ihr assoziiertes Objekt vom Typ `FederalState` zu betrachten. Da die auftretenden Methodenaufrufe dieser Objekte erst durch den Ausführungsplan festgelegt werden, muss jedes Objekt, das in diesem Sinne während der Bearbeitung eines Ausführungsplans behandelt wird, den Namen einer Methode in einen konkreten Methodenaufruf umsetzen können. Dazu besitzt jede Instanz einer von `OOGDMObject` abgeleiteten Klasse die Methode `getMethod`, die zu einem übergebenen Bezeichner – also dem Methodennamen – ein Objekt vom Typ `OOGDMMethod` zurückliefert, das wiederum einen Methodenaufruf kapselt. So besitzt also die Klasse `City` interne Verweise unter den Namen `getInhabitants`, `getName` und `getFederalState`, die auf entsprechende Methodenobjekte verweisen. Während der Abarbeitung objektbasierter Ausführungspläne können also nur Methoden von Anwendungsklassen verwendet werden, die sich über entsprechende Bezeichner ermitteln lassen. Ein direkter Aufruf einer derartigen Operation wird über die Methode `callMethod` erreicht.

Ferner erfordern die ersten beiden Zeilen des in Beispiel 3.19 dargestellen Ausführungsplans, dass sich eine Datenbankwurzel [Pro03, 4] mit dem Bezeichner `City` in der aktuellen Datenbank befindet, welche Instanzen der Klasse `City` enthält, auf die über eine Indexstruktur vom Typ `OOGDMListIndex` – siehe zur Klasse `OOGDMListIndex` auch die Beschreibung des Knotentyps `QEEIndexNode` auf Seite 30 – zugegriffen werden kann. Weitere Details zum Konzept und zur Implementation von Indexstrukturen in GOODAC liefern Melnikov [Mel02, 5.5] und Steinberg [Ste96, 3.4].

Lexikalische Analyse

Es gibt im Rahmen der Anfragebearbeitung in GOODAC zahlreiche textuelle Ausführungspläne, in die die eingangs gestellte OOGQL-Anfrage (siehe Beispiel 3.18) überführt werden kann. Der in Beispiel 3.19 gezeigte textuelle Ausführungsplan repräsentiert daher nur ein mögliches Vorgehen, um die zu Beginn formulierte OOGQL-Anfrage zu beantworten. Er wird im Rahmen der Anfragebearbeitung von der Komponente zur Anfrageoptimierung (siehe auch Kapitel 5) aus einem die ursprüngliche OOGQL-Anfrage repräsentierenden Algebraausdruck erzeugt (siehe auch Kapitel 4). Sobald er vorliegt, kann die Komponente zur Abarbeitung von Ausführungsplänen mit der lexikalischen Analyse beginnen.

Die lexikalische Analyse strukturiert den textuellen Ausführungsplan wie bereits oben erwähnt unter Zuhilfenahme so genannter *Token* (siehe auch Seite 38). Die erste Zeile des in Beispiel 3.19 präsentierten textuellen Ausführungsplans wird gemäß der in Abschnitt 3.2.4 dargestellten Kriterien wie in Beispiel 3.20 gezeigt strukturiert.

Direkt zu Beginn findet sich mit `myIndexCity1` ein Bezeichner, der zuvor nicht über die Klassenmethode `add` in der Klasse `OOGDMFactory` eingetragen worden ist, daher wird das Token `VARIABLE` erzeugt. Die dann folgende Zeichenkette `QEEIndexNode` wird als Klassenname erkannt, für den ein Konstruktorauf-

Beispiel 3.19 Ein möglicher textueller Ausführungsplan zur OOGQL-Anfrage aus Beispiel 3.18

```
myIndexCity1          QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myIndexCity2          QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myQueueCity1          QEEQueue(myIndexCity1);
myQueueCity2          QEEQueue(myIndexCity2);
mySelectNRW           QEESelectNode(myQueueCity1, myQueryNRW1);
myQueryNRW1           QEEQueryExpressionNode(myQueryNRW2,
                                             OOGDMUShort("0"),
                                             "operator==",
                                             myConstNRW);
myQueryNRW2           QEEQueryExpressionNode(myQueryNRW3,
                                             OOGDMUShort("0"),
                                             "getName");
myQueryNRW3           QEEQueryExpressionNode(mySelectNRW,
                                             OOGDMUShort("0"),
                                             "getFederalState");
myConstNRW            QEEConstantExpressionNode(OOGDMString("NRW"));
mySelectBayern        QEESelectNode(myQueueCity2, myQueryBayern1);
myQueryBayern1        QEEQueryExpressionNode(myQueryBayern2,
                                             OOGDMUShort("0"),
                                             "operator==",
                                             myConstBayern);
myQueryBayern2        QEEQueryExpressionNode(myQueryBayern3,
                                             OOGDMUShort("0"),
                                             "getName");
myQueryBayern3        QEEQueryExpressionNode(mySelectBayern,
                                             OOGDMUShort("0"),
                                             "getFederalState");
myConstBayern         QEEConstantExpressionNode(OOGDMString("Bayern"));
myQueueCityNRW        QEEQueue(mySelectNRW);
myQueueCityBayern     QEEQueue(mySelectBayern);
myNestedLoopJoin      QEENestedLoopJoinNode(myQueueCityNRW,
                                            myQueueCityBayern,
                                            myQueryEquality);
myQueryEquality       QEEQueryExpressionNode(myQueryInhabitNRW,
                                             OOGDMUShort("0"),
                                             "operator==",
                                             myQueryInhabitBayern);
myQueryInhabitNRW     QEEQueryExpressionNode(myNestedLoopJoin,
                                             OOGDMUShort("0"),
                                             "getInhabitants");
myQueryInhabitBayern  QEEQueryExpressionNode(myNestedLoopJoin,
                                             OOGDMUShort("1"),
                                             "getInhabitants");
myQueueJoinResult     QEEQueue(myNestedLoopJoin);
myStreamToCollNode    QEEStreamToCollectionNode(myQueueJoinResult,OOGDMBag());
myQueueBagResult      QEEQueue(myStreamToCollNode);
myPrintNode           QEEPrintNode(myQueueBagResult, myPrintExp);
myPrintExp            QEEPrintExpressionNode(myPrintNode);
```

ruf ausgehend von einer textuellen Beschreibung möglich ist, weil sie mit ihrer Methode `tcreate` in der Klasse `OOGDMFactory` vorhanden ist, und somit der Aufruf der dort angesiedelten Klassenmethode `exists` den Wert `true` ergibt. Also wird das Token `CLASSNAME` erzeugt. Die nachfolgende öffnende Klammer gilt als sonstiges Zeichen und wird ohne Erzeugung eines speziellen Tokens weitergeleitet. Anschließend folgen die beiden Klassennamen `OOGDMListIndexScan` und `OOGDMListIndex` sowie zwei weitere öffnende Klammern, bevor die in Anführungszeichen eingeschlossene alphanumerische Folge `City` auftritt. Diese

Beispiel 3.20 Ergebnis der lexikalischen Analyse der ersten Zeile aus Beispiel 3.19

```
VARIABLE CLASSNAME '(' CLASSNAME '(' CLASSNAME '(' VALUE ')' ')' ')' ';'
```

Beispiel 3.21 Ergebnis der lexikalischen Analyse des textuellen Ausführungsplans aus Beispiel 3.19

```
VARIABLE CLASSNAME '(' CLASSNAME '(' CLASSNAME '(' VALUE ')' ')' ')' ';'
VARIABLE CLASSNAME '(' CLASSNAME '(' CLASSNAME '(' VALUE ')' ')' ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ',' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ')' ';'
VARIABLE CLASSNAME '(' CLASSNAME '(' VALUE ')' ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ',' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ')' ';'
VARIABLE CLASSNAME '(' CLASSNAME '(' VALUE ')' ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ')' ';'
VARIABLE CLASSNAME '('VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' VARIABLE ',' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ',' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' VALUE ')' ',' VALUE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' CLASSNAME '(' ')' ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ',' VARIABLE ')' ';'
VARIABLE CLASSNAME '(' VARIABLE ')' ';'
```

wird als Argumentwert identifiziert, sodass das Token `VALUE` erzeugt wird. Die drei dahinter auftauchenden schließenden Klammern werden wiederum als sonstige Zeichen gewertet und einfach weitergeleitet, ebenso wie das abschließende Semikolon.

Weil im Rahmen der lexikalischen Analyse keine Unterteilung nach syntaktischen oder gar semantischen Gesichtspunkten stattfindet, wird der gesamte textuelle Ausführungsplan als ein fortlaufender Ausdruck verstanden, der sich gemäß festgelegter Erkennungsmerkmale durch Token strukturieren lässt. Beispiel 3.21 zeigt schließlich das endgültige Ergebnis der lexikalischen Analyse des gesamten textuellen Ausführungsplans aus Beispiel 3.19.

Syntaktische Analyse

Während der syntaktischen Analyse wird nun überprüft, ob sich die während der vorhergehenden lexikalischen Analyse ermittelte Abfolge von Token und sonstigen Zeichen aus Produktionen einer kontextfreien Grammatik (Abbildung 3.1 auf Seite 24) ableiten lässt.

Als Startsymbol dient das Nichtterminalsymbol `variableObjectPairList`, das sich gemäß der nachfolgenden Produktionen

```
<variableObjectPairList>  ::=  <variableObjectPair> |
                               <variableObjectPair> <variableObjectPairList>
<variableObjectPair>      ::=  VARIABLE <object> ';'
```

auf ein einzelnes Paar – bestehend aus einem Bezeichner und einem Objekt mit einem abschließendem Semikolon – oder eine Liste derartiger Paare ableiten lässt.

Der nähere Aufbau eines solchen Objekts wird über

```
<object> ::= CLASSNAME '(' [ <argumentList> ] ')'
```

als Konstruktoraufruf mit einer optionalen Argumentliste spezifiziert. Diese Argumentliste enthält laut

Beispiel 3.22 Ableitung von Beispiel 3.20 aus dem Startsymbol der Grammatik

```
<variableObjectPairList> → <variableObjectPair>
                         → VARIABLE <object> ';'
                         → VARIABLE CLASSNAME '(' [ <argumentList> ] ')' ';'
                         → VARIABLE CLASSNAME '(' <argument> ')' ';'
                         → VARIABLE CLASSNAME '(' <object> ')' ';'
                         → VARIABLE CLASSNAME '(' CLASSNAME '(' [ <argumentList> ] ')' ')' ';'
                                                ...
                  → VARIABLE CLASSNAME '(' CLASSNAME '(' CLASSNAME '(' <argument> ')' ')' ')' ';'
                  → VARIABLE CLASSNAME '(' CLASSNAME '(' CLASSNAME '(' VALUE ')' ')' ')' ';'
```

```
<argumentList> ::= <argument> |
                   <argument> ',' <argumentList>
<argument>     ::= VALUE |
                   VARIABLE |
                   <object>
```

ein oder mehrere Elemente, wobei jedes Element einen Wert, einen Bezeichner oder einen erneuten Konstruktoraufruf darstellen muss.

Das nachfolgende Beispiel 3.22 zeigt ausführlich, wie sich der in Beispiel 3.20 dargestellte Ausdruck aus dem Startsymbol unter Anwendung der zuvor näher beschriebenen Produktionen der kontextfreien Grammatik ableiten lässt.

Die weiteren in Beispiel 3.21 gezeigten Teilausdrücke lassen sich ebenfalls korrekt aus der Grammatik herleiten. Gleiches gilt auch für den gesamten Ausdruck als Ganzes, sodass der in Beispiel 3.19 präsentierte textuelle Ausführungsplan erfolgreich auf syntaktische Korrektheit überprüft werden kann.

Objekterzeugung und Objektregistrierung

Nachdem zuvor die Analyse des textuellen Ausführungsplans im Vordergrund gestanden hat, wird nun näher beschrieben, wie die einzelnen für den objektbasierten Ausführungsplan erforderlichen Objekte erzeugt und bei der Klasse `OOGDMObjectPool` registriert werden. Dazu muss jede Klasse, deren Name im Rahmen eines textuellen Ausführungsplans auftritt, bereits im Vorfeld unter ihrem Klassennamen eine Methode zur Erzeugung einer neuen Instanz aus einer textuellen Beschreibung bei der Klasse `OOGDMFactory` registriert haben (siehe auch Abschnitt 3.2.4).

Wird nun beispielsweise das Ergebnis der lexikalischen Analyse der ersten Zeile des in Beispiel 3.19 gezeigten textuellen Ausführungsplans erfolgreich einer syntaktischen Analyse unterzogen (siehe auch Beispiele 3.20 und 3.22), so wird im Rahmen der syntaktischen Analyse ausgehend von den innersten Elementen jeder Konstruktoraufruf in ein konkretes Objekt umgesetzt. Der Teilausdruck

OOGDMListIndex("City")

beispielsweise wird – wie oben gezeigt – durch die lexikalische Analyse in die Folge

CLASSNAME '(' VALUE ')'

transformiert, welche von der syntaktischen Analyse als durch die Produktion

<object> ::= CLASSNAME '(' [<argumentList>] ')'

ableitbar erkannt wird. Daher wird nun die Klassenmethode `create` der Klasse `OOGDMFactory` mit dem vorliegenden Klassennamen `OOGDMListIndex` und dem Argument `City`, welches in einer Instanz der Klasse `OOGDMString` gekapselt wird, aufgerufen. Diese Methode aktiviert nun wiederum die in der Klasse `OOGDMFactory` unter dem Bezeichner `OOGDMListIndex` gespeicherte Methode zur Erzeugung eines neuen Objekts aus einer textuellen Beschreibung, wobei zusätzlich das Argument `City` übergeben wird. Als Ergebnis erzeugt diese Operation demzufolge ein neues Objekt vom Typ `OOGDMListIndex` [Mel02, 4.4], das einen Index über alle Instanzen der Klasse `City` darstellt. Dabei bildet diese Instanz der Klasse `OOGDMListIndex` intern nur eine transiente Hülle für die in der Datenbank gespeicherte, vom Datenbanksystem ObjectStore bereitgestellte Datenbankwurzel [Pro03, 4]. Diese Hülle wird nach der vollständigen

Abarbeitung des Ausführungsplans daher wieder zerstört [Ste96, 3.4]. Ihr Einsatz ermöglicht allerdings eine vom zugrunde liegenden Datenbanksystem ObjectStore unabhängige Funktionsweise der Ausführungsebene.

Somit liegt das einzige Argument für den umschließenden Konstruktoraufruf

OOGDMListIndexScan(OOGDMListIndex(“City”))

nun bereits als Objekt vor, sodass analog die neue Instanz der Klasse `OOGDMListIndexScan` [Mel02, 4.4] erzeugt werden kann. Wiederum wird jetzt die Klassenmethode `create` der Klasse `OOGDMFactory` mit dem vorliegenden Klassennamen `OOGDMListIndexScan` und dem zuvor erzeugten Objekt vom Typ `OOGDMListIndex` als einzigem Argument aufgerufen, um eine Instanz der Klasse `OOGDMListIndexScan` zu erhalten. Diese wird nun erneut im abschließenden Konstruktoraufruf

QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex(“City”)))

als Argument verwendet, sodass genauso wie zuvor ein neues Objekt vom Typ `QEEIndexNode` (siehe auch Seite 30) entsteht.

Nun findet die Regel

‹variableObjectPair› ::= `VARIABLE` ‹object› ‘;’

Anwendung, die das zuvor erzeugte Objekt vom Typ `QEEIndexNode` unter dem im textuellen Ausführungsplan angegebenen Bezeichner `myIndexCity1`

myIndexCity1 QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex(“City”)));

über die Klassenmethode `add` in die Klasse `OOGDMObjectPool` einfügt. Somit kann es bei der späteren Abarbeitung des erzeugten objektbasierten Ausführungsplans dort ausgelesen und anschließend verwendet werden.

Auf diese Art und Weise werden nun alle weiteren im textuellen Ausführungsplan aufgeführten Objekte erzeugt und gegebenenfalls unter ihrem Bezeichner in die Klasse `OOGDMObjectPool` eingefügt. Der schließlich erzeugte objektbasierte Ausführungsplan ist in Abbildung 3.8 zu finden. Aus Gründen der Übersichtlichkeit sind dort die im Rahmen der Konstruktoraufrufe übergebenen Bezeichner der benachbarten Kanten- und Knotenobjekte nicht dargestellt. Es wird allerdings deutlich, dass die Assoziationen zwischen unterschiedlichen Kanten- und Knotenobjekten des objektbasierten Ausführungsplans noch nicht bestehen; einzig und allein die unbenannten Objekte, die als Parameter der einzelnen Konstruktoraufrufe übergeben worden sind, sind bereits an die entsprechenden Knotenobjekte gebunden. Die fehlenden Assoziationen werden erst im Rahmen der Abarbeitung dieses objektbasierten Ausführungsplans erzeugt, indem die Bezeichner der assoziierten Objekte über die Klasse `OOGDMObjectPool` aufgelöst werden.

Abarbeitung des erzeugten objektbasierten Ausführungsplans

Nachdem der objektbasierte Ausführungsplan (siehe auch Abbildung 3.8) erzeugt worden ist, kann seine Abarbeitung beginnen. Dazu wird zuerst die Methode `startThread` des Objekts `myPrintNode` aufgerufen. Anschließend ist dieses Objekt für die weitere Abarbeitung des objektbasierten Ausführungsplans verantwortlich. Der Beginn dieses Ablaufs wird im Folgenden ausgehend von der Darstellung in Abbildung 3.9 beschrieben. Auf die Präsentation der übergebenen Parameterwerte im Rahmen der Methodenaufrufe wird zur Vereinfachung verzichtet.

Als erstes sorgt das Objekt `myPrintNode` über seine Klassenmethode `start` dafür, dass seine weitere Arbeit, also insbesondere der Aufruf seiner privaten Methode `run`, in einem neuen leichtgewichtigen Prozess abläuft. Anschließend liest es aus der Klasse `ObjectPool` über deren Methode `get` die beiden Objekte aus, die dort unter den Bezeichnern `myQueueBagResult` und `myPrintExp` gespeichert sind. An dieser Stelle löst das Objekt `myPrintNode` also seine beim Konstruktoraufruf übergebenen Bezeichner in assoziierte Objekte auf. Daraufhin aktiviert es seine einzige eingehende Warteschlange, indem es die Methode `start` des assoziierten Kantenobjekts `myQueueBagResult` verwendet. Dieses Kantenobjekt wiederum macht sich über den Aufruf von `setOutgoingQueue` beim nächsten Knotenobjekt bekannt, bevor es dessen Arbeit über `startThread` in einem neuen leichtgewichtigen Prozess startet. Der weitere Verlauf der Abarbeitung wird hier nur noch kurz angedeutet; allerdings wird deutlich, dass sich dieses Vorgehen solange fortsetzt, bis alle Kanten- und Knotenobjekte aktiviert worden sind. Das Objekt `myPrintExp` löst seine Bezeichner noch nicht auf, da es zu den nicht auf Objektströmen arbeitenden Knotenobjekten gehört.

Abbildung 3.8 Der erzeugte objektbasierte Ausführungsplan zum textuellen Pendant aus Beispiel 3.19

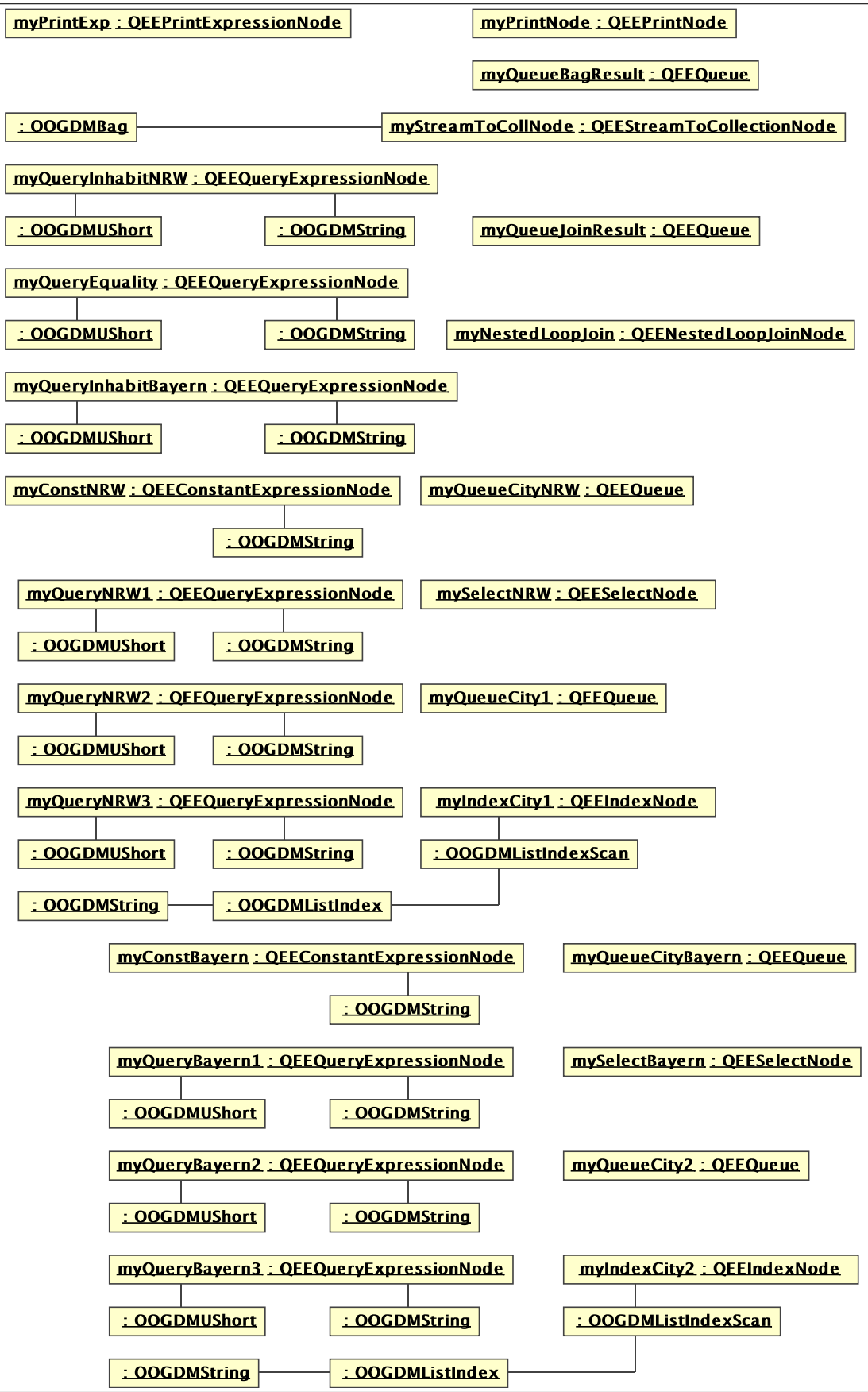


Abbildung 3.9 Beginn der Abarbeitung des objektbasierten Ausführungsplans aus Abbildung 3.8

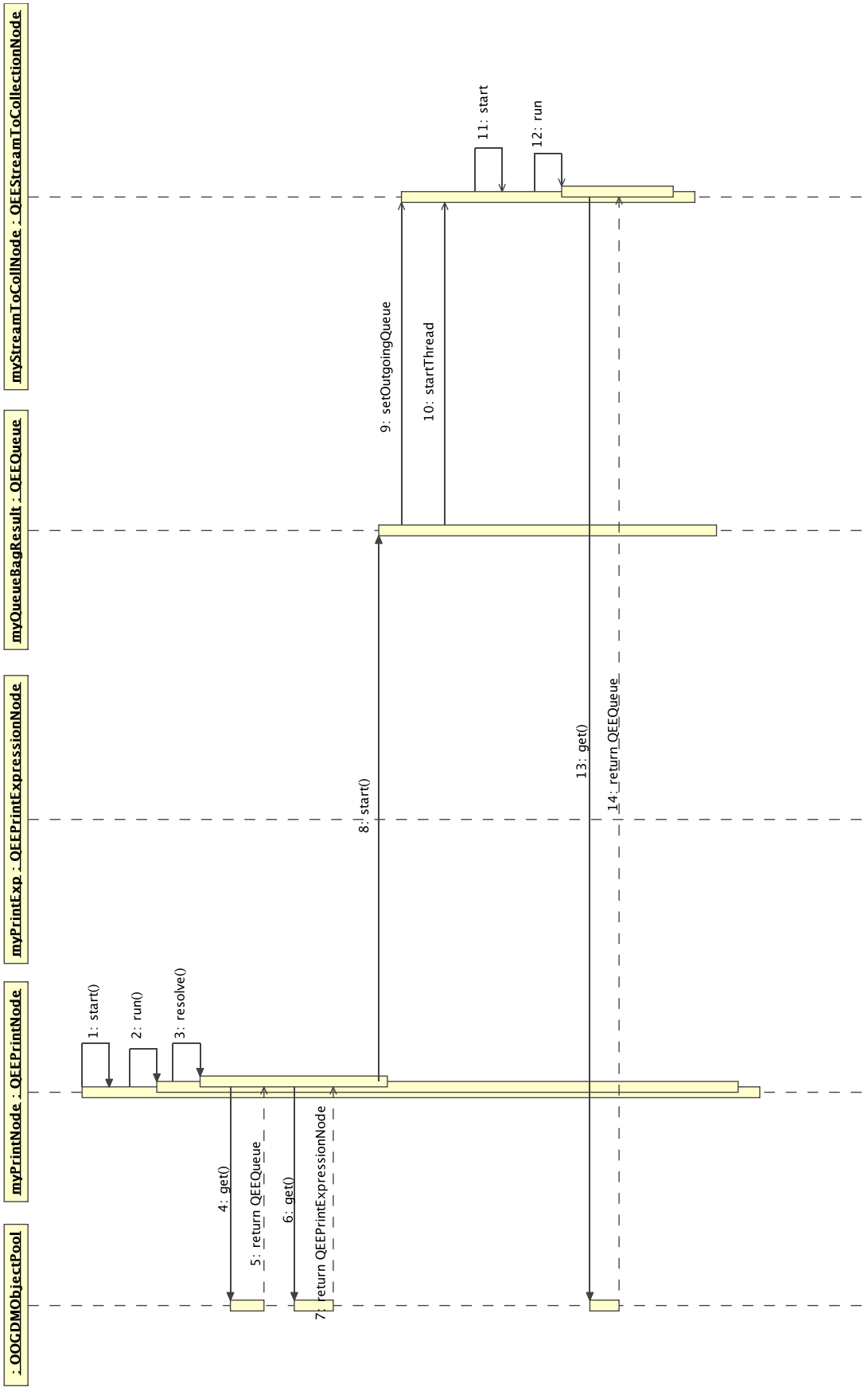


Abbildung 3.10 zeigt den objektbasierten Ausführungsplan, nachdem alle Kanten- und Knotenobjekte wie oben aktiviert worden sind. Es wird dort ebenfalls ersichtlich, dass nur diejenigen Knotenobjekte, die auf Objektströmen arbeiten – also nur Instanzen mit Oberklasse `QEEQueueNode` –, bereits ihre Bezeichner aufgelöst und entsprechende Assoziationen gebildet haben. Die verbleibenden Knotenobjekte – also alle Instanzen mit Oberklasse `QEEExpressionNode` – bilden die erforderlichen Assoziationen mit Hilfe der Klasse `OOGDMObjectPool` erst, sobald sie ein Ergebnis liefern müssen. Dieses Vorgehen stellt eine Möglichkeit dar, um die in ausführbaren Anfragegraphen vorhandenen Zyklen korrekt auf einen objektbasierten Ausführungsplan abzubilden. Schließlich bilden Knoten mit Obertyp `QEEQueueNode` oder mit Obertyp `QEEExpressionNode` für sich gesehen jeweils keine Zyklen, sodass in einem ersten Schritt alle Knoten mit Obertyp `QEEQueueNode` ihre Assoziationen bilden, bevor in einem späteren Schritt die restlichen Knoten ihre Assoziationen erzeugen. Würden im Gegensatz zu diesem gewählten Vorgehen alle Knoten im gleichen Schritt ihre Assoziationen bilden, so müsste zum Beispiel jedes Objekt eines objektbasierten Ausführungsplans selbst darauf achten, ob es bereits damit begonnen hat, seine Assoziationen zu erzeugen, damit eine Endlosrekursion vermieden wird.

In Abbildung 3.11 wird der Verlauf der Abarbeitung des vorliegenden objektbasierten Ausführungsplans am Beispiel der bereits in Abbildung 3.9 gezeigten Objekte dargestellt. Innerhalb seiner Methode `run` erhält das Objekt `myPrintNode` die zu bearbeitenden Objekte des Eingabestroms über die Methode `get` des assoziierten Kantenobjekts `myQueueBagResult`, welches die Elemente des Stroms wiederum durch das Objekt `myStreamToCollNode` über den Aufruf der Methode `put` erhält. Die Methode `get` liefert eine Instanz mit Oberklasse `QEEObject` – in diesem Fall eine Instanz der Klasse `QEEConstantObject` mit einem enthaltenen Objekt vom Typ `OOGDMBag` – zurück. Der ursprüngliche textuelle Ausführungsplan enthält an dieser Stelle eine Instanz der Klasse `OOGDMBag`, weil OOGQL-Anfragen standardmäßig ein derartiges Objekt als Rückgabe liefern [Voi97, 6.2.3.4] und weil zudem die in Beispiel 3.18 gezeigte OOGQL-Anfrage keine weitere Aussage über den Typ der Rückgabe macht.

Das Objekt `myPrintNode` delegiert nun einen wesentlichen Teil seiner Aufgaben, nämlich die Ausgabe des aktuellen Objekts seines Eingabestroms, an das assoziierte Objekt `myPrintExp` durch den Aufruf dessen Methode `evaluate`. Sollte diese dadurch das erste Mal verwendet werden, löst auch `myPrintExp` seine Bezeichner mittels `resolve` unter Zuhilfenahme der Klasse `OOGDMObjectPool` auf, bevor es mit seiner eigentlichen Aufgabe beginnt. Dazu muss es noch das aktuelle Objekt mittels `evaluate` von `myPrintNode` anfordern, bevor es dieses schließlich ausgeben kann.

Dieser Ablauf wiederholt sich solange, bis `myPrintNode` über seine Methode `terminate` zur Beendigung der Arbeit aufgefordert wird oder der eingehende Objektstrom endgültig versiegt ist – siehe dazu auch die Beschreibung der Klasse `QEEEndOfStreamObject` auf Seite 27 sowie die Darstellung der Abarbeitung objektbasierter Ausführungspläne ab Seite 39. In diesen Fällen aktiviert `myPrintNode` seine private Methode `terminateQueue`, die dafür sorgt, dass die Aufforderung zur Beendigung der Arbeit den eingehenden Kantenobjekten – in diesem Fall also `myQueueBagResult` – mitgeteilt wird. Jedes Kantenobjekt ist dann wiederum dafür verantwortlich, die Arbeit des weiteren assoziierten Knotenobjekts – in diesem Fall `myStreamToCollNode` – durch Aufruf von dessen Methode `terminate` zu beenden. In Abbildung 3.11 ruft `myPrintNode` seine private Methode `terminateQueue` auf, nachdem es ein Objekt vom Typ `QEEEndOfStreamObject` aus dem assoziierten Warteschlangenobjekt `myQueueBagResult` ausgelesen hat.

Einen wesentlichen und komplexen Teil während der Abarbeitung des hier behandelten objektbasierten Ausführungsplans stellen die auftretenden Selektionen dar. Daher soll ihr Ablauf am Beispiel der durch das Objekt `mySelectNRW` angestoßenen Selektion aller Instanzen der Klasse `City` nach dem Namen ihres Bundeslandes im Folgenden näher vorgestellt werden. Es wird vorausgesetzt, dass bereits alle erforderlichen Assoziationen gebildet worden sind.

Abbildung 3.12 zeigt die wesentlichen Aspekte der Auswertung der entsprechenden Selektionsbedingung für ein Objekt des eingehenden Stroms. Zuerst liest das Knotenobjekt `mySelectNRW` dieses aktuelle Objekt aus dem assoziierten Kantenobjekt `myQueueCity1` (1) aus. Anschließend ruft es die Methode `evaluate` (2) des ebenfalls assoziierten Objekts `myQueryNRW1` auf, ohne jedoch das aktuelle Element des eingehenden Objektstroms zu übergeben. Das Objekt `myQueryNRW1` kann allerdings seine Arbeit, den Vergleich zweier Objekte gemäß der übergebenen Methode `operator==`, erst durchführen, wenn die entsprechenden Objekte vorliegen, daher ruft es nun seinerseits `evaluate` für `myQueryNRW2` (`2.1`) und `myConstantNRW` (`2.2`) auf. Dieses Vorgehen setzt sich rekursiv fort, sodass `myQueryNRW2` sofort `evaluate` des Objekts `myQueryNRW3` (`2.1.1`) anspricht.

Dieses Objekt erhält nun das zu bearbeitende Element des vorliegenden Eingabestroms durch Aufruf der Methode `QEESelectNode::evaluate` des Objekts `mySelectNRW` (`2.1.1.1`). Für dieses Objekt, eine Instanz der Klasse `QEEConstantObject`, die das aktuelle Datenbankobjekt vom Typ `City` kapselt

Abbildung 3.10 Der objektbasierte Ausführungsplan aus Abbildung 3.8 zu Beginn seiner Abarbeitung

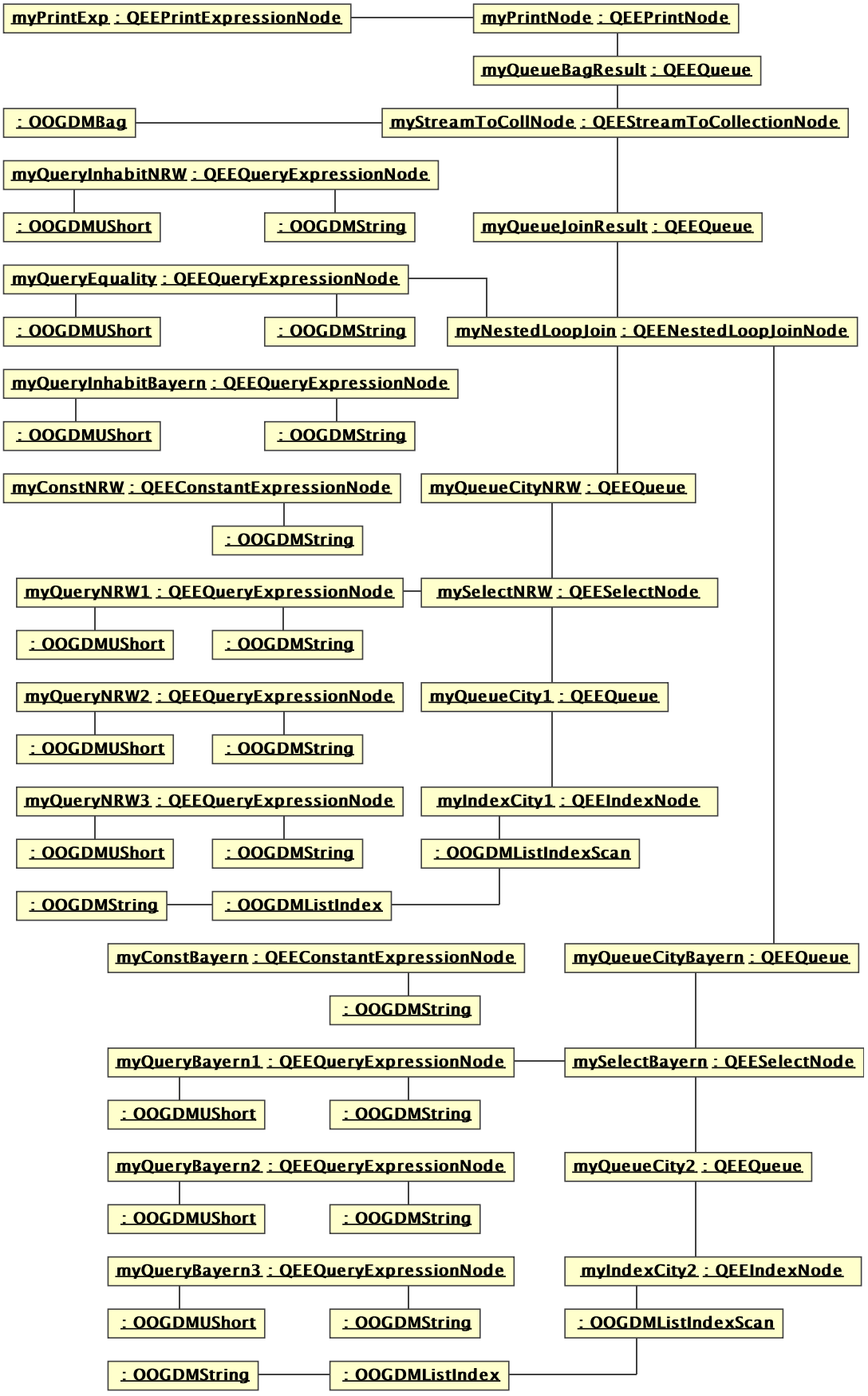


Abbildung 3.11 Verlauf der Abarbeitung des objektbasierten Ausführungsplans aus Abbildung 3.8

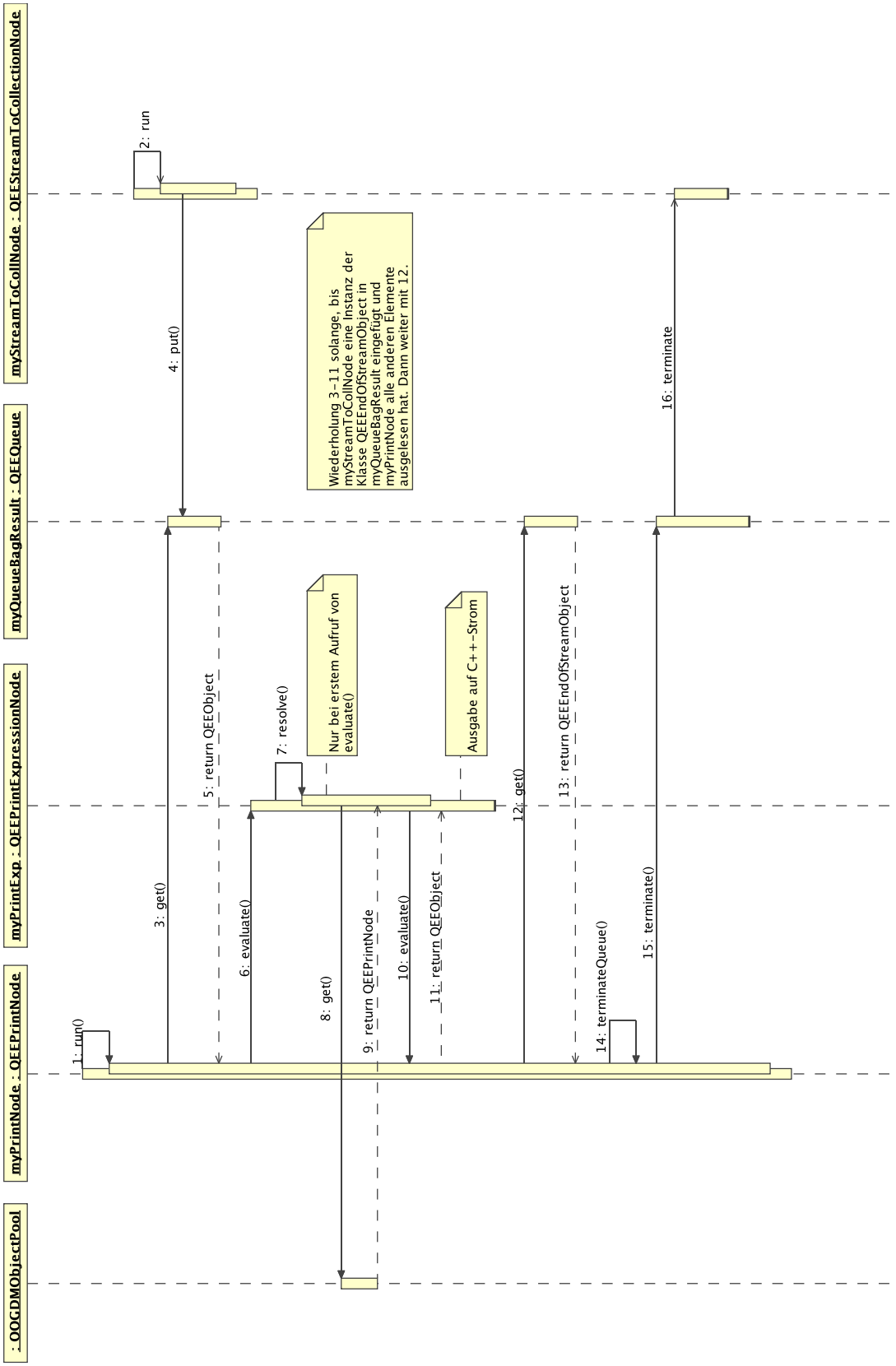
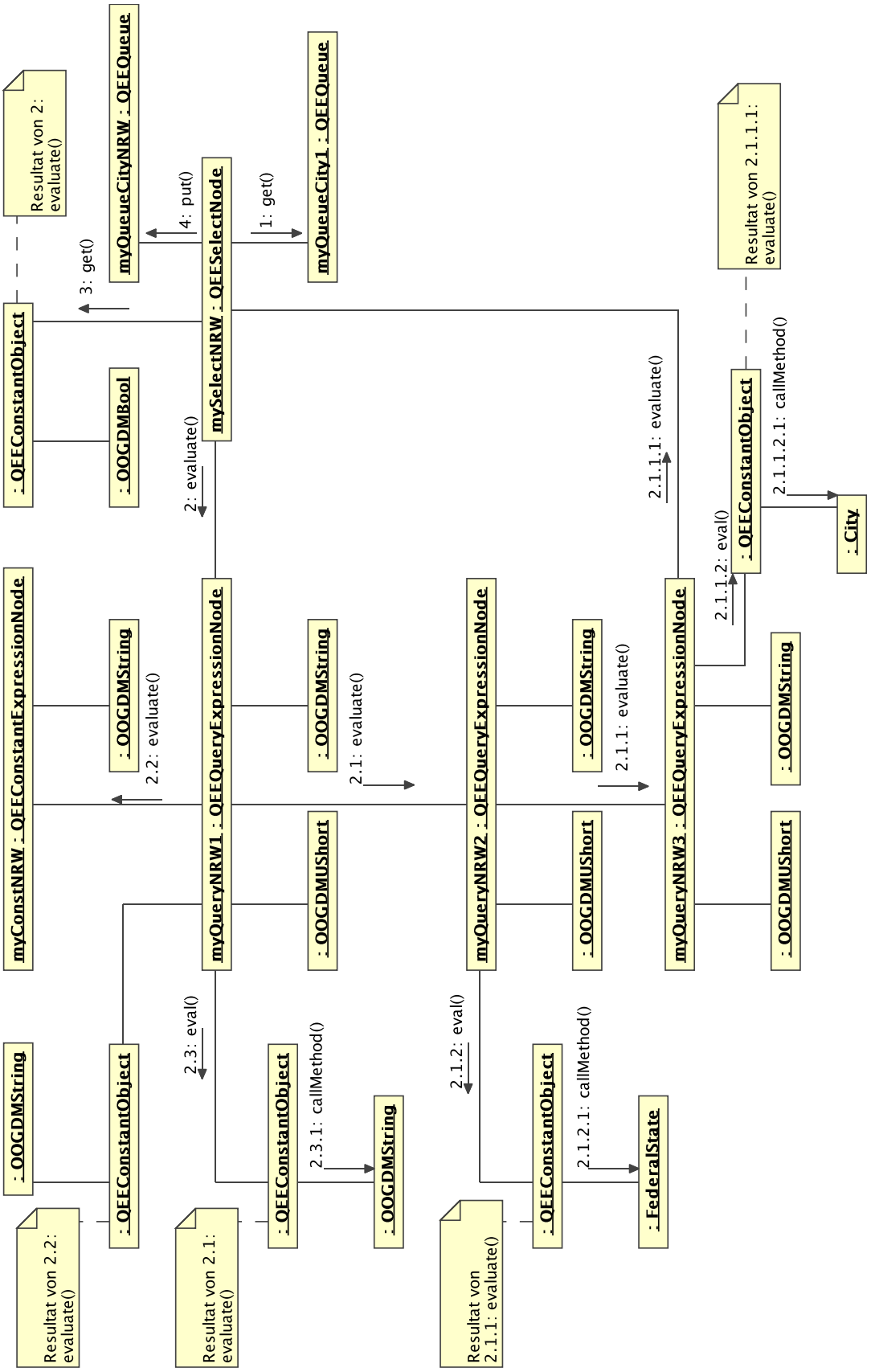


Abbildung 3.12 Berechnung der Selektion im objektbasierten Ausführungsplan aus Abbildung 3.8



(siehe auch die Beschreibung des Knotentyps QEEIndexNode auf Seite 30), wird nun die Methode eval (2.1.1.2) ausgeführt. Dadurch wird callMethod des assoziierten Objekts vom Typ City (2.1.1.2.1) aufgerufen, sodass schließlich City::getFederalState (siehe auch Beispiel 3.19) abgearbeitet wird.

Die entsprechende Rückgabe, eine Instanz der Klasse FederalState, wird nun in einem Objekt des Typs QEEConstantObject – gekapselt an das Objekt myQueryNRW2 – zurückgegeben. Dieses Objekt ruft für diese Rückgabe erneut eval (2.1.2) auf, sodass schließlich über callMethod (2.1.2.1) die Methode FederalState::getName aktiviert wird. Das Objekt myQueryNRW1 erhält damit wiederum ein Objekt des Typs QEEConstantObject, das in diesem Fall eine Instanz der Klasse OOGDMString enthält, und zwar als Repräsentation des Namens des Bundeslandes, zu dem das aktuelle Element vom Typ City des Eingabstroms gehört.

Nachdem nun myQueryNRW1 noch vom Objekt myConstNRW eine die Zeichenkette NRW repräsentierende Instanz der Klasse OOGDMString in einem Objekt vom QEEConstantObject gekapselt erhalten hat, kann es den eigentlichen Vergleich anstoßen. Dafür wird erneut die Methode eval (2.3) eingesetzt, die nun für das erste zurückbekommene Objekt vom Typ OOGDMString dessen Methode callMethod (2.3.1) aufruft, welche schließlich operator== mit der zuvor erwähnten Konstanten NRW als Argumentwert aktiviert. Das Objekt mySelectNRW erhält nun das erzeugte Ergebnis, ein Objekt vom Typ QEEConstantObject mit einer assoziierten Instanz der Klasse OOGDMBool, zurück, sodass es nach QEEConstantObject::get (3) entscheiden kann, ob das Selektionsprädikat für das vorliegende Element des eingehenden Objektstroms zu true ausgewertet worden ist. In diesem Fall fügt mySelectNRW eben dieses Objekt über put (4) in die ausgehende Warteschlange ein, welche durch das Kantenobjekt myQueueCityNRW repräsentiert wird.

Damit wurden nun die wesentlichen Abläufe bei der Verarbeitung des in Beispiel 3.19 gezeigten textuellen Ausführungsplans erläutert. Es wurde ersichtlich, dass sich durch die objektbasierte Darstellung eines Ausführungsplans leicht neue Algorithmen in Form zusätzlicher Knotentypen für ausführbare Anfragegraphen in die Ausführungsebene von GOODAC integrieren lassen. Zudem wurde mit Hilfe der ausführlichen Beschreibung der Abarbeitung eines Ausführungsplans die Funktionalität der Ausführunseebene von GOODAC verdeutlicht. Insbesondere die Bedeutung der einzelnen Parameter bei Erzeugung der Knoten eines ausführbaren Anfragegraphen sowie das Zusammenwirken dieser Knoten wurde herausgestellt. Ausgehend von diesem Beispiel und der zuvor erfolgten allgemeinen Beschreibung der Ausführungsebene lassen sich daher die Abläufe bei der Verarbeitung weiterer Ausführungspläne leicht nachvollziehen. Die unterschiedlichen Erweiterungsmöglichkeiten der Ausführungsebene und ihre Bedeutung im Rahmen der Erweiterung anderer Komponenten zur Anfragebearbeitung in GOODAC werden außerdem in Abschnitt 6.2 aufgegriffen.

Kapitel 4

Deskriptive und ausführbare Algebra – Darstellung von Anfragen und Ausführungsplänen

Im Rahmen dieser Arbeit wurden bereits die Anfragesprache OOGQL (siehe auch Abschnitt 2.2) sowie der Aufbau textueller Ausführungspläne und ihre Bedeutung innerhalb der Anfragebearbeitung (siehe auch Abschnitt 3.2.1) besprochen. Ein OOGQL-Ausdruck stellt eine vom Anwender formulierte Anfrage an GOODAC dar, während ein textueller Ausführungsplan ein konkretes Vorgehen zur Berechnung des zur Beantwortung dieser Anfrage erforderlichen Anfrageresultats repräsentiert. In diesem sowie im nachfolgenden Kapitel werden die noch ausstehenden Schritte bei der Übersetzung einer OOGQL-Anfrage in einen derartigen textuellen Ausführungsplan beschrieben. Damit die in Kapitel 5 näher vorgestellte Anfrageoptimierung in GOODAC vereinfacht wird, kommen sowohl für OOGQL-Anfragen als auch für textuelle Ausführungspläne interne algebraische Repräsentationsformen zum Einsatz. Durch deren Verwendung bestehen die in Kapitel 5 thematisierte Anfrageoptimierung und damit auch die Übersetzung einer OOGQL-Anfrage in einen textuellen Ausführungsplan im Wesentlichen in der Transformation von Ausdrücken der in diesem Kapitel beschriebenen Algebren. In GOODAC erfolgt also – ähnlich wie in anderen Datenbankmanagementsystemen [SKS02, 13 f] – eine Optimierung auf Grundlage algebraischer Repräsentationen von OOGQL-Anfragen und textuellen Ausführungsplänen.

Zur besseren Trennung zwischen der internen Darstellung von OOGQL-Anfragen und der Repräsentation textueller Ausführungspläne werden zwei unterschiedliche Algebren entwickelt, die nach einer Einführung in diesem Kapitel beschrieben werden.

Die *deskriptive Algebra* (Abschnitt 4.3) dient der internen Repräsentation von OOGQL-Anfragen. Daher müssen ihre Ausdrücke die gesamte in OOGQL durch eine deklarative Anfrage darstellbare Funktionalität ausdrücken können. Insbesondere müssen alle Konstrukte der OOGQL eine Entsprechung in der deskriptiven Algebra besitzen, damit der OOGQL-Parser (siehe auch Abschnitt 2.3) jeden gültigen OOGQL-Ausdruck in einen Ausdruck der deskriptiven Algebra überführen kann.

Die *ausführbare Algebra* gewährleistet die interne Repräsentation textueller Ausführungspläne. Daher müssen sich alle für ausführbare Anfragegraphen zugelassenen Knotentypen (siehe auch Abschnitt 3.2.3) sowie die erforderlichen Parameter zur Erzeugung konkreter Knotenobjekte durch Ausdrücke dieser ausführbaren Algebra darstellen lassen. Daneben muss während der Anfragebearbeitung eine korrekte Abbildung eines Ausdrucks dieser Algebra auf einen textuellen Ausführungsplan gewährleistet werden, sodass im Anschluss an den in Kapitel 5 dargestellten Optimierungsprozess die Komponente zur Verarbeitung von Ausführungsplänen in GOODAC (siehe aucb Abschnitt 3.2) mit ihrer Arbeit beginnen kann.

In diesem Kapitel wird also die interne Repräsentation von OOGQL-Anfragen und textuellen Ausführungsplänen innerhalb von GOODAC thematisiert. Dazu werden zuerst verschiedene Konzepte zur Repräsentation von Anfragen in relationalen Datenbanksystemen genannt (Abschnitt 4.1), bevor in Abschnitt 4.2 mit der *Second-Order Signature* näher auf ein spezielles Konzept zur Spezifikation algebraischer Repräsentationen eingegangen wird. Im Anschluss werden in Abschnitt 4.3 und in Abschnitt 4.4 die beiden oben genannten Algebren zur Repräsentation von Anfragen und Ausführungsplänen in GOODAC genauer vorgestellt. Diese beiden Repräsentationen dienen wie zuvor erwähnt als Grundlage für die in Kapitel 5 näher beschriebene Anfrageoptimierung in GOODAC.

4.1 Konzepte zur internen Repräsentation von Anfragen

Eine vom Benutzer gestellte Anfrage an ein Datenbankmanagementsystem muss in ein internes Format überführt werden, das gewissen formalen Ansprüchen genügt, damit eine weitere Bearbeitung dieser Anfrage – insbesondere ihre Optimierung – gewährleistet werden kann. Jarke und Koch [JK84] unterscheiden beispielsweise vier unterschiedliche Möglichkeiten zur internen Repräsentation von Anfragen in relationalen Datenbankmanagementsystemen: den relationalen Kalkül [SKS02, 3.6 f], die relationale Algebra [SKS02, 3.2 f], Anfragegraphen [Dat00, 17.3] und Tableaus [Heu97, 2.2.6.2]. Auf eine Darstellung dieser Repräsentationsformen wird hier verzichtet, weil die angeführten Lehrbücher bereits gute Beschreibungen liefern.

Es hat sich herausgestellt, dass die algebraische Darstellung für die interne Repräsentation von Anfragen besonders geeignet ist, weil sie durch die prozedurale Schreibweise bereits einen Algorithmus zur Berechnung des Anfrageergebnisses vorgegibt [GMUW02, 5]. Durch die Berücksichtigung von Äquivalenzregeln für die relationale Algebra sowie durch die Verwendung eines Kostenmodells lassen sich derart repräsentierte Anfragen in relationalen Datenbankmanagementsystemen gut optimieren [SKS02, 14].

4.2 Second-Order Signature

Güting [Güt93] stellt mit seiner Beschreibung der SOS (**S**econd-**O**rder **S**ignature) ein Konzept bereit, das eine einfache Beschreibung algebraischer Repräsentationen ermöglicht. Dabei werden besondere Schwerpunkte auf die leichte Erweiterbarkeit und auf die einfache maschinelle Verarbeitung der erstellten Spezifikationen gelegt, sodass sich auf der Grundlage der SOS erstellte Algebra-Spezifikationen sehr gut für die Darstellung von Anfragen und Ausführungsplänen im Rahmen einer generischen und erweiterbaren Anfragebearbeitung eignen. Neben Gütings Darstellung liefern auch Carrie [Car04, 2.1], Dieker [Die01, 3.1] und Schmidt [Sch04, 2.1] eine ausführliche Einführung in das SOS-Konzept, daher soll an dieser Stelle nur in einem für das weitere Verständnis erforderlichen Rahmen darauf eingegangen werden.

Die grundlegende Idee der SOS besteht darin, zwei miteinander in Beziehung stehende Signaturen zu verwenden, um zuerst ein Datenmodell und anschließend eine Menge auf diesem Datenmodell arbeitender Operatoren zu definieren, die als Grundlage einer Algebra dienen. Diese Algebra stellt im Wesentlichen eine Interpretation der zuvor definierten Konstrukte – beispielsweise hinsichtlich ihrer Verwendung innerhalb der Anfragebearbeitung in GOODAC – dar. Zur besseren Unterscheidung werden im Folgenden die erste Signatur *Typsignatur* und die zweite Signatur *Operatorsignatur* genannt. Eine Signatur besteht im Allgemeinen aus *Sorten* und *Operatoren*, die eine Menge von *Ausdrücken* definieren [Die01, 3.1].

Typsignatur und Operatorsignatur

Eine *Sorte der Typsignatur* wird gewöhnlich auch als eine *Kind* bezeichnet, während ihre Operatoren *Typkonstruktoren* genannt werden. Die Typsignatur lässt sich zudem wie folgt zu einer *erweiterten Typsignatur* [Güt93] ergänzen, um die einfache Definition weiterer Ausdrücke zu ermöglichen. So ist beispielsweise für Sorten $s_1, \ldots, s_n$ der Typsignatur auch deren Kreuzprodukt $(s_1 \times \ldots \times s_n)$ eine Sorte der Typsignatur, ebenso wie die Liste s^+ zu einer Sorte s der Typsignatur wieder eine Sorte der Typsignatur ist. Zudem ist für $n \geq 0$ sowie Sorten $s_1, \ldots, s_n$ und s der Typsignatur auch $(s_1 \times \ldots \times s_n \rightarrow s)$ eine neue Sorte der Typsignatur. Hier ist beispielsweise **fun**(x_1:s_1, ..., x_n:s_n) t ein Term dieser Sorte der Typsignatur, falls x_i jeweils eine Variable der Sorte s_i der Typsignatur sowie t ein Term der Sorte s der Typsignatur sind. Eine vollständige formale Einführung liefert Güting [Güt93]. Durch diese erweiterte Typsignatur lässt sich anschließend das Typsystem eines Datenbankmanagementsystems definieren. Die Operatorsignatur benutzt die erweiterte Typsignatur, um Operatoren auf den Typen zu definieren.

Im Folgenden soll dieses Konzept an einigen Beispielen erläutert werden. Dabei werden Sorten der Typsignatur in der Form SORTE, Typkonstruktoren als *typeConstructor*, Operatoren als **operator** und Bezeichner in der Form *variable* notiert.

Beispiel 4.1 zeigt eine SOS-Spezifikation für einige Rechenoperationen auf Zahlen. Als erstes wird eine Sorte NUMBER der Typsignatur definiert. Dazu gehören drei unterschiedliche Typkonstruktoren. Die Spezifikation der anschließend genannten Operatoren der Operatorsignatur bindet nun in ihrer Definition einen beliebigen dieser Typkonstruktoren an den Bezeichner *num*. Weiterhin wird deutlich, dass das Resultat der Operatoranwendung ebenfalls durch diesen Typkonstruktor beschrieben wird. Schließlich wird in einer derartigen Definition eines Operators seine Syntax vorgegeben. Dabei dient das Symbol # als Platzhalter für den Operator und __ repräsentiert einen Operanden. An diesem Beispiel wird bereits deutlich, dass es

Beispiel 4.1 SOS-Spezifikation für Rechenoperationen auf Zahlen

Es seien NUMBER eine Sorte der Typsignatur und

$$\rightarrow \quad \text{NUMBER} \quad \underline{\textit{double}}, \underline{\textit{float}}, \underline{\textit{integer}}$$

zugehörige Typkonstruktoren einer erweiterten Typsignatur.
Zudem sind beispielsweise **add**, **div** und **abs** Operatoren der entsprechenden Operatorsignatur mit folgender Definition:

$\forall$ *num* **in** NUMBER.
num $\times$ *num* $\rightarrow$ *num* **add** # (__ , __)
$\forall$ *num* **in** NUMBER.
num $\times$ *num* $\rightarrow$ *num* **div** # (__ , __)
$\forall$ *num* **in** NUMBER.
num $\rightarrow$ *num* **abs** # (__)

ausreicht, jeden Operator nur einmal zu definieren, um alle zulässigen Einsatzmöglichkeiten anzugeben. So wird etwa im vorliegenden Beispiel der Operator **div** bereits für alle gegebenen Typkonstruktoren zur Sorte NUMBER der Typsignatur definiert, wobei durch die Verwendung des Bezeichners *num* sichergestellt wird, dass keine unzulässige Vermischung der Typkonstruktoren auftreten darf.

Durch die Angabe von Sorten der Typsignatur, Typen und Operatoren kann somit eine auf der SOS basierende Algebra für den Einsatz im Rahmen der Anfragebearbeitung definiert werden, indem im Wesentlichen eine Interpretation der zuvor definierten Signaturen angegeben wird, sodass auch eine Beschreibung ihrer Semantik erfolgt [Güt93, 3.3]. Eine derartige Algebra besitzt drei entscheidende Vorteile. So ist sie aufgrund der Möglichkeit zur Angabe weiterer Spezifikationen zur Ergänzung sowohl der Typ- als auch der Operatorsignatur nicht nur bezüglich der verwendeten Operatoren, sondern auch im Hinblick auf das zugrunde liegende Datenmodell leicht erweiterbar. Daher eignet sie sich besonders für den Einsatz in erweiterbaren Datenbankmanagementsystemen. Weiterhin handelt es sich um eine prozedurale Anfragesprache [SKS02, 3.1.5], sodass durch einen Ausdruck dieser Algebra bereits eine Vorschrift zur Berechnung des Anfrageergebnisses vorgegeben wird. Zudem existieren bereits Ansätze für die interne Repräsentation von Anfragen und die Realisierung von Komponenten zur Anfrageoptimierung, die eine Definition beliebiger Datenmodelle und zugehöriger Algebren durch Einlesen entsprechender SOS-Spezifikationen zulassen [Die01, 3.2]. Bei einer Änderung des Datenmodells oder der SOS-Spezifikation müssen nur die innerhalb der Beschreibung der Optimierungsstrategie auftretenden Algebraausdrücke verändert werden. Eine Anpassung der Komponente zur Anfrageoptimierung ist im Rahmen eines derartigen Ansatzes leicht durchführbar und bezieht sich nur auf einige klar definierte Teilaspekte wie beispielsweise die Beschreibung weiterer Transformationen von Algebraausdrücken [Die01, 3.2].

4.3 Die deskriptive Algebra – Interne Repräsentation von Anfragen

Da sich die zuvor beschriebene Spezifikationstechnik gemäß SOS zur Definition einer internen Repräsentation von Anfragen – wie oben erwähnt – sehr gut für den Einsatz im Rahmen einer erweiterbaren Anfragebearbeitung eignet, werden Teile dieses Konzepts auch in GOODAC eingesetzt. Vor allem die Spezifikationstechnik zur Beschreibung der Typen und der Operatoren als Grundlage einer Algebra werden hier verwendet. So lassen sich zum Beispiel OOGQL-Ausdrücke – also von einem Benutzer gestellte Anfragen in der für GOODAC entworfenen Anfragesprache OOGQL (siehe auch Abschnitt 2.2) – in eine algebraische Darstellung überführen (siehe auch Abschnitt 2.3). Diese algebraische Darstellung erleichtert durch eine prozedurale Beschreibung des zur Bestimmung des Anfrageergebnisses erforderlichen Vorgehens die spätere Anfrageoptimierung.

Im Folgenden wird die zugrunde liegende Algebra – die *deskriptive Algebra* – vorgestellt. Dazu wird zuerst das Typsystem dieser deskriptiven Algebra erläutert (Abschnitt 4.3.1), bevor Abschnitt 4.3.2 eine Auswahl der auftretenden Operatoren näher beschreibt. In diesem Zusammenhang wird zudem auf die Semantik der Typen und Operatoren eingegangen, um ein leichteres Verständnis zu ermöglichen. Eine vollständige Darstellung dieser deskriptiven Algebra liefert bereits Carrie [Car04, 2]. Damit eine einfache Umformung und Optimierung deskriptiver Algebraausdrücke möglich wird, existiert zudem eine objektbasierte Darstellung, die in Abschnitt 4.3.3 kurz beschrieben wird.

4.3.1 Typsystem der deskriptiven Algebra

Als Basis des Typsystems der deskriptiven Algebra dienen in der Typsignatur die Sorten CLASS (zur Repräsentation von Klassen), IDENT (für Bezeichner), OBJECTTUPLE (Tupel bestehend aus Bezeichnern und Referenzen auf Objekte), OBJECT (Objekte), REF (Referenzen auf Objekte), STREAM (Ströme von Objekttupeln) und TUPLE (zur Darstellung von Attributname-Objektreferenz-Paaren). Diese Sorten der Typsignatur werden wie oben beschrieben in der Definition der Typkonstruktoren verwendet. In GOODAC existieren dazu die im Folgenden beschriebenen Typkonstruktoren zur Definition der Typen der deskriptiven Algebra. Ihre konkrete Einbettung in die Anfragebearbeitung in GOODAC wird anschließend ebenfalls beschrieben.

Bezeichner: Der konstante Typkonstruktor *ident* erzeugt einen Typ der Sorte IDENT der Typsignatur zur Beschreibung von Bezeichnern.

$\rightarrow$ IDENT *ident*

Referenzen auf Objekte: Durch Angabe des Namens einer Klasse liefert der Typkonstruktor *oid* eine Referenz auf ein Objekt dieser Klasse zurück.

ident $\rightarrow$ REF *oid*

Tupel zur Verwendung als Attributname-Objektreferenz-Paare: Ein Tupel, das Paare von Attributnamen und Objektreferenzen in GOODAC darstellt, wird durch den Typkonstruktor *tuple* erzeugt. Das Prädikat *noDuplicateNames* stellt sicher, dass die als Argument übergebene Liste keine Duplikate enthält.

$\forall$ *list* **in** (*ident* $\times$ REF)*,
noDuplicateNames(*list*).
list $\rightarrow$ TUPLE *tuple*

Objekte: Objekte werden durch Typen beschrieben, die der Typkonstruktor *object* erzeugt. Sie bestehen aus einer Referenz auf sich selbst, einem Tupel von Attributname-Objektreferenz-Paaren für die Darstellung der Attribute zur Beschreibung des aktuellen Objektzustands sowie einer Liste von – jeweils mit einem Bezeichner versehenen – Methoden. Diese Methoden erzeugen ausgehend von einer Argumentliste – eine Liste von Referenzen auf Objekte – und dem Objektzustand einen neuen Objektzustand sowie eine Referenz auf ein weiteres Objekt als Rückgabe. Aus Gründen der Übersichtlichkeit erfolgt im Rahmen dieser Darstellung keine Unterscheidung zwischen statischen und nicht statischen Methoden.

$\forall$ *tuple* **in** TUPLE,
$\forall$ *objRef* **in** REF,
$\forall$ r_i **in** REF,
$\forall$ $args_i$ **in** REF*,
$\forall$ $name_i$ **in** *ident*.
objRef $\times$ *tuple* $\times$ ($name_i$ $\times$ ($args_i$ $\times$ *tuple* $\rightarrow$ *tuple* $\times$ r_i))$^+$ $\rightarrow$ OBJECT *object*

Klassen ohne Oberklasse: Derartige Klassen werden durch den Typkonstruktor *class* erzeugt. Sie werden durch den Namen der Oberklasse und einen entsprechenden Objekttyp dargestellt. Für Klassen ohne Oberklasse wird der aktuelle Klassenname auch als Name der Oberklasse angegeben, um diese Definition mit der nachfolgenden einheitlich zu halten. Klassen ohne Oberklasse werden also in dieser Beschreibung von sich selbst abgeleitet. In dieser Definition kommen erstmals anonyme Variablen zum Einsatz. Diese sind paarweise verschieden und im weiteren Verlauf der Spezifikation nicht mehr referenzierbar. Sie dienen einzig und allein der Vereinfachung der Spezifikation.

$\forall$ *obj*: *object*(*oid*(*name*), __, __) **in** OBJECT.
name $\times$ *obj* $\rightarrow$ CLASS *class*

Klassen mit Oberklasse: Diese Klassen werden ebenfalls durch den Typkonstruktor *class* erzeugt. Im Rahmen dieser Spezifikation kommt mit der Angabe **in extension** (CLASS) ein bisher nicht erwähntes Konzept zum Einsatz. Dieses ermöglicht den Zugriff auf alle bisher spezifierten Klassen, sodass durch diese Verwendung des Konstrukts **in extension** (CLASS) sichergestellt wird, dass eine entsprechende Oberklasse existiert und daher das Ableiten einer weiteren Klassen möglich ist. Weiterhin wird hierdurch gewährleistet, dass die in der zugehörigen Objektdefinition angegebenen

Attributname-Objektreferenz-Paare und Methoden für die angegebene Klasse definiert sind. Daneben sorgt das zuvor nicht verwendete Prädikat *subtype* dafür, dass die genannten Attributname-Objektreferenz-Paare und Methoden den Gegebenheiten in einer Vererbungshierarchie entsprechen. Auch in dieser Spezifikation werden erneut anonyme Variablen verwendet, um die Beschreibung dieses Typkonstruktors zu vereinfachen.
$\forall$ __: *class*(__, *object*(*oid*(*name*), $tuple_1$, $methods_1$)) **in extension** (CLASS),
$\forall$ *obj*: *object*(__, $tuple_2$, $methods_2$) **in** OBJECT,
subtype($tuple_2$, $methods_2$, $tuple_1$, $methods_1$).
$name \times obj \rightarrow$ CLASS *class*

Damit ist das Typsystem der deskriptiven Algebra vollständig beschrieben. Um seine Verwendung jedoch auch im Rahmen der Anfragebearbeitung in GOODAC darstellen zu können, soll nun noch das zugehörige Modell dieser Anfragebearbeitung erläutert werden.

Paare bestehend aus Bezeichnern und Referenzen auf Objekte: Derartige *Objekttupel* dienen nicht der Beschreibung des GOODAC zugrunde liegenden Datenmodells OOGDM, sondern kommen wie zuvor erwähnt im Rahmen der Beschreibung von Anfragen zum Einsatz. Ein einfach aufgebautes Objekttupel besteht aus einem Paar, das einen Bezeichner und eine Referenz auf ein Objekt enthält. Seine Beschreibung erfolgt durch die Verwendung des Typkonstruktors *objectTuple* und die Sorte OBJECTTUPLE der Typsignatur.
$\forall$ __: *class*(__, *object*(*ref*, __, __)) **in extension** (CLASS),
$\forall$ *name* **in** *ident*.
$\langle(name \times ref)\rangle \rightarrow$ OBJECTTUPLE *objectTuple*
Im Allgemeinen können Objekttupel jedoch eine beliebige Anzahl von Komponenten – die jeweils ein derartiges Paar darstellen – besitzen. In diesem Fall müssen die Bezeichner disjunkt sein; dieses wird durch die Verwendung des Prädikats *noDuplicateNames* sichergestellt. Das Konstrukt $(name_i)^+$ beschreibt, dass eine Liste $\langle name_1, \ldots, name_n \rangle$ von Bezeichnern gebildet wird, deren einzelne Elemente den durch die vorherige Spezifikation gebundenen Bezeichnern entsprechen. Über diese Bezeichner ist bei Bedarf ein Zugriff auf die einzelnen Komponenten eines Objekttupels möglich.
$\forall$ __: *objectTuple*($\langle(name_i, ref_i)\rangle$) **in** OBJECTTUPLE,
noDuplicateNames($(name_i)^+$).
$(name_i \times ref_i)^+ \rightarrow$ OBJECTTUPLE *objectTuple*

Ströme von Objekttupeln: Bereits in Abschnitt 3.1 wurde deutlich, dass die einzelnen Operationen während der Abarbeitung eines Ausführungsplans auf Strömen von Objekten arbeiten. Daher erwarten auch die meisten Operatoren der hier beschriebenen deskriptiven Algebra bereits Objektströme als Operanden. Der Typ dieser Ströme – die folglich eine Folge von Objekttupeln darstellen – wird über den Typkonstruktor *stream* beschrieben.
$\forall$ *o* **in** OBJECTTUPLE.
o $\rightarrow$ STREAM *stream*

Beispiel 4.2 veranschaulicht die Bedeutung dieser Typen. Dort wird eine vereinfachte Version der in Abschnitt 3.2.6 zum Einsatz kommenden Klasse `FederalState` gemäß OOGDM-ODL definiert. Diese Klassendefinition besteht im Wesentlichen aus der Angabe des Klassennamens und der Oberklasse sowie einem Attribut und einer Methode zum Auslesen des entsprechenden Werts. Der zugehörige Typ beschreibt diese Klasse durch die definierten Typkonstruktoren. Hier ist der Typkonstruktor *class* von Interesse. Er erwartet als erstes den Namen der Oberklasse und anschließend eine durch *object* erzeugte Objektbeschreibung. Diese enthält wiederum den Typ des Attributs sowie der ebenfalls angegebenen Methode.

4.3.2 Operatoren der deskriptiven Algebra

OOGQL-Anfragen werden – wie bereits oben erwähnt – innerhalb von GOODAC durch Ausdrücke der deskriptiven Algebra dargestellt. Zur Formulierung dieser Ausdrücke existiert eine Vielzahl von Operatoren [Car04, 2], um die gesamte Ausdruckskraft der OOGQL auch durch die deskriptive Algebra darstellen zu können. Der in Abschnitt 2.3 beschriebene OOGQL-Parser sorgt dafür, dass ein OOGQL-Ausdruck in einen Ausdruck der deskriptiven Algebra übersetzt wird. An dieser Stelle sollen jedoch nur die für das weitere Verständnis dieser Ausarbeitung erforderlichen Operatoren aufgeführt werden. Diese werden dazu im Folgenden nicht alphabetisch, sondern gemäß ihres Auftretens in deskriptiven Algebraausdrücken genannt. Eine umfassende Übersicht der Operatoren der deskriptiven Algebra gibt Carrie [Car04, 2].

Beispiel 4.2 Deskriptiver Algebratyp einer in OOGDM-ODL definierten Klasse

Es sei folgende vereinfachte Klassendefinition in OOGDM-ODL (siehe auch Abschnitt 2.4) gegeben.

```
class FederalState : GeoObject {
    attribute OOGDMString name;
    OOGDMString getName() const;
};
```

Dann lautet der zugehörige Typ der deskriptiven Algebra

class(“GeoObject”,
object(*oid*(“FederalState”),
tuple(⟨(“name”, *oid*(“OOGDMString”))⟩),
⟨(“getName”,
(⟨⟩, *tuple*(⟨(“name”, *oid*(“OOGDMString”))⟩) →
tuple(⟨(“name”, *oid*(“OOGDMString”))⟩), *oid*(“OOGDMString”))⟩))

Beispiel 4.3 Deskriptiver Algebratyp eines einfachen deskriptiven Algebraausdrucks

Der folgende Algebraausdruck

City(“ci”)

besitzt den Typ

stream(*objectTuple*(⟨(“ci”, *oid*(“City”))⟩))

Beispiel 4.4 Verwendung der Operatoren **className** und **streamToBag**

Der OOGQL-Ausdruck

select * **from** City ci;

entspricht dem deskriptiven Algebraausdruck

streamToBag(**City**(“ci”))

className: Da die meisten Operatoren der deskriptiven Algebra auf Strömen von Objekten – also Ausdrücken vom Typ *stream*(*o*) – operieren, müssen die in der Datenbank gespeicherten Objekte in diese Form überführt werden. Dazu dient der Operator **className**, der für jede Klasse mit in einer Datenbank gespeicherten Objekten – sichergestellt durch **in extension** (CLASS) in der nachfolgenden Spezifikation – in der Definition der ausführbaren Algebra existiert. Hier wird somit ein Bezeichner als Operatorname verwendet. Damit liefert der nach dem Klassennamen benannte Operator alle Instanzen dieser Klasse zurück. Bei diesem Klassennamen wird im Rahmen der textuellen Notation deskriptiver Algebraausdrücke auf die Verwendung der sonst üblichen Anführungszeichen verzichtet, weil er als Operator zum Einsatz kommt. Zusätzlich ist ein Bezeichner als Parameter bei der Verwendung dieses Operators erforderlich, um einen eindeutigen Namen für die Komponente des den Objektstrom beschreibenden Objekttupels zu erzeugen. Beispiel 4.3 zeigt eine Verwendung dieses Operators in der Form **City** mit Angabe des Bezeichners “ci”. Der ebenfalls dort dargestellte zugehörige Algebratyp verwendet diesen Bezeichner zur eindeutigen Kennzeichnung der entsprechenden Komponente des Objekttupels. In Beispiel 4.4 findet sich dieser Algebraausdruck als Teil eines anderen Ausdrucks wieder, der eine OOGQL-Anfrage intern darstellt. Hier wird vor allem deutlich, dass dieser Operator **className** der deskriptiven Algebra erforderlich ist, um Zugriffe auf Datenbankobjekte im from-Teil einer OOGQL-Anfrage in der internen Darstellung abbilden zu können.
∀ __: *class*(__, *object*(*oid*(*className*), __, __)) **in extension** (CLASS),
∀ *name* **in** *ident*.
name → *stream*(*objectTuple*(⟨(*name*, *oid*(*className*))⟩)) **className** # (__)

streamToBag: Laut Voigtmann [Voi97, 6.2.3.4] liefern OOGQL-Anfragen als Ergebnis ein Objekt vom Typ `OOGDMBag` zurück. Weil die deskriptive Algebra jedoch weitgehend auf Objektströmen operiert, sorgt der Operator **streamToBag** dafür, dass die Elemente eines derartigen Objektstroms in Form

einer Instanz der Klasse `OOGDMBag` bereitgestellt werden. Eine exemplarische Verwendung dieses Operators im Rahmen der internen Darstellung einer einfachen OOGQL-Anfrage zeigt Beispiel 4.4.
∀ *s*: *stream*(*objectTuple*(*list*)) **in** STREAM,
∀ *b*: *oid*("OOGDMBag") **in** REF.
s → *b* **streamToBag** # (__)

streamToSet: Manche Anfragen liefern eine duplikatfreie Ergebnismenge. In diesem Fall bietet es sich daher an, das Ergebnis in Form eines Objekts vom Typ `OOGDMSet` zurückzugeben. Hierzu existiert der analog zu **streamToBag** definierte Operator **streamToSet** der deskriptiven Algebra.
∀ *s*: *stream*(*objectTuple*(*list*)) **in** STREAM,
∀ *b*: *oid*("OOGDMSet") **in** REF.
s → *b* **streamToSet** # (__)

product: Im Rahmen von OOGQL-Anfragen werden häufig Objektströme miteinander verknüpft. Die einfachste derartige Verknüpfung besteht in der Bildung des Kreuzprodukts zweier Objektströme. Hierzu dient der Operator **product**. Er erwartet die beiden zu verknüpfenden Objektströme als Eingabe und liefert einen Strom als Ausgabe, dessen gekapseltes Objekttupel gerade aus der Konkatenation der Objekttupel der Eingabeströme besteht. Dieses wird durch die Verwendung des Prädikats *flatten* gewährleistet. Beispiel 4.5 zeigt die Verwendung dieses Operators bei der internen Darstellung einer OOGQL-Anfrage, die im Wesentlichen das Kreuzprodukt zweier Objektströme bildet.
∀ s_1: *stream*(*objectTuple*($list_1$)) **in** STREAM,
∀ s_2: *stream*(*objectTuple*($list_2$)) **in** STREAM,
∀ *s*: *stream*(*objectTuple*(*list*)) **in** STREAM,
flatten(⟨$list_1$, $list_2$⟩, *list*).
$s_1 \times s_2$ → *s* **product** __ __ #

name: Der Zugriff auf einzelne Komponenten eines Objekttupels erfolgt über den Operator **name**, der durch die Verwendung eines Bezeichners als Operatorname für alle Bezeichner in den Komponenten eines Objekttupels spezifiziert wird. Diese Bezeichner sind nach Konstruktion, also durch die Definition des Typkonstruktors *objectTuple*, eindeutig. Durch die Verwendung des Prädikats *member* wird sichergestellt, dass das durch (*name*, *ref*) gegebene Paar in der dem Objekttupel zugrunde liegenden Liste *list* enthalten ist. Als Ergebnis liefert **name** die zu diesem Bezeichner in einer Komponente des Objekttupels gespeicherte Objektreferenz. Weiterhin kommt dieser Operator **name** bei der textuellen Notation von Algebraausdrücken ebenso wie **className** ohne die sonst üblichen Anführungszeichen zum Einsatz. In Beispiel 4.6 liefert die Anwendung des Operators als **ci**(*x*) eine Referenz auf ein Objekt vom Typ `City`, weil diese im an den Bezeichner *x* gebundenen Objekttupel als zweites Element des Paares enthalten ist, auf das über den Bezeichner `ci` zugegriffen werden kann.
∀ *o*: *objectTuple*(*list*) **in** OBJECTTUPLE,
∀ *name* **in** *ident*,
∀ *ref* **in** REF,
member((*name*, *ref*), *list*).
o → *ref* **name** # (__)

applyobjmethod: Die interne Darstellung von Aufrufen nicht statischer Methoden erfolgt durch Verwendung des Operators **applyobjmethod**. Durch die genaue Angabe der Struktur des Objekts *o* sowie die Verwendung des Prädikats *member* wird sichergestellt, dass nur Methoden genannt werden können, die das entsprechende Objekt besitzt. Ebenso werden die Argumente der Methode, ihr Rückgabetyp sowie die Attributmenge des Objekts in der Spezifikation verwendet, um nur zugelassene Methodenaufrufe zu erlauben. Lediglich eine Unterscheidung zwischen statischen und nicht statischen Methoden findet hier zur Vereinfachung der Darstellung nicht statt. Die letzte Zeile in Beispiel 4.6 zeigt etwa die Darstellung des Methodenaufrufs `ci.getName()` durch einen Ausdruck der deskriptiven Algebra unter Verwendung des Operators **applyobjmethod**.
∀ *o*: *object*(*objectRef*, *tuple*, *methodList*) **in** OBJECT,
∀ *name* **in** *ident*,
∀ *r* **in** REF,
∀ *args* **in** REF^*,
∀ *method* **in** $args \times tuple \rightarrow tuple \times r$,
member(*method*, *methodList*).
$name \times args \times objectRef$ → *r* **applyobjmethod** # (__, __, __)

Beispiel 4.5 Verwendung des Operators **product**

Der OOGQL-Ausdruck

select * **from** City ci, Castle ca;

entspricht dem deskriptiven Algebraausdruck

streamToBag(**City**(“ci”) **Castle**(“ca”) **product**)

Beispiel 4.6 Verwendung der Operatoren **name**, **applyobjmethod**, **applyclassmethod** und **select**

Der OOGQL-Ausdruck

select * **from** City ci **where** (ci.getName() **==** OOGDMString(“Muenster”));

entspricht dem deskriptiven Algebraausdruck

streamToBag(**City**(“ci”) **select** [**fun**(*x*: *objectTuple*(⟨(“ci”, *oid*(City))⟩))
applyobjmethod(
“==”,
⟨**applyclassmethod**(“new”, ⟨**refStringOp**(“Muenster”)⟩, “OOGDMString)⟩,
applyobjmethod(“getName”, ⟨⟩, **ci**(*x*)))])

applyclassmethod: Aufrufe statischer Methoden werden in der deskriptiven Algebra durch den Operator **applyclassmethod** abgebildet. Er ist ähnlich wie **applyobjmethod** definiert, allerdings erfolgt der Methodenaufruf für eine Klasse, sodass anstelle einer Objektreferenz ein Klassenname *cname* erwartet wird. Jede Klasse besitzt zudem die spezielle statische Methode `new`, die zur Darstellung von Konstruktoraufrufen dient. So zeigt Beispiel 4.6 einen Konstruktoraufruf für ein Objekt des Typs `OOGDMString`. Das übergebene Argument `''Muenster''` wird von einem Operator **refStringOp** gekapselt, der dafür sorgt, dass diese Konstante in eine passende Objektreferenz umgewandelt wird. Nach der Definition dieses Operators ist eine Einbettung in einen weiteren durch **applyclassmethod** dargestellten Konstruktoraufruf nicht erforderlich; allerdings führt die Angabe des Konstruktoraufrufs in der OOGQL-Anfrage zu einer entsprechenden Abbildung auf den Operator **applyclassmethod** im gezeigten deskriptiven Algebraausdruck.
$\forall$ *o*: *object*(*oid*(*cname*), *tuple*, *methodList*) **in** OBJECT,
$\forall$ *name* **in** *ident*,
$\forall$ *r* **in** REF,
$\forall$ *args* **in** REF*,
$\forall$ *method* **in** *args* $\times$ *tuple* $\rightarrow$ *tuple* $\times$ *r*,
member(*method*, *methodList*).
name $\times$ *args* $\times$ *cname* $\rightarrow$ *r* **applyclassmethod** # (__, __, __)

select: Selektionen – also die Auswertung eines Prädikats für alle Elemente eines Objektstroms – können durch den Operator **select** in der deskriptiven Algebra realisiert werden. Dieser Operator erwartet neben einem Objektstrom eine Funktion, die auf dem Objekttupel des Stroms arbeitet und das aktuelle Element eines Stroms zu einer Referenz auf eine Instanz der Klasse `OOGDMBool` auswertet. In Abhängigkeit von diesem Ergebnis wird das aktuelle Element des Eingabestroms verworfen oder in den Ausgabestrom eingefügt. Zugriffe auf einzelne Komponenten eines Objekttupels sowie Methodenaufrufe innerhalb dieses Prädikats werden durch die zuvor vorgestellten Konstrukte der deskriptiven Algebra dargestellt. In Beispiel 4.6 wird zu einer OOGQL-Anfrage, die ein Prädikat für alle Objekte vom Typ `City` auswertet, eine entsprechende interne Darstellung durch einen deskriptiven Algebraausdruck unter Einsatz des Operators **select** vorgestellt.
$\forall$ *s*: *stream*(*o*) **in** STREAM.
s $\times$ (*o* $\rightarrow$ *oid*(“OOGDMBool”)) $\rightarrow$ *s* **select** __ # [__]

join: OOGQL-Anfragen, die zu einem deskriptiven Algebraausdruck führen, der sowohl ein kartesisches Produkt als auch eine Selektion mit einem auf dem Ergebnisstrom des Produkts arbeitenden Prädikat enthält, lassen sich intern ebenso durch eine Verbundoperation unter Verwendung des Operators **join** darstellen. Die Spezifikation dieses Operators entspricht daher im Wesentlichen der Vereinigung

Beispiel 4.7 Verwendung des Operators **join**

Der OOGQL-Ausdruck

select * **from** City c1, City c2 **where** (c1.getName() **==** c2.getName());

entspricht dem deskriptiven Algebraausdruck

streamToBag(**City**(“c1”)
City(“c2”)
join [**fun**(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))⟩))
applyobjmethod(“==”,
⟨**applyobjmethod**(“getName”, ⟨⟩, **c2**(*x*))⟩,
applyobjmethod(“getName”, ⟨⟩, **c1**(*x*)))])

Beispiel 4.8 Verwendung des Operators **project**

Der OOGQL-Ausdruck

select ci.getName() **from** City ci;

entspricht dem deskriptiven Algebraausdruck

streamToBag(**City**(“ci”) **project**[⟨(“ci_getName”, **fun**(*x*: *objectTuple*(⟨(“ci”, *oid*(“City”))⟩))
applyobjmethod(“getName”, ⟨⟩, **ci**(*x*)))⟩])

der Spezifikationen für **product** und **select**. Beispiel 4.7 zeigt die Verwendung dieses Operators zur internen Darstellung einer OOGQL-Anfrage, die alle Paare von Städten gleichen Namens bestimmt. Dieses Beispiel zeigt zudem nochmals anschaulich die Verwendung von Bezeichnern, um auf die einzelnen Komponenten eines Objekttupels zugreifen zu können.
∀ s_1: *stream*(*objectTuple*($list_1$)) **in** STREAM,
∀ s_2: *stream*(*objectTuple*($list_2$)) **in** STREAM,
∀ *s*: *stream*(*objectTuple*(*list*)) **in** STREAM,
flatten(⟨$list_1$, $list_2$⟩, *list*).
$s_1 \times s_2 \times$ (*objectTuple*(*list*) → *oid*(“OOGDMBool”)) → *s* **join** __ __ # [__]

project: Einfache Projektionen werden in der deskriptiven Algebra durch den Operator **project** dargestellt. Es existieren in der deskriptiven Algebra weitere Operatoren zur Darstellung von Projektionen – etwa zur Duplikatentfernung im Anfrageresultat oder zur Verwendung nach Anwendung einer Gruppierungs-Operation und der damit verbundenen Typ-Änderung des Eingabestroms. Diese werden jedoch aus Platzgründen an dieser Stelle nicht aufgeführt. Der Operator **project** erwartet neben dem Eingabestrom eine Liste bestehend aus Bezeichnern und Funktionen, die jeweils ein Element dieses eingehenden Objektstroms auf eine Objektreferenz abbilden. Ein derartiger Bezeichner stellt den neuen Attributnamen dar, unter dem eine bestimmte Referenz – also ein Teilergebnis der Projektion und damit ein Attribut eines Objekts des Ausgabestroms – im ausgehenden Strom verfügbar ist. Durch die Verwendung von Funktionen als Argumente sind an dieser Stelle auch beliebig geschachtelte Methodenaufrufe und damit insbesondere erweiterte Projektionen erlaubt. Beispiel 4.8 zeigt die interne Darstellung einer exemplarischen OOGQL-Anfrage, die eine Projektion enthält.
∀ s_1: *stream*(*o*) **in** STREAM,
∀ $name_i$ **in** *ident*,
∀ ref_i **in** REF,
∀ s_2: *stream*(*objectTuple*(($name_i$, ref_i)$^+$)) **in** STREAM,
noDuplicateNames(($name_i$)$^+$).
$s_1 \times (name_i \times (o \rightarrow ref_i))^+$ → s_2 **project** __ # [__]

Die hier gezeigten Operatoren reichen bereits aus, um einen Großteil der in OOGQL möglichen Anfragen durch Ausdrücke der deskriptiven Algebra zu formulieren. Insbesondere werden viele geometrische und temporale Operatoren und Prädikate aus der OOGQL auf Methodenaufrufe der beteiligten Objekte abgebildet, die sich durch eine Verwendung des Operators **applyobjmethod** in Ausdrücken der deskriptiven Algebra darstellen lassen. Carrie [Car04, 2] liefert – wie schon oben erwähnt – die Definition weiterer Operatoren der deskriptiven Algebra.

Beispiel 4.9 OOGQL-Anfrage aus Beispiel 3.18 als deskriptiver Algebraausdruck

Die in Beispiel 3.18 gezeigte OOGQL-Anfrage lässt sich durch folgenden Ausdruck der deskriptiven Algebra darstellen:

streamToBag(
City(“c1”)
City(“c2”)
product
select[
fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))>))
applyobjmethod(
“and”,
<**applyobjmethod**(
“and”,
<**applyobjmethod**(
“==”,
<**applyobjmethod**(“getInhabitants”, <>, **c2**(*x*))>,
applyobjmethod(“getInhabitants”, <>, **c1**(*x*)))>,
applyobjmethod(
“==”,
<**refStringOp**(“Bayern”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c2**(*x*)))))>,
applyobjmethod(
“==”,
<**refStringOp**(“NRW”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c1**(*x*))))))])

Ein etwas größeres Beispiel für einen deskriptiven Algebraausdruck ist in Beispiel 4.9 zu sehen. Dieser Ausdruck dient der internen Darstellung der in Beispiel 3.18 gezeigten OOGQL-Anfrage. Die Komplexität des Ausdrucks ergibt sich vor allem aus den geschachtelten Methodenaufrufen, die hauptsächlich aus den Pfadausdrücken und der booleschen Verknüpfung der zugehörigen Resultate entstehen.

4.3.3 Objektbasierte Darstellung deskriptiver Algebraausdrücke

Um deskriptive Algebraausdrücke leicht verarbeiten zu können, werden sie vor einer genaueren Betrachtung oder Umformung in eine objektbasierte Darstellung überführt. Deren Struktur ähnelt im Wesentlichen dem Aufbau eines ausführbaren Anfragegraphen (siehe auch Kapitel 3), sodass die spätere Erzeugung eines textuellen Ausführungsplans (siehe auch Abschnitt 4.4.3) im Verlauf der Anfragebearbeitung ebenfalls vereinfacht wird. Zusätzlich enthält die objektbasierte Darstellung eines deskriptiven Algebraausdrucks Informationen über seinen Typ, sodass sich beispielsweise leicht der Typ des Anfrageergebnisses bestimmen lässt. Diese objektbasierte Darstellung eines deskriptiven Algebraausdrucks dient später als Grundlage für den Optimierungsprozess in GOODAC (siehe auch Kapitel 5).

Auf eine genaue Darstellung dieses objektbasierten Formats und der zugrunde liegenden Klassenhierarchie wird an dieser Stelle verzichtet, da bereits Carrie sowohl eine ausführliche diesbezügliche Erläuterung [Car04, 3] als auch eine Darstellung der Überführung der textuellen Beschreibung eines Algebraausdrucks in seine objektbasierte Repräsentation [Car04, 5] liefert. Auerdem beschreibt auch Schmidt [Sch04, 3.1] diese Darstellung. Daneben zeigt Carrie, dass sich zu jedem gültigen deskriptiven Algebraausdruck leicht der zugehörige Typ bestimmen lässt [Car04, 4]. Carrie erläutert in diesem Zusammenhang, wie sich für einen gültigen Algebraausdruck sein Typ berechnen lässt. Somit ist es grundsätzlich möglich, Algebraausdrücke auch unter Berücksichtigung ihres Typs zu optimieren.

4.4 Die ausführbare Algebra – Interne Repräsentation von Ausführungsplänen

Die ausführbare Algebra dient als Grundlage für Ausdrücke zur internen Repräsentation textueller Ausführungspläne. Sie ist eng an die deskriptive Algebra angelehnt, um den Optimierungsprozess – in dem insbesondere ein Ausdruck der deskriptiven Algebra in einen Ausdruck der ausführbaren Algebra überführt wird – möglichst einfach zu gestalten. Aus diesem Grund sind alle Typen und fast alle Operatoren der deskriptiven Algebra ebenfalls Bestandteile der ausführbaren Algebra.

Weil die ausführbare Algebra der algebraischen Repräsentation textueller Ausführungspläne (siehe auch Abschnitt 3.2.1) dient, muss sie im Gegensatz zur deskriptiven Algebra insbesondere für jeden in einem ausführbaren Anfragegraphen zugelassenen Knotentyp (siehe auch Abschnitt 3.2.3) einen entsprechenden Operator innerhalb der algebraische Repräsentation definieren. Außerdem werden Ströme von in einer Datenbank gespeicherten Objekten in textuellen Ausführungsplänen nicht durch die Angabe eines Klassennamens, sondern durch die Verwendung einer Indexstruktur erzeugt. Zudem formuliert ein Benutzer eine OOGQL-Anfrage nicht prozedural und geht somit weder auf die verfügbaren Indexstrukturen zum Auslesen aller Instanzen einer Klasse aus der Datenbank noch auf die anzuwendenen Operationen zur Erzeugung des Anfrageergebnisses ein. Ein textueller Ausführungsplan hingegen enthält diese Informationen.

Da sich die Anforderungen an die interne Darstellung von OOGQL-Anfragen und textuellen Ausführungsplänen somit unterscheiden, wird zur besseren Trennung dieser beiden Konzepte eine neue zweite Algebra definiert. Im Rahmen des Optimierungsprozesses muss also ein Ausdruck der deskriptiven Algebra auf einen Ausdruck der ausführbaren Algebra abgebildet werden. Daher stellt die ausführbare Algebra weiterhin alle erforderlichen Operatoren bereit, um diese Abbildung durchführen zu können, sodass nur noch entsprechende Knotentypen in die ausführbare Ebene eingebettet werden müssen, um die durch deskriptive Algebraausdrücke – und damit auch durch OOGQL-Anfragen – darstellbare Funktionalität der Anfragebearbeitung in GOODAC einsetzen zu können.

Eine genaue Beschreibung aller Operatoren der ausführbaren Algebra findet sich bei Schmidt [Sch04, 2], sodass im Rahmen dieser Arbeit nur auf ihre wesentlichen Aspekte eingegangen wird. Eine Übersicht der zusätzlichen Typen liefert hierbei Abschnitt 4.4.1, bevor in Abschnitt 4.4.2 eine Beschreibung wesentlicher Operatoren der ausführbaren Algebra erfolgt. Ausgehend von dieser Darstellung beschreibt Abschnitt 4.4.3 die Erzeugung textueller Ausführungspläne.

4.4.1 Typsystem der ausführbaren Algebra

Alle Typen der deskriptiven Algebra werden in das Typsystem der ausführbaren Algebra übernommen. In der ausführbaren Algebra werden jedoch Indexzugriffe erforderlich, da alle Objekte einer Klasse in GOODAC jeweils nur über einen Index erreichbar sind [Voi97, 7.3.4]. Aus diesem Grund existiert im Typsystem der ausführbaren Algebra eine weitere Sorte INDEX der Typsignatur mit mehreren Typkonstruktoren. Diese Typkonstruktoren entsprechen den in GOODAC verfügbaren und geplanten Indexstrukturen.

Listen: Die Klasse `OOGDMListIndex` [Mel02, 4.4] stellt Instanzen bereit, die eine listenbasierte Indexstruktur für den Zugriff auf alle Objekte einer Klasse repräsentieren (siehe auch Abschnitt 3.2.6). Ein derartiger Index eignet sich nur als Primärindex für die Objekte einer Klasse, da auf ihm keine Suche möglich ist [SKS02, 12.2.1]. Der zugehörige Typkonstruktor *listIndex* erwartet eine Objektreferenz zur eindeutigen Bestimmung der Klasse der indizierten Instanzen, sodass anschließend der Typ einer listenbasierten Indexstruktur für alle Instanzen der zugehörigen Klasse erzeugt wird.
∀ __: *class*(__, *object*(*ref*, __, __)) **in extension** (CLASS).
ref → INDEX *listIndex*

B-Bäume: Ein B-Baum [CLR97, 19] wird in seiner Variante als B^+-Baum in Datenbankmanagementsystemen häufig als Primär- oder Sekundärindex zur Indizierung von Daten, die eindimensionale Schlüssel besitzen, verwendet [SKS02, 12.3]. Der zugehörige Typkonstruktor *btree* erwartet ebenso wie *listIndex* eine Objektreferenz, um die Klasse der indizierten Instanzen zu ermitteln. Weiterhin benötigt er eine Funktion *f*, die eine Objektreferenz dieser Klasse auf eine Instanz mit Oberklasse `OOGDMComparable` abbildet, um den zugehörigen Schlüsselwert zu ermitteln. Schließlich ist eine Referenz *b* auf ein Objekt vom Typ `OOGDMBool` erforderlich, um zu kennzeichnen, ob es sich um einen Primärindex handelt. Die Verwendung des Prädikats *subtype* sowie von **in extension** (CLASS) wurden bereits oben näher erläutert.

∀ *o*: *object*(*ref*, __, __) **in** OBJECT,
∀ __: *class*(__, *o*) **in extension** (CLASS),
∀ o_1: *object*(ref_1, $tuple_1$, $methodList_1$) **in** OBJECT,
∀ *f* **in** *ref* → ref_1,
∀ *b*: *oid*(“OOGDMBool”) **in** REF,
∀ o_2: *object*(*oid*(“OOGDMComparable”), $tuple_2$, $methodList_2$) **in** OBJECT,
∀ __: *class*(__, o_1) **in extension** (CLASS),
∀ __: *class*(__, o_2) **in extension** (CLASS),
subtype($tuple_1$, $methodList_1$, $tuple_2$, $methodList_2$).
ref × *f* × *b* → INDEX *btree*

X-Bäume: Der X-Baum [BKK96] stellt eine Indexstruktur dar, die sich besonders für die effiziente Verwaltung mehrdimensionaler Daten in Form eines Primär- oder Sekundärindex eignet. Eine genauere Beschreibung des X-Baums und eine Übersicht weiterer Datenstrukturen für räumliche und andere mehrdimensionale Daten liefern beispielsweise Breimann und Vahrenhold [BV03]. Die Definition des entsprechenden Typkonstruktors *xtree* ähnelt stark der Definition des Typkonstruktors *btree*; allerdings müssen die Schlüsselwerte beim X-Baum die Eigenschaften so genannter MBRs (**M**inumum **B**ounding **R**ectangles) [BV03] erfüllen. Dieses wird dadurch sichergestellt, dass die als Schlüsselwerte dienenden Objekte die Oberklasse `OOGDMMBR` besitzen müssen.
∀ *o*: *object*(*ref*, __, __) **in** OBJECT,
∀ __: *class*(__, *o*) **in extension** (CLASS),
∀ o_1: *object*(ref_1, $tuple_1$, $methodList_1$) **in** OBJECT,
∀ *f* **in** *ref* → ref_1,
∀ *b*: *oid*(“OOGDMBool”) **in** REF,
∀ o_2: *object*(*oid*(“OOGDMMBR”), $tuple_2$, $methodList_2$) **in** OBJECT,
∀ __: *class*(__, o_1) **in extension** (CLASS),
∀ __: *class*(__, o_2) **in extension** (CLASS),
subtype($tuple_1$, $methodList_1$, $tuple_2$, $methodList_2$).
ref × *f* × *b* → INDEX *xtree*

Beliebige Indexstrukturen: Oft – etwa bei der Definition einiger Operatoren der ausführbaren Algebra – ist der genaue Typ des betrachteten Index nicht von Interesse; stattdessen wird nur Wert darauf gelegt, dass ein Index vorliegt. Daher wird der Typkonstruktor *index* definiert, der eine Objektreferenz zur Bestimmung der Klasse der indizierten Objekte sowie eine Referenz *b* auf eine Instanz der Klasse `OOGDMBool` zur Unterscheidung zwischen Primär- und Sekundärindizes erwartet.
∀ *o*: *object*(*ref*, __, __) **in** OBJECT,
∀ __: *class*(__, *o*) **in extension** (CLASS),
∀ *b*: *oid*(“OOGDMBool”) **in** REF.
ref × *b* → INDEX *index*

Die von den drei speziellen Typkonstruktoren erzeugten Typen der Sorte INDEX der Typsignatur lassen sich wie nachfolgend gezeigt auch formal als Untertypen zu den vom Typkonstruktor *index* erzeugten Obertypen einordnen. Hier wird insbesondere die SOS dahingehend erweitert, dass Ausdrücke, die Operatoren der Operatorsignatur enthalten, zur Definition weiterer Typen der Typsignatur eingesetzt werden.

listIndex(*ref*) < *index*(*ref*, **refBoolOp**(true))
btree(*ref*, __, *b*) < *index*(*ref*, *b*)
xtree(*ref*, __, *b*) < *index*(*ref*, *b*)

4.4.2 Operatoren der ausführbaren Algebra

Basierend auf dem zuvor dargestellten Typsystem existieren neben den aus der deskriptiven Algebra übernommenen Operatoren in der ausführbaren Algebra zusätzliche Operatoren, um die volle Ausdruckskraft textueller Ausführungspläne durch Ausdrücke der ausführbaren Algebra darstellen zu können. An dieser Stelle werden nur einige wesentliche Operatoren vorgestellt, weil sich eine vollständige Übersicht – wie bereits oben erwähnt – schon in der Arbeit von Schmidt [Sch04, 2] findet. In den zugehörigen Beispielen werden zum besseren Verständnis auch die von den Ausdrücken der ausführbaren Algebra repräsentierten textuellen Ausführungspläne gezeigt, deren Aufbau bereits in Abschnitt 3.2 besprochen wurde.

Beispiel 4.10 Verwendung der Operatoren **scan** und **print**

Der textuelle Ausführungsplan

```
myIndexCity     QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myQueueCity     QEEQueue(myIndexCity);
myPrintNode     QEEPrintNode(myQueueCity, myPrintExp);
myPrintExp      QEEPrintExpressionNode(myPrintNode);
```

entspricht dem ausführbaren Algebraausdruck

print
listIndex(*oid*("City"))
"c"
scan [**fun**(*x*: *objectTuple*(‹("c", *oid*("City"))›)) **refBoolOp**(true)]
"cout"

scan: Der Zugriff auf alle in einer Datenbank gespeicherten Instanzen einer Klasse sowie die Konstruktion eines Objektstroms, der diese Instanzen enthält, erfolgt in der ausführbaren Algebra durch das Auslesen einer zugehörigen Indexstruktur mit Hilfe des Operators **scan**. Dieser erwartet nebem dem Index *index* einen Bezeichner *name* für den zu erzeugenden Objektstrom sowie eine Funktion, mit deren Hilfe für jedes im Index gespeicherte Objekt ein Prädikat ausgewertet werden kann, um bereits beim Datenbankzugriff Objekte, die nicht im Anfrageergebnis enthalten sein können, zu verwerfen. Dadurch kann erreicht werden, dass der erzeugte Objektstrom nur eine Teilmenge der in der Datenbank enthaltenen Instanzen einer Klasse enthält. In Beispiel 4.10 wird eine Anwendung dieses Operators gezeigt, die einen Zugriff auf alle in einer als Indexstruktur dienenden Liste gespeicherten Instanzen der Klasse `City` repräsentiert. Das verwendete Prädikat sorgt in diesem Beispiel dafür, dass alle in der Datenbank enthaltenen Objekte des Typs `City` in den Ausgabestrom eingefügt werden.
∀ *index*: *index*(*oid*(*className*), __) **in** INDEX,
∀ *c*: *class*(__, *object*(*oid*(*className*), __, __)) **in** CLASS,
∀ *name* **in** *ident*,
∀ *s*: *stream*(*objectTuple*(‹(*name*, *oid*(*className*))›)) **in** STREAM.
index × *name* × (*oid*(*className*) → *oid*("OOGDMBool")) → *s* **scan** __ __ # [__]

streamToCollection: In der deskriptiven Algebra existieren mehrere Operatoren, um einen Objektstrom in eine Kollektion von Objekten umzuwandeln. Um diese Anzahl zu reduzieren, wird in der ausführbaren Algebra der Operator **streamToCollection** verwendet, der neben einem Strom noch eine Referenz auf ein Kollektionsobjekt erwartet, in das die Elemente des Stroms eingefügt werden sollen. Daher stellt die Referenz auf dieses Kollektionsobjekt das Ergebnis der Operatoranwendung dar. In Beispiel 4.11 kommt der Operator **streamToCollection** zum Einsatz, um alle im Ergebnis – also im ausgehenden Objektstrom – der dort gezeigten Projektion enthaltenen Elemente in eine Kollektion vom Typ `OOGDMBag` einzufügen.
∀ *s*: *stream*(*objectTuple(list)*) **in** STREAM,
∀ *o*: *object*(*ref*, *tuple*$_1$, *methodList*$_1$) **in** OBJECT,
∀ *c*: *object*(*oid*("OOGDMCollection'), *tuple*$_2$, *methodList*$_2$) **in** OBJECT,
∀ __: *class*(__, *o*) **in extension** (CLASS),
∀ __: *class*(__, *c*) **in extension** (CLASS),
subtype(*tuple*$_1$, *methodList*$_1$, *tuple*$_2$, *methodList*$_2$).
s × *ref* → *ref* **streamToCollection** # (__, __)

projection: Der Operator **projection** der ausführbaren Algebra dient ebenfalls der Reduktion möglicher Operatoren in ausführbaren Algebraausdrücken. Seine Definition folgt im Wesentlichen der Beschreibung des deskriptiven Algebraoperators **project**, jedoch besitzt **projection** zwei zusätzliche Operanden, nämlich Referenzen auf Objekte vom Typ `OOGDMBool`. Diese beiden Operanden geben an, ob in der deskriptiven Algebra anstelle des Operators **project** die Operatoren **projectdistinct** (für garantiert duplikatfreie Ergebnismengen), **projectsnapshot** (zur Gewährleistung von Ergebnismengen ohne historische Informationen) oder **projectdistinctsnapshot** (für eine Kombination beider Eigenschaften) [Car04, 2] verwendet worden sind. In Beispiel 4.11 kommt der Operator **projection** zum Einsatz, um über den Aufruf der Methode `getName` jedes Objekt vom Typ `City` auf seinen Namen zu projizieren, der nachfolgend unter dem Attributnamen `name` verfügbar ist. Die Werte der letzten

Beispiel 4.11 Verwendung der Operatoren **streamToCollection** und **projection**

Der textuelle Ausführungsplan

```
myIndexCity     QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myQueueCity     QEEQueue(myIndexCity);
myProjNode      QEEProjectionNode(myQueueCity,
                                  OOGDMList("name"),
                                  OOGDMList(myQuery1));
myQuery1        QEEQueryExpressionNode(myProjNode,
                                       OOGDMUShort("0"),
                                       "getName");
myQueueProj     QEEQueue(myProjNode);
myCollNode      QEEStreamToCollectionNode(myQueueProj,OOGDMBag());
```

entspricht dem ausführbaren Algebraausdruck

```
streamToCollection(
    listIndex(oid("City"))
    "c"
        scan [fun(x: objectTuple(<("c", oid("City"))>)) refBoolOp(true)]
            projection
                [<("name",
                   fun(x: objectTuple(<("c", oid("City"))>))
                   applyobjmethod("getName", <>, c(x)))>]
                refBoolOp(false)
                refBoolOp(false),
    applyclassmethod("new", <>, "OOGDMBag"))
```

beiden Operanden geben an, dass es sich um eine gewöhnliche Projektion handelt, bei der weder die Ergebnismenge duplikatfrei noch ohne historische Information sein muss.
$\forall\ s_1$: *stream*(o) **in** STREAM,
$\forall\ name_i$ **in** *ident*,
$\forall\ ref_i$ **in** REF,
$\forall\ s_2$: *stream*(*objectTuple*($(name_i, ref_i)^+$)) **in** STREAM,
$\forall$ *isDistinct*: *oid*("OOGDMBool") **in** REF,
$\forall$ *isSnapshot*: *oid*("OOGDMBool") **in** REF,
noDuplicateNames($(name_i)^+$).
$s_1 \times (name_i \times (o \rightarrow ref_i))^+ \times isDistinct \times isSnapshot \quad \rightarrow \quad s_2$ **projection** __ # [__] __ __

print: Die Verwendung von **print** repräsentiert die Ausgabe aller Elemente eines Objektstroms bei der Abarbeitung eines Ausführungsplans. Daher erwartet dieser Operator einen Objektstrom *s*, der die auszugebenden Elemente enthält, sowie den Bezeichner eines C++-Ausgabestroms *name* als Operanden. In Beispiel 4.10 sorgt die Verwendung von **print** dafür, dass alle über den Operator **scan** aus der Datenbank ausgelesenen Objekte zur Ausgabe auf den C++-Ausgabestrom `cout` geleitet werden.
$\forall\ s$: *stream*(o) **in** STREAM,
$\forall$ *name* **in** *ident*.
$s \times name \quad \rightarrow$ **print** # __ __

sort: Der Operator **sort** ermöglicht in ausführbaren Algebraausdrücken die Repräsentation von Sortierknoten für ausführbare Anfragegraphen. Neben einem eingehenden Objektstrom erwartet er eine Liste von Paaren. Diese Paare bestehen aus einer Funktion, die jedes Element des eingehenden Objektstroms auf einen Sortierschlüsselwert abbildet, und einer Referenz auf ein Objekt vom Typ `OOGDMBool`, die angibt, ob die Elemente des Objektstroms gemäß der somit ermittelten Schlüsselwerte aufsteigend oder absteigend sortiert werden sollen. Daneben erfordert der durch den Knotentyp `QEESortNode` für ausführbare Anfragegraphen – siehe auch Seite 35 in Abschnitt 3.2.3 – dargestellte Sortieralgorithmus [Bre00, 4.2.1] die Angabe eines weiteren Parameters, sodass **sort** zusätzlich eine Referenz auf ein Objekt vom Typ `OOGDMInt` erwartet. In Beispiel 4.14 wird ein Algebraausdruck gezeigt, in dem der Operator **sort** verwendet wird, um einen aus Objekten vom Typ `City` bestehenden

Strom nach absteigender Einwohnerzahl zu sortieren.
$\forall$ *s*: *stream*(*o*) **in** STREAM,
$\forall$ ref_i **in** REF,
$\forall$ obj_i: *object*(ref_i, $tuple_i$, $methodList_i$) **in** OBJECT,
$\forall$ *obj*: *object*(*oid*("OOGDMComparable"), *tuple*, *methodList*) **in** OBJECT,
$\forall$ __: *class*(__, obj_i) **in extension** (CLASS),
$\forall$ __: *class*(__, *obj*) **in extension** (CLASS),
$\forall$ asc_i: *oid*("OOGDMBool") **in** REF,
subtype(*tuple*, *methodList*, *tuple*, *methodList*).
$s \times ((o \rightarrow ref_i) \times asc_i)^+ \times$ *oid*("OOGDMInt") $\rightarrow$ *s* **sort** __ # [__] __

first: Dieser Operator der ausführbaren Algebra erzeugt einen Objektstrom, der nur das erste Element des eingehenden Stroms enthält. Er repräsentiert somit den in Abschnitt 3.2.3 auf Seite 34 beschriebenen Knotentyp `QEEFirstElemNode` für ausführbare Anfragegraphen. In Kombination mit dem Operator **sort** kann dieser Operator beispielsweise eingesetzt werden, um Maxima oder Minima bezüglich einer totalen Ordnung für die Elemente eines Objektstroms zu bestimmen. Beispiel 4.12 zeigt einen Algebraausdruck, in dem **first** dazu verwendet wird, das erste Element des Resultats einer Verbundoperation zu bestimmen.
$\forall$ *s*: *stream*(*o*) **in** STREAM.
s $\rightarrow$ *s* **first** __ #

nestedLoopJoin: Der Operator **nestedLoopJoin** dient der Darstellung der Verwendung des entsprechenden für textuelle Ausführungspläne zugelassenen Algorithmus zur Berechnung der Verbundoperation – siehe auch die Beschreibung der Klasse `QEENestedLoopJoinNode` in auf Seite 33 – in Ausdrücken der ausführbaren Algebra. Seine Definition folgt der Beschreibung des Operators **join**, sodass an dieser Stelle nicht genauer darauf eingegangen wird. In Beispiel 4.12 kommt dieser Operator zum Einsatz, um alle Paare von Objekten des Typs `City` zu bestimmen, deren Namen übereinstimmen.
$\forall$ s_1: *stream*(*objectTuple*($list_1$)) **in** STREAM,
$\forall$ s_2: *stream*(*objectTuple*($list_2$)) **in** STREAM,
$\forall$ *s*: *stream*(*objectTuple*(*list*)) **in** STREAM,
flatten($\langle list_1, list_2 \rangle$, *list*).
$s_1 \times s_2 \times$ (*objectTuple*(*list*) $\rightarrow$ *oid*("OOGDMBool")) $\rightarrow$ *s* **nestedLoopJoin** __ __ # [__]

mergeJoin: Der Operator **mergeJoin** ermöglicht die Repräsentation des Einsatzes einer weiteren für textuelle Ausführungspläne zugelassenen Verbundoperation [Bre00, 4.2.2] in Ausdrücken der ausführbaren Algebra. Gegenüber dem sehr ähnlichen Operator **nestedLoopJoin** kommt im Rahmen dieser Definition ein weiterer Parameter – an dritter Stelle – zum Einsatz. Dieser dient dazu, die vom entsprechenden Knotentyp `QEEMergeJoinNode` – siehe auch Seite 32 in Abschnitt 3.2.3 – geforderte totale Ordnung auf dem im Prädikat der Verbundoperation zu berücksichtigenden Wertebereich durch diesen Operator **mergeJoin** in der ausführbaren Algebra darstellen zu können. Eine genauere Beschreibung der Bedeutung dieses Parameters findet sich beispielsweise bei der Darstellung des Knotentyps `QEEMergeJoinNode` in Abschnitt 3.2.3 auf Seite 32 oder in den ursprünglichen Erläuterungen von Breimann [Bre00, 4.2.2]. In Beispiel 4.14 kommt dieser Operator zum Tragen, um die Verbundoperation zwischen zwei Strömen mit Objekten vom Typ `City` hinsichtlich gleicher Einwohnerzahl zu berechnen. Damit der Operator **mergeJoin** verwendet werden kann, wird zuvor sichergestellt, dass die eingehenden Objektströme passend geordnet sind.
$\forall$ s_1: *stream*(*objectTuple*($list_1$)) **in** STREAM,
$\forall$ s_2: *stream*(*objectTuple*($list_2$)) **in** STREAM,
$\forall$ *s*: *stream*(*objectTuple*(*list*)) **in** STREAM,
flatten($\langle list_1, list_2 \rangle$, *list*).
$s_1 \times s_2 \times$ (*objectTuple*(*list*) $\rightarrow$ *oid*("OOGDMBool"))
$\times$ (*objectTuple*(*list*) $\rightarrow$ *oid*("OOGDMBool")) $\rightarrow$ *s* **mergeJoin** __ __ __ # [__]

Die Operatoren der ausführbaren Algebra besitzen ebenso wie die Operatoren der deskriptiven Algebra eine objektbasierte Darstellung. Schmidt [Sch04, 3] beschreibt ausführlich, auf welche Art und Weise die dazu entwickelten Klassen in die bei Carrie [Car04, 3 f] beschriebene Klassenstruktur für eine objektbasierte Repräsentation deskriptiver Algebraausdrücke eingebettet werden können. Weiterhin stellt Schmidt notwendige Erweiterungen vor, um auch ausführbare Algebraausdrücke aus ihrer textuellen Darstellung in die zugehörige objektbasierte Repräsentation überführen zu können.

Beispiel 4.12 Verwendung der Operatoren **nestedLoopJoin** und **first**

Der textuelle Ausführungsplan

```
myIndexCity1    QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myIndexCity2    QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myQueue1        QEEQueue(myIndexCity1);
myQueue2        QEEQueue(myIndexCity2);
myJoinNode      QEENestedLoopJoinNode(myQueue1, myQueue2, myQuery1);
myQuery1        QEEQueryExpressionNode(myQuery2,
                                       OOGDMUShort("0"),
                                       "operator==",
                                       myQuery3);
myQuery2        QEEQueryExpressionNode(myJoinNode,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
myQuery3        QEEQueryExpressionNode(myJoinNode,
                                       OOGDMUShort("1"),
                                       "getInhabitants");
myQueue3        QEEQueue(myJoinNode);
myFirstNode     QEEFirstElemNode(myQueue3);
```

entspricht dem ausführbaren Algebraausdruck

```
listIndex(oid("City"))
"c1"
    scan [fun(x: objectTuple(<("c1", oid("City"))>)) refBoolOp(true)]
listIndex(oid("City"))
"c2"
    scan [fun(x: objectTuple(<("c2", oid("City"))>)) refBoolOp(true)]
        nestedLoopJoin[
            fun(x: objectTuple(<("c1", oid("City")), ("c2", oid("City"))>))
            applyobjmethod(
                "==",
                <applyobjmethod("getName", <>, c2(x))>,
                applyobjmethod("getName", <>, c1(x)))]
            first
```

4.4.3 Erzeugung textueller Ausführungspläne

Zum Abschluss des Optimierungsprozesses erzeugt die eingesetzte Komponente zur generischen Anfrageoptimierung (siehe auch Abschnitt 5.2 und Abschnitt 5.3) einen ausführbaren Algebraausdruck, der den abzuarbeitenden textuellen Ausführungsplan repräsentiert. Damit dieser Ausführungsplan jedoch von der Komponente zur Auswertung von Ausführungsplänen bearbeitet werden kann, muss er zuvor aus dem ausführbaren Algebraausdruck erzeugt werden.

Schmidt [Sch04, 6] erläutert bereits umfassend die Erzeugung textueller Ausführungspläne, daher soll hier nur kurz darauf eingegangen werden. Der Grundgedanke besteht darin, dass jedes Objekt in der objektbasierten Darstellung eines ausführbaren Algebraausdrucks, das einen Knoten des ausführbaren Anfragegraphen repräsentiert, seine eigene Ausgabe vornimmt. Dazu erzeugt jedes derartige Objekt in einem ersten Durchlauf einen eindeutigen Bezeichner, unter dem es im zu erzeugenden textuellen Ausführungsplan auffindbar ist. In einem weiteren Durchgang erstellt dann jedes Objekt seinen eigenen Eintrag für den zu erzeugenden textuellen Ausführungsplan, wobei die zu einem Operator gehörenden Operanden durch ihre im ersten Schritt festgelegten eindeutigen Bezeichner dargestellt werden. Sofern ein Operand einen eingehenden Objektstrom liefert, wird im textuellen Ausführungsplan zusätzlich eine textuelle Beschreibung eines zugehörigen Warteschlangenobjekts (siehe auch Abschnitt 3.2.2) erzeugt, das diesen Objektstrom repräsentiert.

Ein so erzeugter textueller Ausführungsplan kann anschließend von der Komponente zur Auswertung von Ausführungsplänen bearbeitet und ausgeführt werden, um das Ergebnis der ursprünglich vorliegenden Anfrage auszugeben.

Beispiel 4.13 Textueller Ausführungsplan zu Beispiel 4.14

Der textuelle Ausführungsplan

```
myIndexCity1    QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myIndexCity2    QEEIndexNode(OOGDMListIndexScan(OOGDMListIndex("City")));
myQueue1        QEEQueue(myIndexCity1);
myQueue2        QEEQueue(myIndexCity2);
mySortNode1     QEESortNode(myQueue1, myQuery1, OOGDMInt("64"));
myQuery1        QEEQueryExpressionNode(myQuery2,
                                       OOGDMUShort("0"),
                                       ">",
                                       myQuery3);
myQuery2        QEEQueryExpressionNode(mySortNode1,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
myQuery3        QEEQueryExpressionNode(mySortNode1,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
mySortNode2     QEESortNode(myQueue2, myQuery4, OOGDMInt("64"));
myQuery4        QEEQueryExpressionNode(myQuery5,
                                       OOGDMUShort("0"),
                                       ">",
                                       myQuery6);
myQuery5        QEEQueryExpressionNode(mySortNode2,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
myQuery6        QEEQueryExpressionNode(mySortNode2,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
myQueue3        QEEQueue(mySortNode1);
myQueue4        QEEQueue(mySortNode2);
myJoinNode      QEEMergeJoinNode(myQueue2,
                                 myQueue3,
                                 myQuery7,
                                 myQuery10);
myQuery7        QEEQueryExpressionNode(myQuery8,
                                       OOGDMUShort("0"),
                                       "operator==",
                                       myQuery9);
myQuery8        QEEQueryExpressionNode(myJoinNode,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
myQuery9        QEEQueryExpressionNode(myJoinNode,
                                       OOGDMUShort("1"),
                                       "getInhabitants");
myQuery10       QEEQueryExpressionNode(myQuery11,
                                       OOGDMUShort("0"),
                                       ">",
                                       myQuery12);
myQuery11       QEEQueryExpressionNode(myJoinNode,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
myQuery12       QEEQueryExpressionNode(myJoinNode,
                                       OOGDMUShort("0"),
                                       "getInhabitants");
```

entspricht dem ausführbaren Algebraausdruck in Beispiel 4.14.

Beispiel 4.14 Verwendung der Operatoren **sort** und **mergeJoin**

Dieser ausführbare Algebraausdruck repräsentiert den textuellen Ausführungsplan aus Beispiel 4.13.

listIndex(*oid*(“City”))
“c1”
scan [**fun**(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”))⟩)) **refBoolOp**(true)]
sort[
⟨(**fun**(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”))⟩))
applyobjmethod(“getInhabitants”, ⟨⟩, **c1**(*x*)),
refBoolOp(false))⟩]
applyclassmethod(“new”, ⟨**refUShortOp**(64)⟩, “OOGDMInt”)
listIndex(*oid*(“City”))
“c2”
scan [**fun**(*x*: *objectTuple*(⟨(“c2”, *oid*(“City”))⟩)) **refBoolOp**(true)]
sort[
⟨(**fun**(*x*: *objectTuple*(⟨(“c2”, *oid*(“City”))⟩))
applyobjmethod(“getInhabitants”, ⟨⟩, **c2**(*x*)),
refBoolOp(false))⟩]
applyclassmethod(“new”, ⟨**refUShortOp**(64)⟩, “OOGDMInt”)
fun(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))⟩))
applyobjmethod(
“>”,
⟨**applyobjmethod**(“getInhabitants”, ⟨⟩, **c2**(*x*))⟩,
applyobjmethod(“getInhabitants”, ⟨⟩, **c1**(*x*)))
mergeJoin[
fun(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))⟩))
applyobjmethod(
“==”,
⟨**applyobjmethod**(“getInhabitants”, ⟨⟩, **c2**(*x*))⟩,
applyobjmethod(“getInhabitants”, ⟨⟩, **c1**(*x*)))]

Kapitel 5

Anfrageoptimierung – Bestimmung eines Ausführungsplans für eine Anfrage

Bisher wurden in dieser Arbeit die meisten Aspekte der Anfragebearbeitung in GOODAC vorgestellt. Sowohl die Übersetzung einer OOGQL-Anfrage in einen Ausdruck der deskriptiven Algebra (siehe auch Abschnitt 2.3) als auch die Erzeugung eines textuellen Ausführungsplans ausgehend von einem Ausdruck der ausführbaren Algebra (siehe auch Abschnit 4.4.3) sowie die Abarbeitung eines derartigen Ausführungsplans (siehe auch Abschnit 3.2) wurden schon thematisiert. Mit der Anfrageoptimierung und der damit verbundenen Übersetzung eines deskriptiven Algebraausdrucks in einen Ausdruck der ausführbaren Algebra wurde eine Kernaufgabe der Anfragebearbeitung in GOODAC jedoch bisher unberücksichtigt gelassen. Daher soll die Anfrageoptimierung in erweiterbaren Datenbanksystemen und vor allem in GOODAC nun in diesem Kapitel eingeführt werden. Im Rahmen der Anfrageoptimierung in GOODAC kommt eine neue Komponente zur erweiterbaren generischen Anfrageoptimierung zum Einsatz, die ebenfalls in diesem Kapitel vorgestellt wird.

Nicht nur in GOODAC, sondern in allen Datenbanksystemen stellt die Optimierung von Anfragen einen wesentlichen Teil der Anfragebearbeitung dar [SKS02, 14]. Mit einer derartigen Optimierung einer Anfrage werden zwei Ziele verfolgt: Zum einen muss die Anfrage in einen Ausführungsplan überführt werden, der Informationen über die anzuwendenden Algorithmen und die Art der Weiterleitung von Zwischenergebnissen während der Berechnung des Anfrageergebnisses enthält (siehe auch Kapitel 3), zum anderen soll ein in Bezug auf die benötigte Ausführungszeit und den erforderlichen Speicherplatzbedarf guter Ausführungsplan bestimmt werden und insbesondere der schlechteste unberücksichtigt bleiben [Vos99, 15.3.1]. Die Auswahl eines derartigen Ausführungsplans erfolgt in der Regel unter Berücksichtigung der folgenden beiden Gesichtspunkte:

Heuristiken: Es existiert eine Vielzahl anerkannter Regeln zur Transformation eines eine Anfrage repräsentierenden Algebraausdrucks in einen neuen Algebraausdruck, der zum ursprünglichen äquivalent ist, aber in der Regel zu einer schnelleren Berechnung des Anfrageergebnisses führt. Zahlreiche Beispiele für derartige Umformungsregeln liefern Garcia-Molina *et. al.* [GMUW02, 16.2].

Kostenmodell: Der Aufwand zur Abarbeitung eines Ausführungsplans wird durch die Anwendung eines Kostenmodells abgeschätzt. Häufig werden dazu Statistiken zu Rate gezogen, um beispielsweise Informationen über die Häufigkeit bestimmter Attributwerte [EN00, 18.4.2] – etwa in Form von Histogrammen [CB02, 20.6.2] – zu erhalten. Für jeden in Frage kommenden Ausführungsplan werden bei diesem Vorgehen die erwarteten Kosten berechnet, bevor schließlich in Erwartung einer schnellen und Speicherplatz-sparenden Abarbeitung der nach dem vorliegenden Kostenmodell günstigste Ausführungsplan ausgewählt wird [GMUW02, 16.4 f].

Die meisten Datenbankmanagementsysteme verwenden im Rahmen der Anfrageoptimierung eine Kombination dieser beiden Techniken. Durch die Anwendung bestimmter Heuristiken lassen sich recht einfach Kandidaten für den auszuwählenden Ausführungsplan im Vorfeld der Kostenberechnung bestimmen, wobei oft einige Alternativen bereits durch die Anwendung der Heuristiken verworfen werden, um den Aufwand zur Kostenberechnung zu reduzieren [SKS02, 14].

In diesem Zusammenhang existieren viele Ansätze und Arbeiten, die Optimierungsstrategien, Untersuchungen zur Effizienz des Einsatzes bestimmter Heuristiken, Möglichkeiten zur Einschränkung des Suchraums oder neue Kostenmodelle vorstellen. Diese Aspekte werden hier nicht ausführlicher thematisiert, weil eine ausreichende Betrachtung den Rahmen dieser Arbeit sprengen würde; erste Übersichten geben die entsprechenden Ausführungen und die zugehörigen Literaturangaben von Elmasri und Navathe [EN00, 18], Garcia-Molina *et. al.* [GMUW02, 15 f], Silberschatz *et. al.* [SKS02, 14] sowie Vossen [Vos99, 15]. Zudem werden in dieser Arbeit weder eine effiziente Optimierungsstrategie noch ein neues Kostenmodell vorgestellt, da sich in der Literatur eine Vielzahl von möglichen Verfahren finden lässt. Einzig und allein in Abschnitt 5.3.2 und in Abschnitt 5.3.3 wird grob eine grundlegende Vorgehensweise zur Transformation von Algebraausdrücken in GOODAC beschrieben, um eine lückenlose Betrachtung der Anfragebearbeitung in GOODAC zu gewährleisten. Ein kurzer Ausblick hinsichtlich der Anfrageoptimierung in GOODAC findet sich am Ende dieser Arbeit in Abschnitt 6.2.

In diesem Kapitel wird ausgehend von einer Beschreibung erweiterbarer Komponenten zur Anfrageoptimierung (Abschnitt 5.1) eine neue generische und erweiterbare Komponente vorgestellt (Abschnitt 5.2). Diese ermöglicht eine einfache Beschreibung der anzuwendenden Optimierungsstrategie und der Transformationsvorschriften für Algebraausdrücke. Sie lässt sich in vielen Datenbankmanagementsystemen einsetzen, sofern dort eine algebraische Repräsentation von Anfragen und Ausführungsplänen gegeben ist. Dieses Kapitel schließt mit einer Beschreibung der Anfrageoptimierung unter Einsatz dieser Komponente in GOODAC (Abschnitt 5.3), um eine beispielhafte Verwendung dieser Komponente zu zeigen und die Beschreibung der Anfragebearbeitung in GOODAC zu vervollständigen. In diesem letzten Abschnitt finden sich auch Beispiele zur Optimierung von Anfragen.

5.1 Erweiterbare Komponenten zur Anfrageoptimierung

Erweiterbare Datenbankmanagementsysteme [Car87] besitzen zahlreiche Anwendungsgebiete. So stellt beispielsweise GOODAC nur ein GIS-Kernsystem dar, das von Anwendungsentwicklern – zum Beispiel durch das Hinzufügen neuer Anwendungsklassen – und von Systemprogrammierern – etwa durch neue Algorithmen auf der Ausführungsebene – ergänzt werden kann.

Diese Ergänzungsmöglichkeiten führen dazu, dass nicht nur die Ausführungsebene und das Datenmodell erweiterungsfähig sein müssen, sondern dass auch die Komponente zur Anfrageoptimierung nicht nur die Ergänzungen der Ausführungsebene und des Datenmodells berücksichtigen, sondern auch Möglichkeiten bereitstellen muss, um das Kostenmodell und die gesamte Optimierungsstrategie entsprechend anpassen zu können [KD99, 1]. Neben der grundsätzlichen Fähigkeit, Erweiterungen zu berücksichtigen, sollte eine Komponente zur Anfrageoptimierung zudem klar definierte Schnittstellen anbieten, die einfach zu handhabende Modifikationen und Ergänzungen erlauben. Dazu sollte sich etwa die Optimierungsstragie durch eine für Systemprogrammierer leicht verständliche Repräsentation ausdrücken lassen [WM99, 1].

Durch diese Möglichkeiten zur Ergänzung werden nicht nur die urspünglichen Entwickler eines Datenbankmanagementsystems in die Lage versetzt, Erweiterungen des Systems vorzunehmen oder die Optimierungsstrategie zu verändern, sondern auch erfahrene Datenbankadministratoren oder Berater können das System an die vorliegenden Gegebenheiten anpassen. Zudem bieten sich gerade im universitären Umfeld entwickelte Forschungsprototypen für den Einsatz in der Lehre an, sodass hier – etwa durch die einfache Modifizierbarkeit der Optimierungskomponente sowie die Visualisierung des Optimierungsprozesses – die Untersuchung der Auswirkungen kleiner Änderungen an der Optimierungsstrategie oder des Einfügens neuer Algorithmen auf der Ausführungsebene ermöglicht werden sollte. In diesem Kontext bietet es sich beispielsweise an, die Beschreibung der Optimierungsstrategie nicht zu kompilieren, um erforderliche Änderungen schnell durchführen und verschiedene Strategien leicht gegenüber stellen zu können. Doch auch die Entwickler einer Optimierungskomponente profitieren von einem derartigen Ansatz, da sie so leicht die unterschiedlichen Algebraausdrücke vergleichen können, die durch die Anwendung verschiedener Optimierungsstrategien erzeugt werden. Erst wenn die zu verwendende Strategie endgültig festgelegt worden ist, kann sie gegebenenfalls in eine andere Form gebracht und gemeinsam mit der Optimierungskomponente kompiliert werden, um Geschwindigkeitsvorteile während der späteren Optimierung im Produktiveinsatz des Systems zu erzielen.

Im Folgenden werden nun die Optimierungskomponenten einiger Datenbankmanagementsysteme sowie einige Ansätze zur Erzeugung Datenbankmanagementsystem-spezifischer Optimierungskomponenten auf ihre Erweiterbarkeit hin untersucht. Dabei ist die Art des unterstützten Datenmodells nicht immer wesentlich, da im Idealfall die größere Flexibilität der Komponenten dafür sorgt, dass bei Bedarf ein anderes

Datenmodell verwendet werden kann.

Mehrere Ansätze verwenden im Rahmen des Optimierungsprozesses Transformationsregeln [Gra87]. Dabei handelt es sich gewöhnlich um eine textuelle Beschreibung der Umformung bestimmter Algebraausdrücke. Durch die Verwendung derartiger Regeln sind Systemprogrammierer in der Lage, einige Teile des Optimierungsprozesses, die nicht die eigentliche Optimierungsstrategie betreffen, nachträglich anzupassen und zu erweitern, ohne den Quellcode der Komponente zur Anfrageoptimierung ändern zu müssen. Zudem zeigen Aberer *et. al.* [ACB97], dass der Einsatz von Transformationsregeln bei einer geschickten Wahl des Regelsatzes die Leistungsfähigkeit einer Komponente zur Anfrageoptimierung weder hinsichtlich ihrer Effizienz noch in Bezug auf die Qualität der erzeugten Ergebnisse beeinträchtigt. Alles in allem eignen sich regelbasierte Ansätze zur Anfrageoptimierung sehr gut für den Einsatz in erweiterbaren und objektorientierten Datenbankmanagementsystemen.

Die in diesem Abschnitt vorgestellten Ansätze ermöglichen jedoch ausschließlich eine Beschreibung algebraischer Transformationen durch Regeln. Manche Verfahren stellen in diesem Zusammenhang verschiedene Möglichkeiten zur Verwendung dieser Regeln durch die Optimierungsstrategie bereit, um die Reihenfolge vor ihrer Anwendung neu zu bestimmen oder nur einen Teil der Regeln im Rahmen des Optimierungsprozesses auszuwählen. Becker und Güting [BG92] präsentieren beispielsweise einen Regelsatz zur Anfrageoptimierung im Gral-System, der es ermöglicht, alle von ihrem Prototyp-System unterstützten Anfragen zu optimieren. Weitergehende Funktionalität zur Formulierung von weiteren Teilen der Optimierungsstrategie – wie zum Beispiel den Vergleich von Algebraausdrücken hinsichtlich der erwarteten Kosten bei Verwendung des repräsentierten Vorgehens zur Beantwortung einer Anfrage – durch Regeln bietet keiner der im Folgenden dargestellten Ansätze an.

System R

System R [SAC+79] war eines der ersten voll funktionsfähigen relationalen Datenbankmanagementsysteme [SKS02, 1.10]. Sein Einfluss findet sich in vielen später entwickelten Systemen wieder, beispielsweise im weiter unten angeführten Starburst. Die Komponente zur Anfrageoptimierung in System R setzt bereits eine Bottom-up-Strategie ein, bei der erst die einzelnen Operanden eines Operators optimiert werden, bevor der Operator selbst zum Zuge kommt und unter anderem in einen Ausdruck der Zielalgebra überführt wird. Die Optimierungsstrategie in System R berücksichtigt interessante Sortierungen (so genannte *interesting orders*), um bereits durch vorherige Operationen sortierte Teilergebnisse einzusetzen und somit den Aufwand für einen gesonderten Sortierschritt – beispielsweise im Rahmen der Duplikateliminierung oder der sortierten Ausgabe des Anfrageergebnisses – einzusparen. System R und damit auch die verwendete Komponente zur Anfrageoptimierung sind in keinerlei Hinsicht erweiterbar: Das Datenmodell, die Optimierungsstrategie und das Kostenmodell sind unveränderbar; ebenso können keine neuen Indexstrukturen, Datentypen oder Algebraoperatoren hinzugefügt werden.

EXODUS

Das erweiterbare Datenbankmanagementsystem EXODUS [GD87] besitzt einen regelbasierten Generator zur Erzeugung konkreter Komponenten zur Anfrageoptimierung [Gra87]. Dieser Generator erhält als Eingabe eine Beschreibung des Datenmodells, der Algebraoperatoren, der Indexstrukturen sowie der verfügbaren Datentypen über ein so genanntes *Model Description File*. Diese Beschreibung beschränkt sich allerdings auf die Aufzählung dieser Elemente und die Angabe wesentlicher Eigenschaften wie beispielsweise die Stelligkeit von Algebraoperatoren. Eine weitergehende Spezifikation dieser Argumente oder die Angabe eines Resultattyps erfolgen nicht. Für jeden definierten Operator müssen zudem in C geschriebene Funktionen zur Realisierung einer Kostenfunktion und Ermittlung weiterer Eigenschaften des Operators bereitgestellt werden. Die Menge sowie die Bezeichnung der zu definierenden Funktionen sind unveränderbar festgelegt. Zusätzlich erhält der Generator einen Menge von textuell beschriebenen Transformationsregeln, die mögliche Umformungen von Algebraausdrücken darstellen. Ein exemplarisches *Model Description File* für grundlegende Operatoren der relationalen Algebra und einige zugehörige Transformationsregeln findet sich in Graefes Arbeit [Gra87].

Aus diesen unterschiedlichen Eingaben erzeugt der Generator zuerst Quellcode der Programmiersprache C und anschließend eine Komponente zur Anfrageoptimierung, die sowohl durch einen Systemprogrammierer als auch durch das Datenbankmanagementsystem selbst um weitere Regeln zur Transformation von Algebraausdrücken ergänzt werden kann. Änderungen der oben genannten Eingabeparameter erfordern die vollständige Neuübersetzung der Komponente zur Anfrageoptimierung. Weiterhin sind sowohl das

Kostenmodell als auch die Optimierungsstrategie vorgegeben; allerdings kann die Komponente zur Anfrageoptimierung Erfahrungen über frühere Anfragen nutzen, um entsprechende Veränderungen an der eingesetzten Strategie vorzunehmen. Eine Änderung des Kostenmodells erfordert immer eine neue Erzeugung der Optimierungskomponente.

OGL

Die Optimierer-Generierungs-Sprache OGL (**O**ptimizer **G**enerater **L**anguage) [SS90] ermöglicht die Definition von Regeln zur Umformung von Algebraausdrücken. Diese Regeln können in Modulen organisiert werden, wobei für jedes Modul eine unterschiedliche Optimierungsstrategie zum Einsatz kommen kann. Die Unterteilung in Module bietet sich beispielsweise an, um Regeln mit unterschiedlichen Aufgaben oder verschiedenen Voraussetzungen bereits im Vorfeld voneinander zu trennen. OGL eignet sich zur Anbindung an ein beliebiges Datenbankmanagementsystem, in dem Anfragen und Ausführungspläne durch Algebraausdrücke repräsentiert werden. Somit können nicht nur das Datenmodell und das Kostenmodell frei gewählt werden, sondern auch die Berücksichtigung neuer Operatoren, Datentypen und Indexstrukturen wird ermöglicht. Allerdings ist die verwendete Optimierungsstrategie kaum veränderbar, einzig und allein einige zur Kompilierzeit der Komponte zur Anfrageoptimierung zu wählende Parameterwerte können für die Optimierung jeder Anfrage neu festgelegt werden und so die Optimierungsstrategie beeinflussen. Sollen andere Parameter berücksichtigt oder eine Optimierungsstrategie grundlegend verändert werden, ist eine komplette Neuübersetzung der Komponente zur Anfrageoptimierung mit einem entsprechenden Programmieraufwand im Vorfeld erforderlich. Zudem wird nur eine feste Menge von vier unterschiedlichen Optimierungsstrategien bereitgestellt. Das Hinzufügen neuer Strategien erfordert eine sehr gute Kenntnis der gesamten Optimierungskomponente und der bestehenden Module, in denen sie eingesetzt werden soll.

GOM

Im Rahmen des GOM-Projekts [KM90] findet eine regelbasierte Komponente zur Anfrageoptimierung Anwendung, deren Regeln ebenfalls in Gruppen zusammengefasst werden. Sowohl die verwendete grundlegende Optimierungsstrategie als auch das Kostenmodell sind unveränderbar, wobei jedoch die Strategie durch Angabe bestimmter Parameterwerte für jede Gruppe leicht angepasst werden kann. So kann beispielsweise festgelegt werden, ob höchstens eine oder so viele Regeln dieser Gruppe wie möglich auf einen gegebenen Algebraausdruck angewendet werden sollen. Durch die Angabe einer Reihenfolge für die Regeln innerhalb einer Gruppe können bestimmte Regeln bevorzugt zur Anwendung kommen. Boolesche Operationen und mengenwertige Operationen werden durch fest eingebaute Gruppen von Regeln behandelt, die eine Sonderrolle im Rahmen des Optimierungsprozesses einnehmen. Derartige Regeln können jederzeit aufgerufen werden, ohne die aktuelle Regelgruppe zu verlassen. Alle übrigen Gruppen von Regeln können nur sequentiell eingesetzt werden, wobei zu jeder Regelgruppe eine Nachfolgergruppe spezifiziert werden muss. Weiterhin kommt im GOM-Projekt ein festgelegtes objektorientiertes Datenmodell zum Einsatz, das keine Einführung neuer Operatoren oder Datentypen erlaubt. Einzig und allein neue Indexstrukturen können hinzugefügt werden.

Starburst

Im Optimierungsprozess von Starburst [PHH92] kommen ebenso Regeln zum Einsatz, die eine interne Repräsentation der gegebenen Anfrage umformen. Die Regeln zur reinen Transformation von Algebraausdrücken werden zu Gruppen (so genannten *rule groups*) zusammengefasst und kompiliert; daher muss bei jeder Änderung zumindest ein Teil der Komponente zur Anfrageoptimierung neu erzeugt werden. Die Anwendung dieser Regeln erfolgt nach einer festgelegten Strategie, die jedoch im Vorfeld der Erzeugung der Optimierungskomponente angepasst werden kann, indem beispielsweise die Reihenfolge der Regeln innerhalb einer Gruppe verändert wird. Als unterliegendes Datenmodell ist ausschließlich das relationale Modell zugelassen. Transformationsregeln zur Abbildung von Algebraausdrücken, die von Benutzern gestellte Anfragen darstellen, auf die Repräsentation eines Ausführungsplans werden interpretiert, sodass ohne Neuübersetzung der Komponente zur Anfrageoptimierung insbesondere neue Indexstrukturen und Operatoren der ausführbaren Ebene berücksichtigt werden können [LFL88]. Auch neue Datentypen können eingefügt werden, ebenso wie das Kostenmodell veränderbar ist. In Starburst werden Kosten nicht als Ergebnis nach Anwendung einer bestimmten Kostenfunktion, sondern als eine Eigenschaft eines Ausführungsplans angesehen. Das erleichtert ihre Verwendung, da somit durch den Einsatz unterschiedlicher Eigenschaften

auch die gleichzeitige Berücksichtigung unterschiedlicher Kostenmodelle – zum Beispiel für Vergleichszwecke – möglich wird. Der Name der zu berücksichtigenden Eigenschaft wird dazu in den Transformationsregeln angegeben, während der konkrete Wert einer Eigenschaft für jeden Ausführungsplan bei der Anwendung einer Transformationsregel von der Komponente zur Anfrageoptimierung berechnet wird. Weitere Eigenschaften von Ausführungsplänen repräsentieren etwa vorliegende Sortierungen des durch Teile eines Ausführungsplans repräsentierten Anfrageergebnisses, die für die Auswahl nachfolgender Algorithmen zur Realisierung bestimmter Operationen ausgenutzt werden können. Allerdings erfordert die Verwendung neuer Eigenschaften immer auch die Veränderung der Transformationsregeln, sodass eine Neuübersetzung des Anfrageoptimierers unvermeidlich ist. Die Verwendung von Eigenschaften für Ausführungspläne wurde aufgrund der Mächtigkeit dieses Konzepts in EGO übernommen (siehe auch Abschnitt 5.2.1). Dort können sowohl deskriptive als auch ausführbare Algebraausdrücke beliebige Eigenschaften besitzen.

Gral

Die in Gral [BG92] eingesetzte Komponente zur regelbasierten Anfrageoptimierung strukturiert die Regeln ebenfalls in Gruppen. Für jede dieser Gruppen lässt sich eine von drei festgelegten Optimierungsstrategien zur Anwendung der enthaltenen Regeln bestimmen. Diese Strategien berücksichtigen die Reihenfolge der Regeln innerhalb der Gruppe, die Anzahl der bisher angewendeten Regeln sowie ein Kostenmodell für Ausführungspläne. Eine Ergänzung weiterer Strategien ist allerdings nicht vorgesehen. Gral und seine Komponente zur Anfragebearbeitung stellen einen einfachen Weg bereit, um neue Attributtypen, Operatoren und Indexstrukturen in das System einzubetten. Hierzu ist im Wesentlichen nur die Definition neuer Transformationsregeln erforderlich, die diese neuen Bestandteile des Datenmodells und der zum Einsatz kommenden Algebren berücksichtigen. Allerdings sind neben den drei Optimierungsstrategien auch das Kostenmodell und das zugrunde liegende relationale Datenmodell unveränderbar. Dieses Kostenmodell ist zudem sehr einfach strukturiert. Daneben müssen einige Parameter zur Berechnung der Kosten bei der Definition der Transformationsregeln angegeben werden. Sollten diese Werte verändert werden müssen, um etwa neue Operatoren oder Indexstrukturen zu berücksichtigen, ist eine sehr gute Kenntnis der gesamten Regelmenge vonnöten.

Blackboard Architecture

Das Konzept der Blackboard Architecture [KMP93] stellt einen allgemeinen Rahmen zur Erzeugung einer konkreten Komponente zur Anfrageoptimierung bereit. Es werden hier keine Regeln, sondern aus dem Bereich der Künstlichen Intelligenz bekannte Wissensbasen (so genannte *knowledge sources*) eingesetzt, um die Erweiterbarkeit der erzeugten Komponente zu gewährleisten. Dabei wird eine erste Repräsentation einer Anfrage unter Zuhilfenahme einer ersten Wissensbasis zu alternativen Darstellungen umgeformt, die erneut durch die Verwendung einer weiteren Wissensbasis transformiert werden. Dieser Vorgang setzt sich fort, bis aus der Menge der letztendlich erzeugten Ausführungspläne der nach einem festgelegten Kostenmodell am günstigsten erscheinende ausgewählt wird. Kemper *et. al.* verwenden für ihren Optimierungsansatz sieben unterschiedliche Wissensbasen, die jeweils unterschiedliche Aufgaben im Rahmen des Optimierungsprozesses wahrnehmen. Das verwendete Kostenmodell kann durch einige Parameter angepasst werden, ein grundsätzlicher Austausch ist allerdings nur durch die Erzeugung einer völlig neuen Komponente zur Anfrageoptimierung möglich. Ebenso sind für eine bestehende Komponente keine Änderungen der verwendeten Optimierungsstrategie, des Datenmodells sowie der eingesetzten Operatoren und Datentypen realisierbar. Schließlich erfordert die Verwendung neuer Wissensbasen oder die Veränderung der bestehenden Komponenten eine vollständige Regenerierung der Optimierungskomponente.

VODAK

VODAK [AF93] stellt ebenfalls Möglichkeiten zur Erweiterbarkeit der an der Optimierung einer Anfrage beteiligten Komponenten bereit. So lassen sich leicht sowohl neue Algebraoperatoren und Datentypen als auch Datenmodelle und Kostenmodelle berücksichtigen, indem die bestehenden algebraischen Repräsentationen entsprechend verändert oder erweitert werden. Für die Aktivierung der Änderungen ist jedoch auch bei kleinen Modifikationen eine Neuübersetzung von Teilen des Datenbankmanagementsystems erforderlich. Weiterhin können neue Indexstrukturen leicht in die Komponente zur Anfrageoptimierung integriert werden, indem ebenfalls eine Erweiterung der algebraischen Repräsentation erfolgt. Einzig und allein die verwendete Optimierungsstrategie kann nur schwer verändert werden. Nur die Erweiterung um benutzerdefinierte Äquivalenzregeln zur Transformation von Algebraausdrücken ist vorgesehen.

Volcano

Der im Rahmen von Volcano [GM93] eingesetzte Generator benötigt zur Erzeugung einer konkreten Komponente zur regelbasierten Anfrageoptimierung eine Beschreibung des Daten- und des Kostenmodells sowie der verwendeten Algebra, Datentypen und Indexstrukturen. Diese Beschreibung erfolgt textuell nach einem vorgegebenen Rahmen. Anschließend wird diese Beschreibung von Volcano in den C++-Quellcode übersetzt, der anschließend mit den übrigen Bestandteilen einer Optimierungskomponente kompiliert werden kann. Soll eine dieser beschriebenen Komponenten verändert oder erweitert werden, muss ein neuer Generierungsprozess ausgehend von der modifizierten textuellen Beschreibung erfolgen. Das grobe Konzept der Optimierungsstrategie in Volcano ist festgelegt, allerdings kann ein Systemprogrammierer aussichtsreiche Funktionen (so genannte *promising functions*) bereitstellen, die die Reihenfolge der anzuwendenden Regeln unter Berücksichtigung des Kostenmodells beeinflussen können, um die in einer bestimmten Situation am aussichtsreichsten erscheinenden Regeln festzulegen. Diese zugrunde liegende Optimierungsstrategie kann somit selbst bei einer Regenerierung der Optimierungskomponente ausschließlich durch Bereitstellung einer neuen aussichtsreichen Funktion verändert werden. Die Aufteilung des zu betrachtenden Suchraums und die Anzahl der zu berücksichtigenden Alternativen im Rahmen des Optimierungsprozesses ist für alle durch Volcano erzeugten Optimierer gleich. In Volcano wird die Menge der während der Anfrageoptimierung einzusetzenden Regeln in Transformationsregeln zur Formulierung algebraischer Äquivalenzumformungen und Implementationsregeln zur Abbildung von algebraischen Operatoren auf konkrete Algorithmenrepräsentationen unterteilt. Diese Regeln müssen vor Erzeugung einer konkreten Optimierungskomponente kompiliert werden, bevor sie zum Einsatz kommen können.

Der hier beschriebene Generator zur Erzeugung einer Optimierungskomponente aus dem Volcano-Projekt kann auch in anderen Datenbankmanagementsystemen Verwendung finden, solange dort ebenfalls eine im geforderten Format vorliegende textuelle Beschreibung der zugrunde liegenden Bestandteile – zum Beispiel des Datenmodells oder der zum Einsatz kommenden Algebra – existiert.

Open OODB

Die Anfragebearbeitung im objektorientierten Datenbankmanagementsystem Open OODB [BMG93] wird durch die Anwendung von Volcano realisiert. Dadurch befindet sich die Erweiterbarkeit auf dem gleichen Niveau wie bei anderen durch Volcano realisierten Komponenten zur Anfragebearbeitung. Allerdings benennen Blakeley *et. al.* bereits einige Schwachstellen Volcanos – beispielsweise die schwere Verständlichkeit der eingesetzten Regeldefinitionen oder die fehlende Möglichkeit zur Transformation boolescher Ausdrücke und logischer Verknüpfungen –, sodass hier schon einige Ansätze für Verbesserungsmöglichkeiten im Rahmen der Entwicklung einer neuen Komponente zur Anfrageoptimierung deutlich werden.

Prairie

Unter Verwendung von Volcano stellt Prairie [DB95] ebenfalls einige Möglichkeiten bereit, um die Verwendung dieses Generators für die Erzeugung einer Komponente zur Anfrageoptimierung zu vereinfachen. Es werden zwar hinsichtlich der Erweiterbarkeit keine zusätzlichen Möglichkeiten geschaffen, allerdings werden die verwendeten Regeln im Vergleich zu Volcano besser strukturiert und einige sehr technische Details von Volcano gekapselt, sodass sich Prairie als Schnittstelle zur Verwendung von Volcano im Rahmen der Generierung einer Komponente zur Anfrageoptimierung eignet. Weiterhin wurden mit Prairie im Vergleich zu Volcano einige Konstrukte für Spezialfälle entfernt ohne die Funktionalität der Regeln einzuschränken. Daneben sorgt Prairie im Rahmen der Kapselung von Volcano dafür, dass einige für Volcano erforderliche textuelle Beschreibungen sich direkt aus den in Prairie gegebenen Regelspezifikationen erzeugen lassen.

MOOD

Auch MOOD [DAOD95] stellt eine durch Volcano erzeugte Komponente zur Anfrageoptimierung in einem objektorientierten Datenbankmanagementsystem bereit. Dadurch ergeben sich hier die zuvor für Volcano, Open OODB und Prairie beschriebenen Eigenschaften. Allerdings bleiben die Autoren mit ihrer Untersuchung hinter den bei der Entwicklung von Open OODB und Prairie erzielten Ergebnissen zurück. Insbesondere gehen sie nicht auf Schwierigkeiten oder Nachteile bei der Verwendung von Volcano zur Erzeugung einer Optimierungskomponente für ihr Datenbankmanagementsystem ein.

EDS

Die in EDS [FG94] zum Einsatz kommende Optimierungskomponente stellt neben reinen Transformationsregeln auch Möglichkeiten zur Formulierung der Optimierungsstrategie durch Regeln bereit. Allerdings werden für die Repräsentation der Strategie nur zwei Alternativen zur Auswahl der Transformationsregeln angeboten, die sich miteinander kombinieren lassen. Diese entsprechen im Wesentlichen einer sequentiellen oder zufälligen Anwendung der verfügbaren Regeln mit der Option zur Limitierung der Anzahl der erfolgenden Regelanwendungen. Zusammen mit der Möglichkeit, Bedingungen und Kostenberechnungen in den Transformationsregeln zu verwenden, wird so eine erweiterbare und anpassbare Komponente zur Anfragebearbeitung geschaffen. Allerdings stützen sich alle Regeln sehr stark auf bestehende Komponenten von EDS ab. So kann das zugrunde liegende relationale Datenmodell nicht verändert werden; auch ein Hinzufügen neuer Operatoren und Datentypen sowie ihre Verwendung im Rahmen der Anfragebearbeitung werden dadurch erschwert. Aber auch die Regeln zur Formulierung der Optimierungsstrategie benutzen vorwiegend die fest vorgegebene Funktionalität von EDS. Im Wesentlichen verwenden Sie jeweils eine Kombination der beiden zur Verfügung stehenden Verfahren zur Anwendung der Transformationsregeln und rufen zudem weitere in C++ programmierte Funktionen auf. Aus diesem Grund führen Veränderungen der eingesetzten Optimierungsstrategie oder des verwendeten Kostenmodells zu hohem Programmieraufwand.

COKO-KOLA

In COKO-KOLA [CZ98a] kommen ebenfalls Regeln im Rahmen der Anfragebearbeitung zum Einsatz. Vor ihrem Einsatz ist zwar eine Übersetzung erforderlich, allerdings können sie unabhängig von der Komponente zur Anfrageoptimierung verändert und ergänzt werden. Daneben können auch neue Operatoren in das vorgegebene Datenmodell KOLA aufgenommen werden, dessen grundlegende Struktur allerdings nicht veränderbar ist. Die zur Verfügung stehenden Datentypen und Indexstrukturen lassen sich in COKO-KOLA nicht ergänzen. Im Rahmen der Optimierungsstrategie können nur vier vorgegebene Arten der Regelanwendung zum Einsatz kommen, eine Erweiterung ist hier nicht vorgesehen. Durch eine Kombination dieser Verfahren und eine geschickte Wahl des Zusammenspiels der definierten Transformationsregeln wird allerdings die Mächtigkeit der Transformationsregeln deutlich. Daneben bietet COKO-KOLA zusätzlich noch einige fest eingebaute Verfahren zur Traversierung der Baum-Repräsentation eines Algebraausdrucks sowie zur Berücksichtigung bestimmter Beziehungen zwischen den betrachteten Daten an [CZ98b], um beispielsweise Schlüsselattribute [SKS02, 3.1.3] zu erkennen. Weiterhin wird – im Gegensatz zu durch Volcano erzeugten Komponenten zur Anfrageoptimierung – auch die Optimierung boolescher Ausdrücke unterstützt. Der Optimierungsprozess im Rahmen von COKO-KOLA ermöglicht nur die Betrachtung genau eines Algebraausdrucks zu einem bestimmten Zeitpunkt, sodass insbesondere keine Alternativen miteinander verglichen werden können und demzufolge die Optimierungsstrategie sehr einfach aufgebaut ist. Dies liegt vor allem darin begründet, dass die Regeln in COKO-KOLA formal auf Widerspruchsfreiheit und Vollständigkeit getestet werden sollen. Hierzu ist eine möglicht einfache Optimierungsstrategie ohne die Betrachtung von möglichen Alternativen erforderlich, um einen entsprechenden Beweis führen zu können.

Opt++

Die Komponente zur Anfrageoptimierung in Opt++ [KD99] ermöglicht eine gute Erweiterbarkeit nicht nur bezüglich des Daten- und Kostenmodells sowie der verwendeten Optimierungsstrategie, sondern gewährleistet auch das einfache Hinzufügen neuer Operatoren, Datentypen und Indexstrukturen. In Opt++ wird hierbei allerdings keine regelbasierte Optimierungsstrategie eingesetzt, stattdessen gewährleisten die objektorientierte Programmierung und strikte Modularisierung die umfangreiche Funktionalität der Anfrageoptimierung. Dieses führt allerdings dazu, dass bereits kleine Änderungen eine teilweise Neuübersetzung der Komponente zur Anfrageoptimierung erfordern.

Venus

Venus [WM99] ist eine in C++ eingebettete Regelsprache für Datenbankmanagementsysteme. Die basierend auf Venus entwickelte erweiterbare Komponente zur Anfrageoptimierung ermöglicht ebenfalls eine sehr große Erweiterbarkeit. Nicht nur das Daten- und das Kostenmodell sind erweitbar, zudem können auch neue Operatoren und Datentypen sowie zusätzliche Indexstrukturen im Rahmen der Anfrageoptimierung berücksichtigt werden. Weiterhin bietet die Optimierungsstrategie einige unterschiedliche Möglichkeiten

zur Auswahl und Anwendung von Optimierungsregeln, jedoch lassen sich keine Modifikationen an der verwendeten Strategie vornehmen. Das Hinzufügen neuer Strategien zur Regelanwendung ist ebenfalls nicht vorgesehen. Die Regeln werden in Venus in Modulen organisiert und innerhalb dieser Module nach Prioritäten geordnet. Innerhalb der Regeln lassen sich weitere Module, jedoch keine einzelnen Regeln aktivieren. Dieses muss über nur eine Regel enthaltende Module simuliert werden. Sobald ein Modul aktiviert wurde, werden alle Regeln dieses Moduls gemäß ihrer Prioritäten auf die Elemente der eingehenden Menge von Algebraausdrücken angewendet. Es ist also durchaus möglich, dass während des gesamten Optimierungsprozesses eine Vielzahl alternativer Algebraausdrücke zwischen den Modulen ausgetauscht wird. Die Anwendung aller Regeln eines Moduls führt schnell dazu, dass die Anzahl der alternativen Algebraausdrücke rapide steigt. Daher müssen früh Module zum Einsatz kommen, die über eine Kostenfunktion die Anzahl der Alternativen reduzieren. Die Definition, Änderung und Erweiterung der eingesetzten Regeln gestaltet sich zudem recht komplex, da von einem Systemprogrammierer zuerst eine textuelle Beschreibung entsprechender Venus-Module erzeugt werden muss. Diese Venus-Module lassen sich anschließend in C++-Quellcode übersetzen, bevor die vollständige Komponente zur Anfrageoptimierung neu kompiliert wird. Selbst kleine Änderungen an den verwendeten Regeln führen demzufolge zu einer umfangreichen Neuübersetzung des entwickelten Systems.

ObjSQL

Kvalbein verfolgt mit ObjSQL einen anderen als die bisher vorgestellten Ansätze [Kva02]. In ObjSQL wird eine Komponente zur Anfrageoptimierung und Anfrageausführung bereitgestellt, die sich nicht verändern oder erweitern lässt. Allerdings ist ObjSQL unabhängig von der Art der Speicherung der zu verarbeitenden Daten. Während der Ausführung einer Anfrage greift ObjSQL nämlich auf eine anwendungsspezifische Schnittstelle zu, die Daten in einem an das relationale Modell angelehnten Format bereitstellt. Daher lässt sich diese Komponente in zahlreichen Gebieten einsetzen, in denen Datenbanksysteme bisher keine Rolle spielen – beispielsweise bei der Verwaltung von Zugriffsrechten für das zugrunde liegende Dateisystem. Hinsichtlich der eingesetzten Konzepte zur Anfrageoptimierung fällt ObjSQL jedoch hinter fast alle anderen Systeme zurück.

PostgreSQL

Stellvertretend für kommerzielle und andere einen bedeutenden Marktanteil besitzende Datenbankmanagementsysteme wird an dieser Stelle kurz die Komponente zur Anfrageoptimierung in PostgreSQL [Pos03] vorgestellt. In PostgreSQL sind sowohl das Datenmodell und das Kostenmodell als auch die Algebraoperatoren und Indexstrukturen vorgegeben und unveränderbar. Allerdings lassen sich durch die Unterstützung objektrelationaler Gesichtspunkte neue Datentypen definieren und innerhalb des Optimierungsprozesses berücksichtigen [Pos03, 33.2]. Auch die auf Statistiken beruhende Optimierungsstrategie steht im Wesentlichen fest [Pos03, 13], allerdings sorgen verschiedene zur Laufzeit änderbare Parameter dafür, dass sich die Optimierungsstrategie an besondere Gegebenheiten anpassen lässt [Pos03, 16.4.1]. Eine weitere Veränderung und Erweiterung der Optimierungsstrategie ist zwar nicht möglich, dafür stellt PostgreSQL aber bereits eine sehr gute und ausgereifte Strategie zur Verfügung, die sich im jahrelangen Praxiseinsatz bewährt hat. Zudem ist dieses Datenbankmanagementsystem aufgrund der Betonung seiner Praxistauglichkeit weniger für Forschungszwecke als vielmehr für den alltäglichen Betrieb ausgelegt.

Zusammenfassung

Mit Hilfe der durch Tabelle 5.1 gegebenen Übersicht lassen sich die zuvor dargestellten Eigenschaften der einzelnen Komponenten zur Anfrageoptimierung in erweiterbaren Datenbankmanagementsystemen leicht zusammenfassen. Es wird deutlich, dass sich insgesamt zwei Gruppen von Systemen hinsichtlich ihrer leichten und umfangreichen Erweiterbarkeit abgrenzen lassen.

Starburst und Gral bieten jeweils eine sehr einfach zu erlernende Schnittstelle, um Änderungen und Ergänzungen des Datenmodells und der ausführbaren Ebene während des Optimierungsprozesses zu berücksichtigen. Insbesondere ist hierzu zumindest teilweise keine Übersetzung von Teilen der Komponente zur Anfrageoptimierung erforderlich. Allerdings sind sowohl die Optimierungsstrategie als auch – zumindest in Gral – das Kostenmodell nur wenig flexibel und nicht erweiterbar; zudem lässt sich das zugrunde liegende relationale Datenmodell nicht austauschen.

Opt++ sowie VODAK, Volcano, Prairie und Venus bieten jeweils gute Erweiterungsmöglichkeiten in allen betrachteten Richtungen, wobei sich die Optimierungsstrategie in Opt++ deutlich besser verändern

Tabelle 5.1 Erweiterbarkeit verschiedener Komponenten zur Anfrageoptimierung

DBMS	Algebra-operatoren	Datentypen	Index-strukturen	Daten-modell	Strategie	Kosten-modell
System R	○	○	○	○	○	○
EXODUS	(●)	(●)	(●)	(○)	(○)	(○)
OGL	(●)	(●)	(●)	(●)	(○)	(●)
GOM	○	○	(●)	○	(○)	○
Starburst	●	(●)	●	○	(○)	(●)
Gral	●	●	●	○	(○)	(○)
Blackb.	(●)	(○)	(●)	(○)	(○)	(○)
VODAK	(●)	(●)	(●)	(●)	(○)	(●)
Volcano	(●)	(●)	(●)	(●)	(○)	(●)
Open OODB	(●)	(●)	(●)	(●)	(○)	(●)
Prairie	(●)	(●)	(●)	(●)	(○)	(●)
MOOD	(●)	(●)	(●)	(●)	(○)	(●)
EDS	(○)	(○)	(○)	○	(○)	(○)
COKO	○	○	○	○	(○)	○
Opt++	(●)	(●)	(●)	(●)	(●)	(●)
Venus	(●)	(●)	(●)	(●)	(○)	(●)
ObjSQL	○	○	(●)	○	○	○
PostgreSQL	○	●	○	○	(○)	○

- ○ Nicht erweiterbar
- (○) Geringe Erweiterbarkeit, Kompilation der Erweiterungen erforderlich
- (●) Gute Erweiterbarkeit, Kompilation der Erweiterungen erforderlich
- ● Gute Erweiterbarkeit, keine Kompilation der Erweiterungen erforderlich

Tabelle 5.2 Wünschenswerte Erweiterbarkeit einer Komponente zur Anfrageoptimierung

DBMS	Algebra-operatoren	Datentypen	Index-strukturen	Daten-modell	Strategie	Kosten-modell
Gral ∪ Opt++	●	●	●	(●)	(●)	(●)
Variante 1	●	●	●	(●)	●	(●)
Variante 2	●	●	●	●	●	●

lässt als bei allen anderen Komponenten zur Anfrageoptimierung. Allerdings erfordern alle Systeme für jede Änderung oder Ergänzung eine teilweise Neuübersetzung, die sich vor allem bei einer Venus-basierten Anfrageoptimierung aufgrund der engen Anbindung von Venus an C++ recht aufwendig gestaltet. Die anderen Komponenten setzen keine Regeln zur Anfrageoptimierung ein, daher verlieren diese Systeme leicht an Modularität. Außerdem muss sich ein Systemprogrammierer in jedem Fall sehr gut in das System einarbeiten, um Änderungen vornehmen zu können. Die Schnittstellen sind zwar meistens klar definiert, erfordern allerdings eine große Kenntnis der Zusammenhänge des Systems. Insbesondere kleine Modifikationen einer bestehenden Optimierungsstrategie werden durch die genannten Punkte in allen Systemen erschwert.

5.2 EGO – Eine neue erweiterbare generische Komponente zur Anfrageoptimierung

Die Untersuchung veschiedener Komponenten zur Anfrageoptimierung im vorherigen Abschnitt hat ergeben, dass kein System eine in jeder Hinsicht leicht erweiterbare Komponente zur Anfrageoptimierung anbietet. Tabelle 5.2 zeigt, dass bereits eine Vereinigung der Vorzüge von Gral und Opt++ zu einer deutlich besseren Erweiterbarkeit gegenüber den im vorherigen Abschnitt vorgestellten Systemen führen würde.

Die hauptsächlichen Ergänzungen in Datenbankmanagementsystemen betreffen Operatoren, Datentypen und Indexstrukturen, daher sollten sich diese ähnlich wie in Gral ohne eine teilweise Neuübersetzung des Systems auch im Kontext der Anfrageoptimierung berücksichtigen lassen. Allerdings darf eine erweiterbare Komponente zur Anfrageoptimierung daneben keine zu strengen Anforderungen an das Datenmodell

stellen, um die Ergänzungs- und Modifikationsmöglichkeiten nicht zu stark einzuschränken. Auch das Kostenmodell sollte zwar modifizierbar sein, jedoch treten hier Anpassungen nicht so häufig wie beispielsweise hinsichtlich der eingesetzten Optimierungsstrategie auf. Daher sollte sich diese ebenso leicht verändern oder austauschen lassen wie Informationen über Operatoren und Indexstrukturen. Eine Zusammenfassung dieser Ziele liefert die in Tabelle 5.2 gezeigte Variante 1, während Variante 2 optimale Erweiterbarkeit unter allen zuvor genannten Aspekten garantiert. Auch Lee *et. al.* [LFL88] weisen bereits darauf hin, dass die Behandlung von Regeln als Daten und eine entsprechende Interpretation durch die Komponente zur Anfrageoptimierung zahlreiche Vorteile in sich birgt. Sie übertragen dieses Konzept allerdings nicht auf die Formulierung der Optimierungsstrategie.

In diesem Abschnitt wird eine neue generische, erweiterbare Komponente zur Anfrageoptimierung vorgestellt, die alle Eigenschaften der zuvor erwähnten Variante 1 erfüllt und fast der optimalen Variante 2 entspricht. Diese neue Komponente *EGO* (**E**xtensible **G**eneric **O**ptimizer) ist an kein spezielles Datenbankmanagementsystem gebunden, stellt aber trotzdem jegliche erforderliche Funktionalität zur Anfrageoptimierung bereit und kann problemlos in einer Vielzahl von Datenbankmanagementsystemen eingesetzt werden. Insbesondere die Möglichkeit zur regelbasierten Formulierung der Optimierungsstrategie stellt einen wesentlichen Fortschritt von EGO gegenüber den zuvor untersuchten erweiterbaren Komponenten zur Anfrageoptimierung dar. Weiterhin existiert – etwa im Gegensatz zu Open OODB [BMG93] – in EGO nur ein Typ von Optimierungsregeln, sodass eine Vereinfachung der Benutzerschnittstelle und eine größere Ausdruckskraft realisiert wird. Durch die Möglichkeit, Optimierungsregeln in Gruppen zusammenzufassen, kann eine Aufteilung in unterschiedliche Regeltypen bei Bedarf dennoch simuliert werden.

Abschnitt 5.2.1 präsentiert in diesem Zusammenhang die bereitgestellten Schnittstellen, die Systemprogrammierern und Anwendungsentwicklern zur Verfügung stehen, um diese Komponente zur Anfrageoptimierung in einem konkreten Datenbankmanagementsystem einzusetzen. In Abschnitt 5.2.2 wird nachfolgend die interne Realisierung von EGO ausführlich dargestellt. Abschnitt 5.2.3 enthält schließlich einige ausführliche Beispiele, um den Einsatz der bis dahin vorgestellten Konzepte näher zu erläutern.

5.2.1 Bereitgestellte Schnittstellen

In diesem Abschnitt werden die Möglichkeiten vorgestellt, mit denen EGO an ein konkretes Datenbankmanagementsystem angebunden werden kann. Den wichtigsten Platz nehmen dabei die Optimierungsregeln ein, daher wird ihr Aufbau zuerst allgemein und anschließend durch Beispiele dargestellt. Diese Optimierungsregeln können eine textuelle Beschreibung von Algebraausdrücken enthalten. Deren Übersetzung in ein internes Format wird ebenfalls kurz thematisiert.

Als Voraussetzung an ein Datenbankmanagementsystem für die problemlose Einbettung von EGO in den Prozess der Anfragebearbeitung ist zu beachten, dass sich sowohl vom Benutzer gestellte Anfragen als auch Ausführungspläne durch textuelle Algebraausdrücke repräsentieren lassen müssen.

Aufbau und Funktionalität der Optimierungsregeln

Die Optimierungsregeln stellen den Kern zur Steuerung des Optimierungsprozesses dar. Mit ihrer Hilfe ist es möglich, neben der Transformation von Algebraausdrücken eine Strategie zur Optimierung von Anfragen zu formulieren.

Dadurch, dass die Regeln textuell angegeben und durch EGO interpretiert werden, können leicht Änderungen und Ergänzungen sowohl bezüglich der Transformation von Algebraausdrücken als auch hinsichtlich der eingesetzten Optimierungsstrategie erfolgen. Weiterhin können so verschiedene Ansätze zur Anfrageoptimierung miteinander verglichen oder die Auswirkungen bestimmter Veränderungen der Optimierungsregeln – etwa zu Schulungszwecken – veranschaulicht werden.

Im Folgenden werden der Aufbau und die Funktionalität der Optimierungsregeln mit Hilfe von Produktionen einer kontextfreien Grammatik in EBNF [Sch97] mit Startsymbol `rulegroups` beschrieben. Diese Darstellungsart findet sich auch bei der Beschreibung der OOGQL (siehe auch Abschnitt 2.2) sowie der Präsentation textueller Ausführungspläne in GOODAC (siehe auch Abschnitt 3.2.1). Eine vollständige und zusammenfassende Darstellung des Aufbaus der Optimierungsregeln liefern die Abbildungen 5.1 und 5.2. Allerdings wird dort aus Platzgründen nicht auf die Terminalsymbole eingegangen. Ebenso können bestimmte Teile der Typkorrektheit – etwa bei Verwendung des Resultats einer weiteren Regelanwendung – erst nach Kenntnis aller Regeln überprüft werden. Auch auf eine Betrachtung dieser Aspekte wird hier verzichtet.

Abbildung 5.1 Produktionen für den Aufbau von Optimierungsregeln in EBNF, Teil 1

‹assignment›	::=	‘=’ | ‘←’
‹binaryBoolOperator›	::=	AND | OR | XOR
‹callRuleOrPrimitiveStatement›	::=	STRING ‘(’ ‹parameterList› ‘)’
‹collection›	::=	’[’ [‹collectionList›] ’]’ | collection ‘.’ ‹collectionCollectionMethod› | STRING ‘.’ ‹collectionCollectionMethod›
‹collectionBoolMethod›	::=	ISEMPTY ‘(’ ‘)’ | ISEQUAL ’(’ ‹collection› ‘)’ | ISEQUAL ’(’ STRING ‘)’ | CONTAINS ‘(’ ‹collection› ‘)’ | CONTAINS ‘(’ ‹expressionOrType› ‘)’ | CONTAINS ‘(’ STRING ‘)’ | CONTAINS ‘(’ ‹rule › ‘)’
‹collectionCollectionMethod›	::=	APPEND ‘(’ ‹collection› ‘)’ | APPEND ‘(’ STRING ‘)’ | APPEND ‘(’ ‹callRuleOrPrimitiveStatement› ‘)’ | DELETE ‘(’ INTEGER [‘,’ STRING [‘.’ ‹comparison› ‘(’ ‹property› ‘)’ | ‘.’ ‹comparison› ‘(’ STRING ‘)’]] ‘)’ | GET ‘(’ INTEGER [‘,’ STRING [‘.’ ‹comparison› ‘(’ ‹property› ‘)’ | ‘.’ ‹comparison› ‘(’ STRING ‘)’]] ‘)’ | ‹insertionKeyword› ‘(’ STRING ‘)’ | ‹insertionKeyword› ‘(’ ‹expressionOrType› ‘)’ | ‹insertionKeyword› ‘(’ ‹collection› ‘)’ | ‹insertionKeyword› ‘(’ ‹rule› ‘)’ | ‹insertionKeyword› ‘(’ ‹callRuleOrPrimitiveStatement› ‘)’ | INTERSECT ‘(’ ‹collection› ‘)’ | INTERSECT ‘(’ STRING ‘)’ | INTERSECT ‘(’ ‹callRuleOrPrimitiveStatement› ‘)’ | MAKEUNIQUE ‘(’ ‘)’ | MINUS ‘(’ ‹collection› ‘)’ | MINUS ‘(’ STRING ‘)’ | MINUS ‘(’ ‹callRuleOrPrimitiveStatement› ‘)’
‹collInPattern›	::=	LBRACKPATT [‹collListInPattern›] RBRACKPATT
‹collectionList›	::=	[‹collectionList› ‘,’] ‹collection› | [‹collectionList› ‘,’] STRING | [‹collectionList› ‘,’] ‹expressionOrType›
‹collListInPattern›	::=	[‹collListInPattern› ‘,’] ‹collInPattern› | [‹collListInPattern› ‘,’] COLLSTRINGINPATTERN | [‹collListInPattern› ‘,’] ‹expressionOrType›
‹collectionValueMethod›	::=	SIZE ‘(’ ‘)’
‹comparison›	::=	ISLESS | ISLESSOREQUAL | ISEQUAL | ISGREATEROREQUAL | ISGREATER
‹condition›	::=	TRUE | FALSE | ‹unaryBoolOperator› ‹condition› | ‘(’ ‹condition› [‹binaryBoolOperator› ‹condition›] ‘)’ | [‹pattern› ‹assignment›] ‹callRuleOrPrimitiveStatement› | ‹collection› ‘.’ ‹collectionBoolMethod› | STRING ‘.’ ‹collectionBoolMethod› | ‹property› [‘.’ ‹comparison› ‘(’ ‹property› ‘)’ | ‘.’ ‹comparison› ‘(’ STRING ‘)’] | STRING ‘.’ ISEQUAL ‘(’ ‹expressionOrType› ‘)’
‹expressionOrType›	::=	{ ALGEBRASTRING }
‹group›	::=	GROUP GROUPNAME ‹rules› ENDGROUP
‹insertionKeyword›	::=	INSERT | INSERTUNIQUE
‹optElse›	::=	[ELSE ‹statementList›]
‹optProtected›	::=	[PROTECTED]
‹parameter›	::=	‹parameterPatternList› | INTEGER
‹parameterList›	::=	[[‹parameterList› ‘,’] ‹parameter›]
‹parameterPatternList›	::=	[‹parameterPatternList› ‘∼’] ‹expressionOrType› | [‹parameterPatternList› ‘∼’] ‹rule› | [‹parameterPatternList› ‘∼’] ‹collection› | [‹parameterPatternList› ‘∼’] STRING

Abbildung 5.2 Produktionen für den Aufbau von Optimierungsregeln in EBNF, Teil 2

‹pattern›	::=	‹collInPattern› ':' ‹type› | STRING ':' ‹type› |
		STRING ':' EXPRTYPE '(' ‹expressionOrType› ')' |
		‹expressionOrType› ':' EXPRTYPE '(' ‹expressionOrType› ')' |
		‹expressionOrType› ':' ‹type›
‹patternList›	::=	[‹patternList› ~] ‹pattern›
‹property›	::=	STRING '.' GETPROPERTY '(' ‹property› ')' |
		STRING '.' GETPROPERTY '(' STRING ')' |
		STRING '.' collectionValueMethod |
		‹collection› '.' collectionValueMethod
‹returnStatement›	::=	RETURN STRING | RETURN ‹expressionOrType› |
		RETURN ‹collection› | RETURN ‹rule› |
		RETURN ‹callRuleOrPrimitiveStatement›
‹rule›	::=	RULE ‹optProtected› STRING ':' ‹type› ‹patternList› ';' ‹statementList›
‹rulegroups›	::=	‹group› [‹rulegroups›]
‹rules›	::=	[‹rules›] ‹rule›
‹statement›	::=	‹callRuleOrPrimitiveStatement› |
		IF ‹condition› THEN ‹statementList› ‹optElse› ENDIF |
		‹pattern› ‹assignment› ‹callRuleOrPrimitiveStatement› |
		‹pattern› ‹assignment› STRING |
		‹pattern› ‹assignment› ‹expressionOrType› |
		‹pattern› ‹assignment› ‹collection› |
		FORALL ‹pattern› IN ‹collection› DO ‹statementList› ENDFOR |
		FORALL ‹pattern› IN STRING DO ‹statementList› ENDFOR
‹statementList›	::=	[‹sListWithoutReturn›] [‹returnStatement ›;']
‹sListWithoutReturn›	::=	[‹sListWithoutReturn›] ‹statement› ';'
‹type›	::=	COLLTYPE '<' ‹type› '>' |
		EXPRTYPE | RULETYPE | STRINGTYPE | TYPETYPE
‹unaryBoolOperator›	::=	NOT

Ein Systemprogrammierer gibt Optimierungsregeln in einer textuellen Beschreibung an. Diese Regeln werden gewöhnlich zu benannten Gruppen zusammengefasst, wobei jede dieser Gruppen beliebig viele Regeln enthalten kann:

‹rulegroups› ::= ‹group› [‹rulegroups›]
‹group› ::= GROUP GROUPNAME ‹rules› ENDGROUP
‹rules› ::= [‹rules›] ‹rule›

Eine einzelne Optimierungsregel ist im Wesentlichen wie folgt aufgebaut:

‹rule› ::= RULE ‹optProtected› STRING ':' ‹type› ‹patternList› ';' ‹statementList›

Eingeleitet wird eine entsprechende Definition durch das Schlüsselwort `RULE`. Folgt anschließend das optional anzugebende Schlüsselwort `PROTECTED`, so kann die Regel später nur durch Angabe ihres nun folgenden Namens angewendet werden. Andernfalls ist eine Regelanwendung auch möglich, ohne dass ihr Name bekannt ist – beispielsweise bei der Ausführung aller Regeln einer bestimmten Gruppe. Diese Unterscheidung bietet sich an, um einige Regeln vor versehentlicher Ausführung zu schützen. Eine Regeldefinition wird mit dem Rückgabetyp der Regel fortgesetzt.

‹type› ::= COLLTYPE '<' ‹type› '>' \|
EXPRTYPE \| RULETYPE \| STRINGTYPE \| TYPETYPE

Im Rahmen von Regeln sind also insgesamt fünf unterschiedliche Typen zulässig, zu denen entsprechende *Elemente* existieren:

COLLTYPE '<' ‹type› '>': Kollektionen können Elemente eines Typs – gegebenenfalls erneut eine Kollektion – beinhalten. Auch Regelgruppen werden als Kollektionen aufgefasst. So können in Regeln beispielsweise Kollektionen von Algebraausdrücken verwendet werden, um unterschiedliche Repräsentationen von Anfragen oder Ausführungsplänen miteinander zu vergleichen, während es etwa

Kollektionen von Regeln ermöglichen, alle enthaltenen Regeln auf einen gegebenen Algebraausdruck anzuwenden.

EXPRTYPE: Algebraausdrücke repräsentieren in der Regel Anfragen oder Ausführungspläne.

RULETYPE: Auch Regeln können einen Rückgabetyp darstellen – etwa bei der Extraktion einer bestimmten Regel aus einer Kollektion von Regeln.

STRINGTYPE: Zur einfachen Übergabe Benutzer-definierter Parameter und zur Behandlung bestimmter Eigenschaften der verarbeiteten Algebraausdrücke stellt EGO einen Typ für Zeichenketten bereit.

TYPETYPE: Nicht nur Algebraausdrücke, sondern auch die Typen eines Algebraausdrucks spielen während des Optimierungsprozesses eine wesentliche Rolle – beispielsweise bei der Überprüfung, ob ein gegebener Algebraausdruck ein einzelnes oder einen Strom von Datenbank-Elementen repräsentiert.

In EGO existieren also fünf verschiedene Arten von Elementen: Kollektionen, Algebraausdrücke, Regeln, Zeichenketten und Algebratypen.

Weiterhin enthält jede Regeldefinition eine Liste von Mustern. Wird eine Regel aufgerufen, kommt sie nur zur Anwendung, falls die übergebenen Parameterwerte mit der vorgegebenen Musterliste übereinstimmen. Die einzelnen Muster werden durch ein Tildesymbol (~) voneinander abgegrenzt.

```
<patternList>   ::=  [ <patternList> ~ ] <pattern>
<pattern>       ::=  <collInPattern> ':' <type> | STRING ':' <type> |
                     STRING ':' EXPRTYPE '(' <expressionOrType> ')' |
                     <expressionOrType> ':' EXPRTYPE '(' <expressionOrType> ')' |
                     <expressionOrType> ':' <type>
```

Einfache Muster bestehen nur aus einem Bezeichner `STRING` und einer Typangabe. Bei Aufruf der Regel wird der übergebene Parameterwert bei Typgleichheit an eben diesen Bezeichner gebunden. Zusätzlich besteht die Möglichkeit, komplexe Kollektionen, Algebraausdrücke und Algebratypen als Muster zu verwenden. Kollektionen können durch nähere Spezifikation der enthaltenen Elemente

```
<collInPattern>      ::=  LBRACKPATT [ <collListInPattern> ] RBRACKPATT
<collListInPattern>  ::=  [ <collListInPattern> ',' ] <collInPattern> |
                          [ <collListInPattern> ',' ] COLLSTRINGINPATTERN |
                          [ <collListInPattern> ',' ] <expressionOrType>
```

genauer bestimmt werden. So würde etwa `[elemA,elemB]` eine Kollektion mit genau zwei Elementen definieren. Der Typ dieser Elemente – etwa `STRING` – wird durch den Typ dieser Kollektion – beispielsweise `Coll<STRING>` – festgelegt. Für mehrfach geschachtelte Kollektionen lassen sich die Elemente jeder Stufe auf die gleiche Art und Weise beschreiben.

Algebraausdrücke und Algebratypen werden durch die Produktion

```
<expressionOrType>   ::=  { ALGEBRASTRING }
```

näher definiert. Weil an die textuelle Darstellung von Algebraausdrücken keine Anforderungen gestellt werden, können diese auch nicht durch EGO analysiert werden. Stattdessen werden erkannte Algebraausdrücke und Algebratypen an eine für jedes konkrete Datenbankmanagementsystem zu entwickelnde Konvertierungskomponente (siehe auch das Ende dieses Abschnitts) weitergereicht, die eine interne objektbasierte Darstellung der Algebraausdrücke und Algebratypen erzeugt, die sich im Rahmen der von EGO gesteuerten Anfrageoptimierung verwenden lassen.

Sollten die übergebenen Parameter bei Aufruf einer Regel auf die Bestandteile der Musterliste passen, kommt schließlich die Anweisungsliste der Regeldefinition zur Ausführung. Sie wird gegebenfalls mit einer Rückgabeanweisung abgeschlossen.

```
<statementList>       ::=  [ <sListWithoutReturn> ] [ <returnStatement > ';' ]
<sListWithoutReturn>  ::=  [ <sListWithoutReturn> ] <statement> ';'
<returnStatement>     ::=  RETURN STRING | RETURN <expressionOrType> |
                           RETURN <collection> | RETURN <rule> |
                           RETURN <callRuleOrPrimitiveStatement>
```

Als Rückgabewerte kommen konkrete Ausprägungen der oben genannten Typen zum Tragen. Zusätzlich besteht die Möglichkeit, ein – beispielsweise infolge des angegebenen Musters zu Beginn der Regeldefinition – unter einem Bezeichner zugreifbares Element zurückzuliefern. Schließlich kann eine weitere Regel oder ein Primitiv aufgerufen werden; das zugehörige Resultat bildet in diesem Fall auch den Rückgabewert der vorliegenden Regel.

Primitive stellen in diesem Kontext zusätzliche Funktionalität bereit, die vom Systemprogrammierer in der Programmiersprache C++ realisiert worden ist. Zum einen lassen sich hierdurch bereits vorhandene Konzepte in den neuen Optimierungsprozess einbinden, zum anderen können auf diese Art komplexe Berechnungen erfolgen, falls sie sich nicht durch die Optimierungsregeln darstellen lassen. Die Ausdruckskraft der Optimierungsregeln in EGO ist allerdings bereits so mächtig, dass dieses nur in Ausnahmefällen zum Tragen kommt. Werden die neuen Primitive zudem möglicht allgemein definiert, eignen sie sich zum Einsatz unter verschiedenen Datenmodellen und Optimierungsstrategien, wodurch der Aufwand sowohl zur Anpassung von EGO an ein anderes Datenbankmanagementsystem als auch zum Vergleich unterschiedlicher Optimierungsstrategien reduziert werden kann. Ein typisches Einsatzgebiet für Primitive stellen zudem Zugriffe auf den Systemkatalog eines konkreten Datenbankmanagementsystems dar.

Im Vorfeld einer Rückgabeanweisung können allerdings noch weitere Anweisungen in beliebiger Kombination auftreten.

```
<statement>  ::=  <callRuleOrPrimitiveStatement> |
                  IF <condition> THEN <statementList> <optElse> ENDIF |
                  <pattern> <assignment> <callRuleOrPrimitiveStatement> |
                  <pattern> <assignment> STRING |
                  <pattern> <assignment> <expressionOrType> |
                  <pattern> <assignment> <collection> |
                  FORALL <pattern> IN <collection> DO <statementList> ENDFOR |
                  FORALL <pattern> IN STRING DO <statementList> ENDFOR
```

Neben dem Aufruf anderer Regeln oder Primitive werden auch bedingte Verzweigungen, Zuweisungen und Iterationen über die Elemente einer Kollektion ermöglicht.

Nachdem die grundlegende Struktur vorgestellt worden ist, werden nun drei wesentliche Bestandteile der Optimierungsregeln näher beleuchtet: Kollektionen, Eigenschaften von Algebraausdrücken sowie Bedingungen in bedingten Verzweigungen.

Kollektionen können im Wesentlichen als doppelt verkettete lineare Listen [OW96, 1.5.2] mit zusätzlicher Funktionalität aufgefasst werden. Neben dem Einfügen von Elementen ist es möglich, Kollektionen zu vereinigen, zu schneiden, enthaltene Duplikate zu entfernen und die Differenz zwischen zwei Kollektionen zu bilden. Zudem bieten sich weitere Möglichkeiten an, weil die Elemente einer Kollektion gemäß ihrer Einfügereihenfolge sortiert vorliegen. So können die ersten oder letzten n Elemente einer Kollektion gelöscht oder in Form einer neuen Kollektion zurückgeliefert werden. Für Algebraausdrücke besteht zudem die Möglichkeit, die ersten oder letzten n Elemente bezüglich einer anderen Ordnung durch Angabe einer entsprechenden Eigenschaft zu ermitteln. Dadurch können beispielsweise Kostenmodelle im Optimierungsprozess verwendet werden, indem die Kosten eines Ausführungsplans als Eigenschaftswert des ihn repräsentierenden Algebraausdrucks aufgefasst werden.

Diese Berechung von Eigenschaftswerten stellt eine wesentliche Funktionalität von Algebraausdrücken dar. Der Systemprogrammierer ist dafür zuständig, die verwendeten Eigenschaften für alle Algebraausdrücke bereitzustellen. In Optimierungsregeln kann ein Eigenschaftswert durch den Aufruf der Methode `getProperty` für einen konkreten Algebraausdruck bestimmt werden. Zudem muss gewährleistet sein, dass alle Eigenschaftswerte eine äquivalente textuelle Darstellung besitzen. Eine gegebenenfalls auf dem ursprünglichen Wertebereich existierende Ordnung darf in dieser äquivalenten Repräsentation nicht zunichte gemacht werden. Dadurch wird es möglich, konstante Eigenschaftswerte als Zeichenketten in Regeldefinitionen anzugeben, die sich mit den Eigenschaftswerten eines Algebraausdrucks vergleichen lassen. Beispielsweise könnte so eine Obergrenze für die erlaubten assoziierten Kosten eines Ausführungsplans angegeben werden. Zudem können die ersten oder letzten n Algebraausdrücke einer Kollektion wesentlich schneller durch textuelle Vergleiche ermittelt werden, als wenn für jeden Vergleich der Eigenschaftswert neu berechnet oder zumindest ein komplexer Vergleich von Objekteigenschaften stattfinden müsste. Die äquivalente textuelle Darstellung der Eigenschaftswerte wird also zudem für derartige Vergleiche innerhalb von EGO zwischengespeichert.

Bedingungen in bedingten Verzweigungen werden schließlich durch folgende Produktionen beschrieben:

Beispiel 5.1 Eine Optimierungsregel zur Vertauschung von Operanden

```
RULE
exchangeOperandsInBinExpression: Expr
{ leftOperand operator rightOperand }: Expr;
if (operator.getProperty(commutative)) then
   return { rightOperand operator leftOperand };
endif;
```

```
<condition>  ::=  TRUE | FALSE | <unaryBoolOperator> <condition> |
                  '(' <condition> [ <binaryBoolOperator> <condition> ] ')' |
                  [ <pattern> <assignment> ] <callRuleOrPrimitiveStatement> |
                  <collection> '.' <collectionBoolMethod> |
                  STRING '.' <collectionBoolMethod> |
                  <property> [ '.' <comparison> '(' <property> ')' |
                                               '.' <comparison> '(' STRING ')' ] |
                  STRING '.' ISEQUAL '(' <expressionOrType> ')'
```

Es können also neben den booleschen Literalen `TRUE` und `FALSE` Verknüpfungen mit Hilfe boolescher Operatoren zwischen existierenden Bedingungen gebildet werden. Zusätzlich können bereits hier andere Regeln oder Primitive aufgerufen werden. Genau dann, wenn dieser Aufruf ein Ergebnis liefert und – falls angegeben – auf das entsprechende Muster passt, gilt sein Wert im Rahmen dieser Bedingung als `TRUE` und wird – sofern vorhanden – an den im Muster enthaltenen Bezeichner gebunden. Dieser Bedingungstyp findet sich mit geringerer Mächtigkeit bereits als Regelart in COKO-KOLA [CZ98a]; weil sich diese Regelart dort als nützlich erweist, wurde sie als Bedingungstyp auch in EGO aufgenommen. Weiterhin kann als Teil einer Bedingung eine Kollektions-Methode aufgerufen werden, die einen booleschen Wert als Ergebnis liefert. Neben der ausführlichen Angabe einer Kollektion kann hierzu auch der Bezeichner einer Kollektion verwendet werden. Schließlich ist es möglich, Algebraausdrücke, Algebratypen oder Eigenschaftswerte von Algebraausdrücken miteinander zu vergleichen.

Beispiele für Optimierungsregeln

Nachdem bisher nur die allgemeine Struktur der Optimierungsregeln in EGO betrachtet worden ist, sollen nun einige Beispiele die Ausdruckskraft der Regeln verdeutlichen.

Die in Beispiel 5.1 gezeigte Optimierungsregel `exchangeOperandsInBinExpression` liefert bei erfolgreicher Anwendung einen Algebraausdruck als Ergebnis zurück. Sie ist nicht als geschützt gekennzeichnet, sodass sie auch ohne Angabe ihres Namens, beispielsweise als Element einer Kollektion, zur Anwendung kommen kann. Als Muster dient ein Algebraausdruck, der aus einem Operator mit zwei Operanden besteht. Wird die Regel mit einem entsprechenden Algebraausdruck als Parameterwert aufgerufen, so kommen die weiteren Anweisungen zum Tragen.

Als erstes wird hier über die Analyse des Eigenschaftswerts `commutative` des übergebenen Operators überprüft, ob dieser eine Vertauschung seiner Operanden zulässt. Ist dieses der Fall, wird ein neuer Algebraausdruck zurückgeliefert, der eben aus dieser Vertauschung entsteht, und die Optimierungsregel gilt als erfolgreich angewendet. Anderenfalls wird keine Rückgabe erzeugt; die Anweisungsliste ist beendet und die Regelausführung gilt somit als fehlgeschlagen.

Beispiel 5.2 zeigt die Optimierungsregel `applyRuleForAllMembersOfCollection`, die eine weitere Regel auf Algebraausdrücke anwendet und das Ergebnis, eine Kollektion von Algebraausdrücken, zurückliefert. Diese Regel ist durch das Schlüsselwort `PROTECTED` geschützt, daher kann sie nur über die Angabe ihres Namens aufgerufen werden. Durch die Musterliste wird festgelegt, dass die Parameterwerte beim Aufruf der Regel eine Kollektion `c` von Algebraausdrücken und eine weitere Regel `r` darstellen müssen. Werden entsprechende Argumente bereitgestellt, beginnt die weitere Ausführung der Regel. Hier wird zuerst eine neue leere Kollektion von Algebraausdrücken erzeugt, bevor in der folgenden Schleife über alle Elemente der übergebenen Kollection `c` iteriert wird. Jeder dieser Algebraausdrücke wird dazu an den Bezeichner `e` gebunden. Anschließend wird die Regel `r` für diesen Algebraausdruck ausgeführt. Wenn sie ein Ergebnis liefert, dann wird dieses an den Bezeichner `newE` gebunden und ihre Ausführung gilt als erfolgreich, sodass dieses Ergebnis in die Kollektion `newCollection` eingefügt wird. Weil alle Operationen auf Kollektionen die Ursprungskollektion unverändert lassen und stattdessen eine veränderte Kollektion als Rückgabe bereitstellen, muss das Ergebnis dieser Operation erneut an den gleichen Bezeichner gebunden werden. Diese

Beispiel 5.2 Eine Optimierungsregel zur Anwendung einer Regel auf Elemente einer Kollektion

```
RULE PROTECTED
applyRuleForAllMembersOfCollection: Coll<Expr>
c: Coll<Expr> ~ r: Rule;
newCollection: Coll<Expr> = [];
forall e: Expr in c do
   if (newE : Expr = r(e)) then
      newCollection : Coll<Expr> = newCollection.insert(newE);
   endif;
endfor;
return newCollection;
```

Anweisung stellt also eine Zuweisung dar, sodass auch der Typ dieser neuen Kollektion `newCollection` angegeben werden muss, obwohl zuvor eine andere Kollektion unter diesem Namen existiert hat. Nachdem über alle Algebraausdrücke `e` der ursprünglichen Kollektion `c` iteriert worden ist, liefert die letzte Anweisung der Regel das Ergebnis in Form von `newCollection` zurück.

Diese Regel lässt sich leicht derart verändern, dass nicht nur eine einzige Regel, sondern mehrere Regeln auf jeden in der übergebenen Kollektion enthaltenen Algebraausdruck angewendet werden. Dazu muss die Musterliste so verändert werden, dass sie als zweites Argument keine Regel, sondern eine Kollektion von Regeln erwartet. Weiterhin wird die bestehende Schleife in eine zusätzliche eingebettet, die über alle Regeln dieser Kollektion iteriert. Unter Zuhilfenahme der Kollektions-Operation `get` können so Optimierungsansätze – wie sie beispielsweise durch Paul [Pau97] im Rahmen des CROQUE-Projekts [HSGK] verfolgt werden – durch Optimierungsregeln dargestellt werden, die aus einer Kollektion von Regeln nur eine bestimmte Anzahl auswählen. Kröger *et. al.* [KPH98] liefern eine weitergehende Einführung in dieses Konzept und einen Vergleich der Unterstützung der Anordnung und Auswahl bestimmer Optimierungsregeln durch einige bereits in Abschnitt 5.1 vorgstellte Komponenten zur Anfrageoptimierung.

Auch anhand der Transformationsregeln in COKO-KOLA [CZ98a] lässt sich die Funktionalität der Optimierungsregeln in EGO näher beleuchten. In COKO-KOLA existieren vier unterschiedliche – und zwar fest vorgegebene und nicht erweiterbare – Algorithmen zur Anwendung – Cherniack und Zdonik sprechen von *firing* oder *Feuern* – der dortigen Transformationsregeln. Die Optimierungsregeln in EGO sind mächtig genug, um die dort realisierte Funktionalität auszudrücken. Durch die Formulierung dieser Algorithmen durch Optimierungsregeln wird zudem eine leichtere Anpassbarkeit und Erweiterung der entsprechenden Verfahren gewährleistet. Die Verfahren und eine mögliche Darstellung durch Optimierungsregeln in EGO werden im Folgenden kurz erläutert:

Conditional Firing: Eine durch das Ergebnis vorheriger Regelausführungen bedingte Regelanwendung kann in Optimierungsregeln durch bedingte Verzweigungen und dort auftretende Regelanwendungen dargestellt werden – siehe etwa `exchangeOperandsInBinExpression` in Beispiel 5.1.

Explicit Firing: Die explizite Regelanwendung entspricht der Ausführung von Optimierungsregeln durch Angabe ihres Namens. Transformationsregel können in COKO-KOLA bei expliziter Anwendung grundsätzlich sowohl von vorne nach hinten als auch von hinten nach vorne gelesen und ausgeführt werden. Es existiert also zu jeder Transoformationsregel ein inverses Gegenstück. EGO unterstützt jedoch keine bidirektionalen Transformationen von Algebraausdrücken, da die eingesetzten Optimierungsregeln dafür zu komplex sind. Durch die Formulierung einer zweiten Optimierungsregel kann dieses Verhalten allerdings simuliert werden. Der bei der expliziten Regelanwendung in COKO-KOLA zurückgelieferte implizite boolesche Wert als weitere Aussage über den Erfolg einer Regelanwendung führte zur Bereitstellung eben dieser Eigenschaft in den Optimierungsregeln von EGO – siehe etwa `applyRuleForAllMembersOfCollection` in Beispiel 5.2.

Selective Firing: Die selektive Ausführung von Regeln dient vor allem der Überprüfung gegebener Algebraausdrücke auf Muster und dem Binden von Teilausdrücken an Bezeichner. Eine Regel in COKO-KOLA wird nur ausgeführt, falls ein Algebraausdruck dem Muster entspricht. In diesem Fall können Teile des Algebrausdrucks an im Muster auftretende Bezeichner gebunden werden, sodass sie innerhalb der Regel unter diesem Bezeichner verfügbar sind. Optimierungsregeln in EGO haben diese Fähigkeiten nicht nur im Rahmen der initialen Musterliste, stattdessen existieren Anweisungen, die eine Überprüfung der Algebraausdrücke auf bestimmte Muster auch während der Ausführung von Regeln durch den Einsatz von Zuweisungen erlauben – siehe etwa die in Beispiel 5.2 gezeigte Regel

`applyRuleForAllMembersOfCollection`. Dort wird zum Beispiel dem Bezeichner `newE` das Ergebnis der Regelanwendung `r(e)` zugewiesen. Allerdings ist diese Zuweisung nur erfolgreich, falls es sich um einen Algebraausdruck handelt. Beispiel 5.1 zeigt eine Regel, deren initiale Musterliste für das Binden der Teile eines Algebraausdrucks an entsprechende Bezeichner sorgt, die im weiteren Verlauf der Regel erneut verwendet werden.

Traversal Firing: COKO-KOLA unterstützt die Traversierung von Algebraausdrücken – in einer baumartigen Repräsentation – durch eingebaute Primitive. Optimierungsregeln in EGO können diese Traversierung direkt ausdrücken, ohne Primitive zu verwenden und sind daher auch in diesem Punkt flexibler als COKO-KOLA – siehe etwa `BU` in Beispiel 5.3.

Weiterhin zeigt Beispiel 5.3 eine Gruppe von Regeln, die das von Cherniack und Zdonik präsentierte Beispiel des *CNF-Query-Rewrite* in COKO-KOLA [CZ98a] durch Optimierungsregeln in EGO ausdrückt.

Die Regeln `d1` und `d2` dienen der eigentlichen Transformation von Algebraausdrücken und stellen nur eine alternative Formulierung der ursprünglichen Regeln ohne erweiterte Funktionalität dar. Sie transformieren einen gegebenen Algebraausdruck so, dass er in CNF (**C**onjunctive **N**ormal **F**orm) [Bri01, 1.2] überführt wird, also aus Konjunktionen von Disjunktionen der auftretenden Literale besteht. Sind allerdings die auftretenden Operanden keine einfachen Literale, so müssen auch diese eventuell umgeformt werden.

Aus diesem Grund ist eine Traversierung der Baum-Repräsentation des Algebraausdrucks erforderlich, die bei den Blättern beginnt und an der Wurzel endet. Die Regel `BU` übernimmt gerade diese Aufgabe, indem sie eine gegebene Regel zuerst auf die Kinder `child1` und `child2` des aktuellen Knotens `node` und dann auf den gesamten aktualisierten Teilbaum einschließlich des Knotens anwendet. Im Gegensatz zur Realisierung in COKO-KOLA lässt sich `BU` durch eine einzige Optimierungsregel ausdrücken und muss nicht als Primitiv realisiert werden.

Die Optimierungsregel `CNFAux` ist ähnlich der entsprechenden Regel in COKO-KOLA aufgebaut. Allerdings wird hier im Gegensatz zur urspünglichen Regel genau untersucht, welche der Regeln `d1` oder `d2` erfolgreich angewendet werden konnte. Im Fall einer erfolgreichen Anwendung werden erneut die Operanden untersucht, bevor der aktualisierte Ausdruck zurückgegeben wird. Konnte weder `d1` noch `d2` erfolgreich angewendet werden, reicht es, den ursprünglichen Ausdruck unverändert als Ergebnis bereitzustellen. Die Optimierungsregel `CNFAux` überprüft zu Beginn ihrer Anwendung, ob eine erfolgreiche Regelanwendung von `d1` für den gegebenen Algebraausdruck `booleanExpr` möglich ist. Sollte dieses der Fall sein, wird die Rückgabe an den Bezeichner `newExpr1` gebunden und im Folgenden durch eine Zuweisung auf die drei Bezeichner `left1`, `mid1` und `right1` aufgeteilt. Anschließend wird die aktuelle Regel auf den rechten und den linken Teilausdruck angewendet, bevor der durch Verwendung der zugehörigen Ergebnisse entstehende Algebraausdruck zurückgeliefert wird. Der Ablauf für eine Anwendung der Regel `d2` erfolgt analog. Insgesamt entspricht diese Regel `CNFAux` der gleichnamigen Regel in der Beschreibung von Cherniack und Zdonik [CZ98a, 2.3.2].

Für die Aktivierung des zuvor beschriebenen Umformungsprozesses ist die Optimierungsregel `CNF` zuständig. Sie ruft die Regel `BU` für einen gegebenen Algebraausdruck `booleanExpr` und die Regel `CNFAux` auf und liefert das Ergebnis zurück. Die Optimierungsregel `BU` sorgt wie oben beschrieben dafür, dass eine Traversierung der Baumrepräsentation des Algebraausdrucks erfolgt, wobei in jedem Schritt die Regel `CNFAux` angewendet wird.

Weitere Beispiele für die Verwendung der Optimierungsregeln in EGO finden sich im Rahmen der Beschreibung der in GOODAC verwendeten Optimierungsstrategie in Abschnitt 5.3.2. Die volle Ausdruckskraft und Funktionalität der Regeln kann sich aber erst bei einem Vergleich unterschiedlicher Optimierungsstrategien und der Visualisierung des Einflusses von Änderungen der eingesetzten Optimierungsregeln zeigen. Bisher ist aber schon deutlich geworden, dass die hier vorgestellten Optimierungsregeln eine wesentlich größere Funktionalität und damit auch deutlich mehr Möglichkeiten im Rahmen des Optimierungsprozesses bieten als die Transformationsregeln anderer regelbasierter Ansätze zur Anfrageoptimierung.

Konvertierung von Algebraausdrücken

Einen wesentlichen Gesichtspunkt hinsichtlich der Unabhängigkeit von einem konkreten Datenbankmanagementsystem und der Erweiterbarkeit der Optimierunsgkomponente EGO stellt die Behandlung von Algebraausdrücken und Algebratypen dar. Wie bereits durch die Produktion

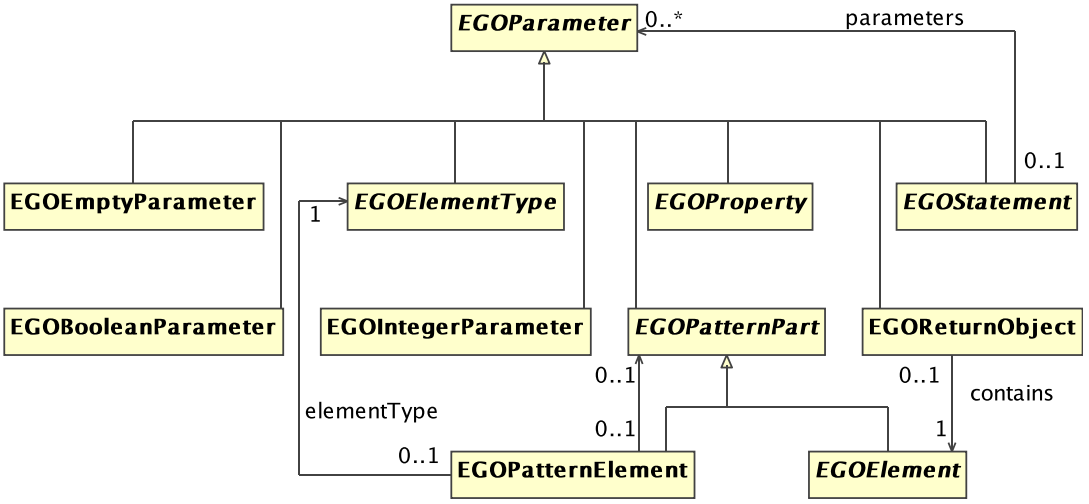
‹expressionOrType› ::= { ALGEBRASTRING }

Beispiel 5.3 Formulierung des *COKO-KOLA-CNF-Query-Rewrite* [CZ98a] durch Optimierungsregeln

```
GROUP CNF_REWRITE

RULE PROTECTED
d1: Expr
{ ( p and q ) or r } : Expr;
return { ( p or r ) and ( q or r ) };

RULE PROTECTED
d2: Expr
{r or ( p and q )}: Expr;
return {( r or p ) and ( r or q )};

RULE PROTECTED
BU: Expr
{child1 node child2}: Expr ~ aRule: Rule;
newChild1: Expr = BU(child1);
newChild2: Expr = BU(child2);
return aRule({newChild1 node newChild2});

RULE PROTECTED
CNFAux: Expr
booleanExpr: Expr;
if (newExpr1: Expr = d1(booleanExpr)) then
   {left1 mid1 right1}: Expr = newExpr1;
   newLeft1 : Expr = CNFAux(left1);
   newRight1 : Expr = CNFAux(right1);
   return {newLeft1 mid1 newRight1};
else
   if (newExpr2: Expr = d2(booleanExpr)) then
      {left2 mid2 right2}: Expr = newExpr2;
      newLeft2 : Expr = CNFAux(left2);
      newRight2 : Expr = CNFAux(right2);
      return {newLeft2 mid2 newRight2};
   else
      return booleanExpr;
   endif;
endif;

RULE PROTECTED
CNF: Expr
booleanExpr: Expr;
return BU(booleanExpr ~ CNFAux);

ENDGROUP
```

deutlich wird, werden Algebraausdrücke und Algebratypen – eingeschlossen in geschweifte Klammern – nicht von EGO analysiert. Stattdessen erfolgt die Verarbeitung durch eine Algebra-spezifische Komponente, die ein Systemprogrammierer oder ein Entwickler des zugehörigen Datenbankmanagementsystems bereitstellen muss. Diese Komponente konvertiert die textuelle Beschreibung dieser Algebraausdrücke und Algebratypen in ein internes generisches objektbasiertes Format im Rahmen von EGO und gewährleistet die Einhaltung bestimmter Voraussetzungen, etwa zur Überprüfung auf Übereinstimmung mit einem vorgegebenen Muster. Um also EGO an ein konkretes Datenbankmanagementsystem anzubinden, muss ein Systemprogrammierer neben der erforderlichen textuellen Beschreibung der Optimierungsregeln auch eine Komponente zur Konvertierung von Algebraausdrücken entwickeln. Der Optimierungsprozess wird von EGO ausgehend von diesen Optimierungsregeln und den darin enthaltenen Anweisungen unter Verwendung der in das interne Format konvertierten Algebraausdrücke gesteuert.

Abbildung 5.3 Grundlegende Klassenhierarchie zur Realisierung von EGO



Eine genauere Darstellung dieses Konzepts findet sich zum einen im Rahmen der Präsentation der internen Realisierung von EGO – siehe vor allem die Beschreibung übergreifender Klassen auf Seite 101 in Abschnitt 5.2.2 – und zum anderen bei der Beschreibung der Einbindung von EGO in den Prozess der Anfragebearbeitung in GOODAC (Abschnitt 5.3.1).

5.2.2 Interne Realisierung

In diesem Abschnitt wird die interne Realisierung von EGO vorgestellt. Dabei wird jeweils auf die zugehörigen Bestandteile der zuvor beschriebenen Struktur der textuellen Darstellung von Optimierungsregeln eingegangen, sodass in diesem Abschnitt für jeden Bestandteil der Optimierungsregeln eine entsprechende interne Repräsentation gezeigt wird. Die dargestellten Klassen besitzen größtenteils den Präfix `EGO`, um zu verdeutlichen, dass es sich bei ihnen um einen Bestandteil der Komponente zur erweiterbaren und generischen Anfrageoptimierung handelt. Bei der Darstellung und Beschreibung der einzelnen Methoden wird aus Übersichtlichkeitsgründen größtenteils auf die Präsentation der zu verwendenden Parametertypen verzichtet.

Basisklassen

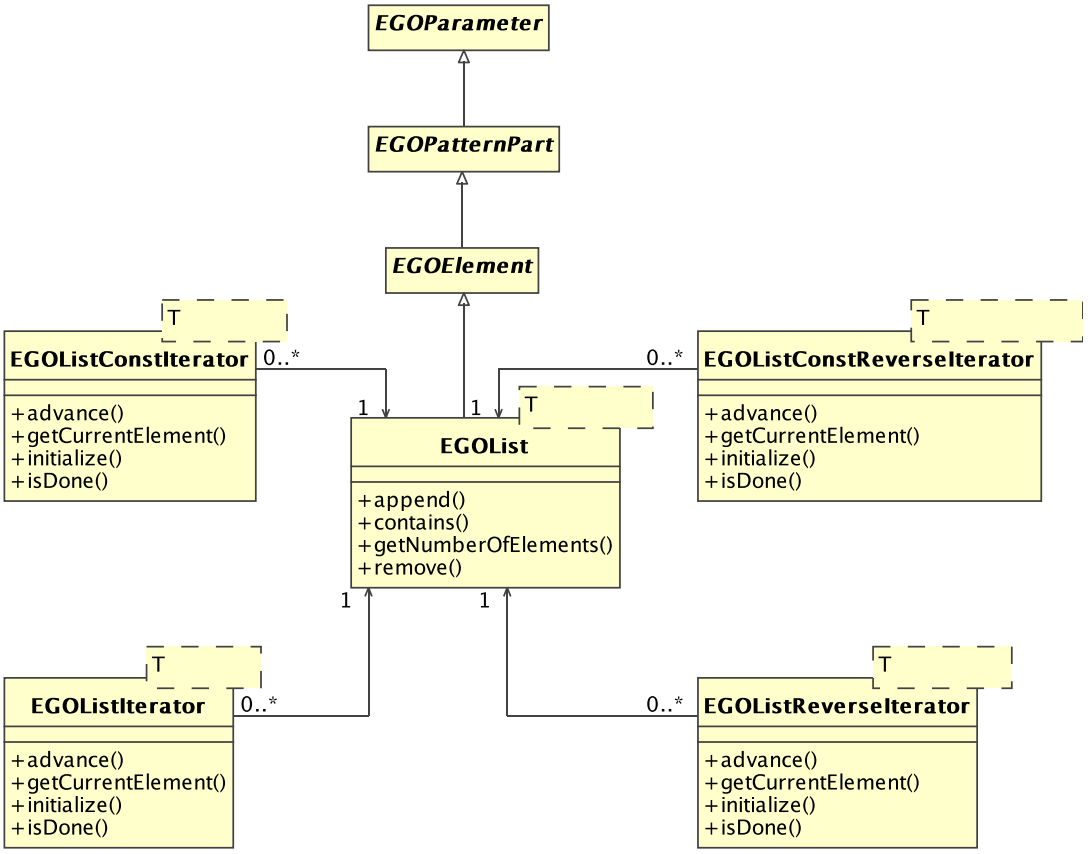
Abbildung 5.3 zeigt den obersten Teil der Klassenhierarchie in EGO. Wie zuvor erwähnt, werden die enthaltenen Attribute und Methoden nicht näher beschrieben. Die abstrakte Klasse `EGOParameter` dient als Basis für die meisten in EGO enthaltenen Klassen. Dadurch wird gewährleistet, dass nahezu alle auftretenden Objekte als Argumentwert – etwa für die weiter unten beschriebenen Anweisungsobjeke – verwendet werden können.

Auch einige der übrigen Klassen haben einen eher übergeordneten Charakter. So dienen Unterklassen von `EGOElementType` der Unterscheidung möglicher Elementtypen (siehe auch Abbildung 5.5), während Unterklassen zu `EGOStatements` einzelne Anweisungen aus Optimierungsregeln repräsentieren (siehe auch Abbildung 5.7). Die Klasse `EGOProperty` dient als Basis zur internen Darstellung der Eigenschaften von Algebraausdrücken. Zur Repräsentation einer konkreten Eigenschaft muss ein Systemprogrammierer entsprechende Klassen von `EGOProperty` ableiten, bevor diese im Rahmen des Optimierungsprozesses Einsatz finden können.

Die Klassen `EGOBooleanParameter` und `EGOIntegerParameter` dienen der Handhabung boolescher und ganzzahliger Argumentwerte im Rahmen von EGO. Weiterhin kann durch die Verwendung einer Instanz der Klasse `EGOEmptyParameter` ausgedrückt werden, dass kein konkreter Parameterwert vorhanden ist. Im weiteren Verlauf (siehe auch die Beschreibung von Abbildung 5.5) wird noch eine Klasse `EGOStringParameter` präsentiert, die jedoch an anderer Stelle in der Klassenhierarchie angesiedelt ist, weil ihre Instanzen der Darstellung eines der fünf in EGO zugelassenen Elementtypen in EGO (siehe auch Seite 84) dienen.

Ein Objekt vom Typ `EGOReturnObject` repräsentiert den Rückgabewert einer Regelanwendung. Da – wie bereits oben im Zusammenhang mit Bedingungen für bedingte Verzweigungen beschrieben wurde –

Abbildung 5.4 Listen in EGO



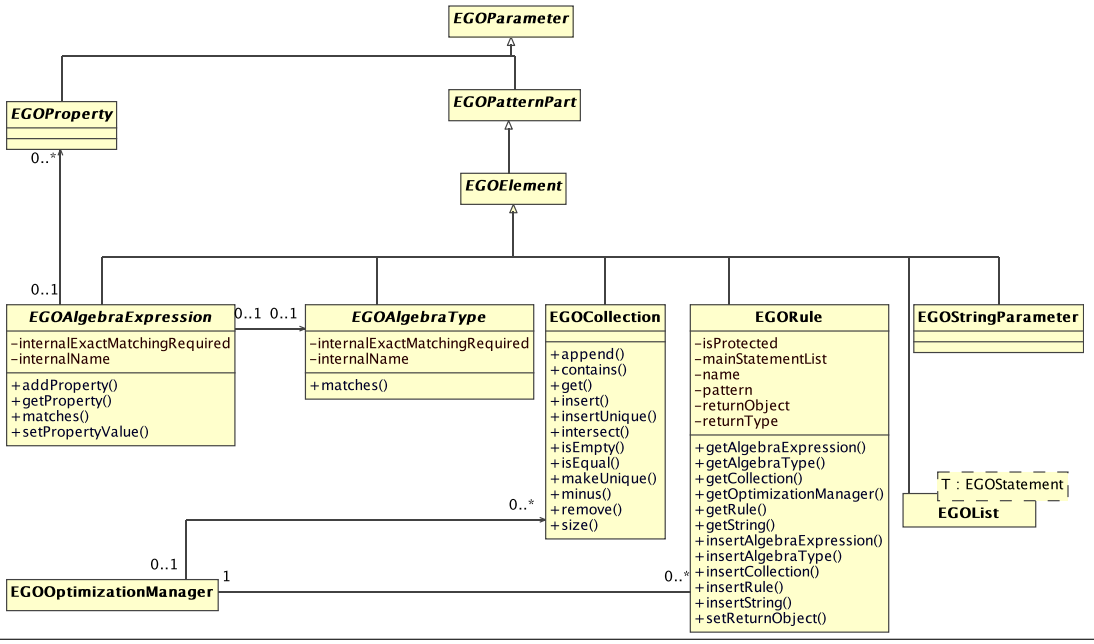
im Allgemeinen nicht nur das Ergebnis einer Regelanwendung, sondern auch Informationen über das Erreichen einer Rückgabeanweisung innerhalb dieser Regelanwendung von Interesse sind, kapselt ein derartiges Objekt beide Informationen. Abhängig vom Aufrufkontext der zuvor ausgeführten Optimierungsregel kann anschließend entschieden werden, welche der verfügbaren Informationen benötigt wird. Wird also zum Beispiel eine Optimierungsregel innerhalb der Bedingung einer bedingten Verzweigung aufgerufen, wird die Information über das Erreichen einer Rückgabeanweisung aus der zurückgelieferten Instanz der Klasse `EGOReturnObject` ausgelesen. Wird das Resultat der Regelanwendung etwa sofort als Parameter zum Aufruf einer weiteren Optimierungsregel verwendet, muss dieses eigentliche Ergebnis der ursprünglichen Regelanwendung aus der erhaltenen Instanz der Klasse `EGOReturnObject` extrahiert werden.

Die verbleibenden drei Klassen dienen der internen Realisierung von Musterlisten in Regeln. Musterlisten können im Allgemeinen aus mehreren Teilen bestehen, die durch das Tildesymbol (~) voneinander abgegrenzt werden. Eine Instanz einer Unterklasse zur abstrakten Klasse `EGOPatternPart` dient der Darstellung eines derartigen Musterelements. Dabei kann es sich einerseits um einen einfachen Bezeichner handeln, der ein beliebiges Element eines bestimmten Typ repräsentiert. Derartige Teile einer Musterliste lassen sich durch Instanzen der Klasse `EGOPatternElement` darstellen, wobei die in Abbildung 5.3 gezeigte Assoziation `elementType` die Angabe des Typs dieses Elements realisiert. Andererseits können die Teile einer Musterliste innerhalb einer Optimierungsregel auch näher spezifiziert werden, indem neben dem Typ kein Bezeichner, sondern ein konkretes Element über die zweite gezeigte Assoziation angegeben wird. Diese Teile einer Musterliste werden erneut durch Objekte mit Oberklasse `EGOPatternPart` innerhalb von EGO dargestellt. Dabei handelt es sich um konkrete Elemente – beispielsweise Algebraausdrücke oder Kollektionen – mit Oberklasse `EGOElement`, die ebenfalls abseits der Muster in Optimierungsregeln auftreten – etwa bei der Rückgabe des Ergebnisses einer Regelanwendung innerhalb einer Instanz vom Typ `EGOReturnObject`.

Listen

Um innerhalb der Optimierungskomponente problemlos Listen von Objekten verwalten und als Argument übergeben zu können, existiert die parametrisierte Klasse `EGOList<T>` (siehe auch Abbildung 5.4). Konkrete

Abbildung 5.5 Klassen für die verschiedenen Elementarten im Rahmen von Optimierungsregeln in EGO



Instanzen dieser Klasse unterstützen alle wesentlichen Operationen für geordnete Listen mit Elementen vom Typ `T`, insbesondere das Anhängen und Löschen von Elementen sowie die Abfrage der Elementzahl oder die Überprüfung auf das Vorkommen eines Elements. Die ebenfalls in Abbildung 5.4 gezeigten Iterator-Klassen ermöglichen einen sequentiellen Zugriff auf die Listenelemente und folgen im Wesentlichen dem von Stroustrup vorgestellten Iterator-Konzept [Str98, 19].

Weil Instanzen der Klasse `EGOList<T>` als Argumentwerte im Rahmen der internen Realisierung von EGO auftauchen, muss `EGOList<T>` in der Klassenhierarchie unterhalb von `EGOParameter` angeordnet werden. Weiterhin ist es möglich, dass mehrere Optimierungsregeln den gleichen Namen, allerdings unterschiedliche Musterlisten besitzen. Bei einem Aufruf über den Regelnamen wird in diesem Fall die erste Regel ausgewählt, auf deren Musterliste die übergebenen Argumente passen. Daher tauchen intern nicht nur einzelne Optimierungsregeln und Kollektionen von Optimierungsregeln, sondern auch Listen von Optimierungsregeln auf. Aus diesem Grund müssen derartige Listen auf die gleiche Art und Weise wie andere Elemente der Optimierungsregeln – beispielsweise Algebraausdrücke oder Kollektionen – verwendet werden können; also findet sich die Klasse `EGOList<T>` als direkte Unterklasse zu `EGOElement` innerhalb von EGO wieder.

Listen und Kollektionen dienen beide der Verwaltung anderer Elemente. Allerdings wird trotzdem strikt zwischen beiden unterschieden, weil Kollektionen in Optimierungsregeln und ihrer textuellen Beschreibung auftreten, sodass ein Systemprogrammierer ihre Funktionalität kennen muss, während Listen nur der internen Verwaltung von Elementen in EGO dienen. Daraus lassen sich unterschiedliche Aufgaben von Listen und Kollektionen ableiten, sodass die Verwendung zweier verschiedener Klassen für die Verwaltung von Elementen in EGO sinnvoll erscheint.

Elemente

Die in Optimierungsregeln auftretenden Elemente werden wie schon oben erwähnt durch Unterklassen zu `EGOElements` modelliert. Neben der zuvor beschriebenen Klasse `EGOList<T>` existieren fünf weitere Elementklassen zur internen Darstellung von Algebraausdrücken, Algebratypen, Kollektionen, Optimierungsregeln und Zeichenketten.

Zeichenketten werden durch Objekte vom Typ `EGOStringParameter` dargestellt. Ihre Verwendung als Element in Optimierungsregeln dient vor allem der Repräsentation von Eigenschaftswerten, dem Test von Optimierungsansätzen und der direkten Visualisierung einfacher Strategien. So kann ein Systemprogrammierer zum Beispiel textuelle Optimierungsregeln entwerfen, die eine vereinfachte Form der geplanten Optimierungsstrategie realisieren und auf Zeichenketten anstatt auf Algebraausdrücken arbeiten. Somit kann er sich mit dem Konzept von EGO vertraut machen und erste Ideen ausprobieren, ohne dass er bereits eine Komponente zur Konvertierung von Algebraausdrücken in das interne Format von EGO bereitstellen muss.

Kommt es bei der Ausführung dieser Regeln zu unerwarteten Ergebnissen, kann die Fehlersuche auf die formulierten Optimierungsregeln sowie die Implementation von EGO selbst beschränkt werden. Daneben kommen Instanzen der Klasse `EGOStringParameter` auch als Argumentwerte im Rahmen des von EGO gesteuerten Optimierungsprozesses zum Einsatz.

Eine gewöhnliche an ein konkretes Datenbankmanagementsystem gestellte Anfrage lässt sich optimieren, indem ihre Datenbank-spezifische algebraische Repräsentation durch einen generischen Algebraausdruck – also eine Instanz mit Oberklasse `EGOAlgebraExpression` – dargestellt wird. Derartige Objekte besitzen neben einem Namen im Wesentlichen zwei besondere Fähigkeiten. Zum einen können sie bestimmte Eigenschaftsobjekte assoziieren, um die schon oben erwähnten Eigenschaften von Algebraausdrücken zu realisieren. Beispielsweise könnte so ausgedrückt werden, dass der so dargestellte Algebraausdruck eine Sortierung des repräsentierten Anfrageergebnisses gewährleistet. Derartige Eigenschaften werden intern durch Instanzen einer Unterklasse der Klasse `EGOProperty` (siehe auch die Beschreibung der Basisklassen auf Seite 91) dargestellt. Zum anderen können generische Algebraausdrücke auf Übereinstimmung mit einem Muster getestet werden, wobei Muster, die Algebraausdrücke repräsentieren, in EGO ebenfalls durch Instanzen mit Oberklasse `EGOAlgebraExpression` abgebildet werden. Schließlich können Algebraausdrücke im internen generischen Format von EGO noch eine Typangabe besitzen, die eine differenzierte Klassifikation der einzelnen Algebraausdrücke gewährleistet. Eine genauere Darstellung der Algebraausdrücke findet sich im Rahmen der Beschreibung von Abbildung 5.8.

Algebratypen lassen sich durch Objekte mit Oberklasse `EGOAlgebraType` darstellen. Sie können wie schon die zuvor beschriebenen Algebraausdrücke ebenfalls einen Namen besitzen und auf Übereinstimmung mit Mustern getestet werden. Eine genauere Darstellung und Unterteilung dieser Algebratypen liefert die Beschreibung von Abbildung 5.9.

Optimierungsregeln werden in EGO durch Instanzen der Klasse `EGORule` abgebildet. Diese Regelobjekte dienen vor allem als Datenspeicher zur Repräsentation des Kontextes einer Regelanwendung. Dabei finden sich neben den charakteristischen Daten einer Optimierungsregel – wie beispielsweise dem Namen oder der vor Beginn einer Regelanwendung zu überprüfenden Musterliste – vor allem Attribute und Methoden zur Verwaltung der im Rahmen einer Regelanwendung auftretenden Daten in der Klasse `EGORule` wieder. So werden etwa Ergebnisse von Zuweisungen über die nach Typen unterschiedenen Einfügemethoden in den Regelkontext übernommen, sodass sie durch Aufruf der entsprechenden Methoden mit Präfix `get` wieder ausgelesen werden können. Die in Abschnitt 5.2.1 beschriebene Funktionalität der Optimierungsregeln wird durch Instanzen zur Oberklasse `EGOStatement` realisiert, eine genaue Darstellung findet sich in den Erläuterungen zu Abbildung 5.7. Regelobjekte in EGO enthalten aus diesem Grund nur eine Anweisungsliste `mainStatementList` vom Typ `EGOList<EGOStatement>`, deren enthaltene Anweisungsobjekte den Anweisungen der repräsentierten textuellen Optimierungsregel entsprechen.

Kollektionsobjekte, also Instanzen der Klasse `EGOCollection`, sind in der Lage, alle in Optimierungsregeln auftretenden Kollektionen zu repräsentieren. Sie unterstützen alle der für Kollektionen zugelassen Operationen, sodass eine direkte Abbildung der Anwendung dieser Operationen auf Methodenaufrufe eines Kollektionsobjekts erfolgen kann. Wie bereits oben bei der Beschreibung der Listen (siehe auch Seite 92) deutlich wurde, realisieren Kollektionsobjekte die Funktionalität der in Optimierungsregeln auftretenden Kollektionen, während Listen der internen Verwaltung von Elementen dienen.

Eine weitere Anwendung finden Kollektionsobjekte bereits im Rahmen des Einlesens und Analysierens der Optimierungsregeln. Hier wird jede Gruppe von Optimierungsregeln zu einem Kollektionsobjekt umgewandelt, das die entsprechenden Regelobjekte enthält. Ein solches Kollektionsobjekt wird unter dem zugehörigen Namen der Regelgruppe in einer Instanz der Klasse `EGOOptimizationManager` gespeichert. Eine derartige Instanz ist für die übergeordnete Steuerung des Optimierungsprozesses zuständig. Zusätzlich ist jedes Regelobjekt direkt unter seinem Namen in dieser Instanz der Klasse `EGOOptimizationManager` verfügbar, sodass ein effizienter Zugriff auf bestimmte Regelobjekte möglich wird, falls eine Anweisung innerhalb einer Optimierungsregel dem Aufruf einer weiteren Optimierungsregel über die Angabe deren Regelnamens entspricht. Die Darstellung der weiteren Funktionalität der Klasse `EGOOptimizationManager` erfolgt im Rahmen der Beschreibung von Abbildung 5.10.

Elementtypen

Die in Abbildung 5.6 dargestellten Klassen dienen der Repräsentation von Elementtypen in EGO. Weil die Klasse `EGOList<T>` – wie bereits oben erwähnt – nur aus internen Gründen von `EGOElement` erbt und daher insbesondere Listen nicht als Elemente in Optimierungsregeln auftreten, reicht es aus, nur die restlichen fünf zuvor erwähnten Elementarten durch zugehörige Elementtypobjekte zu berücksichtigen. Für

Abbildung 5.6 Klassen zur Typkennzeichnung in EGO

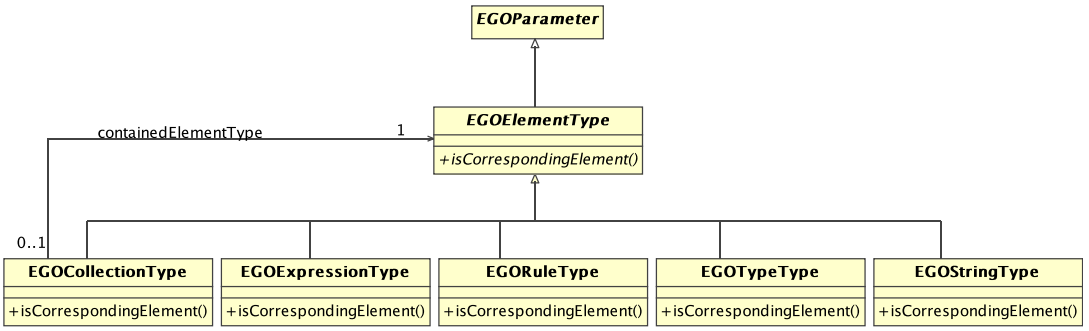
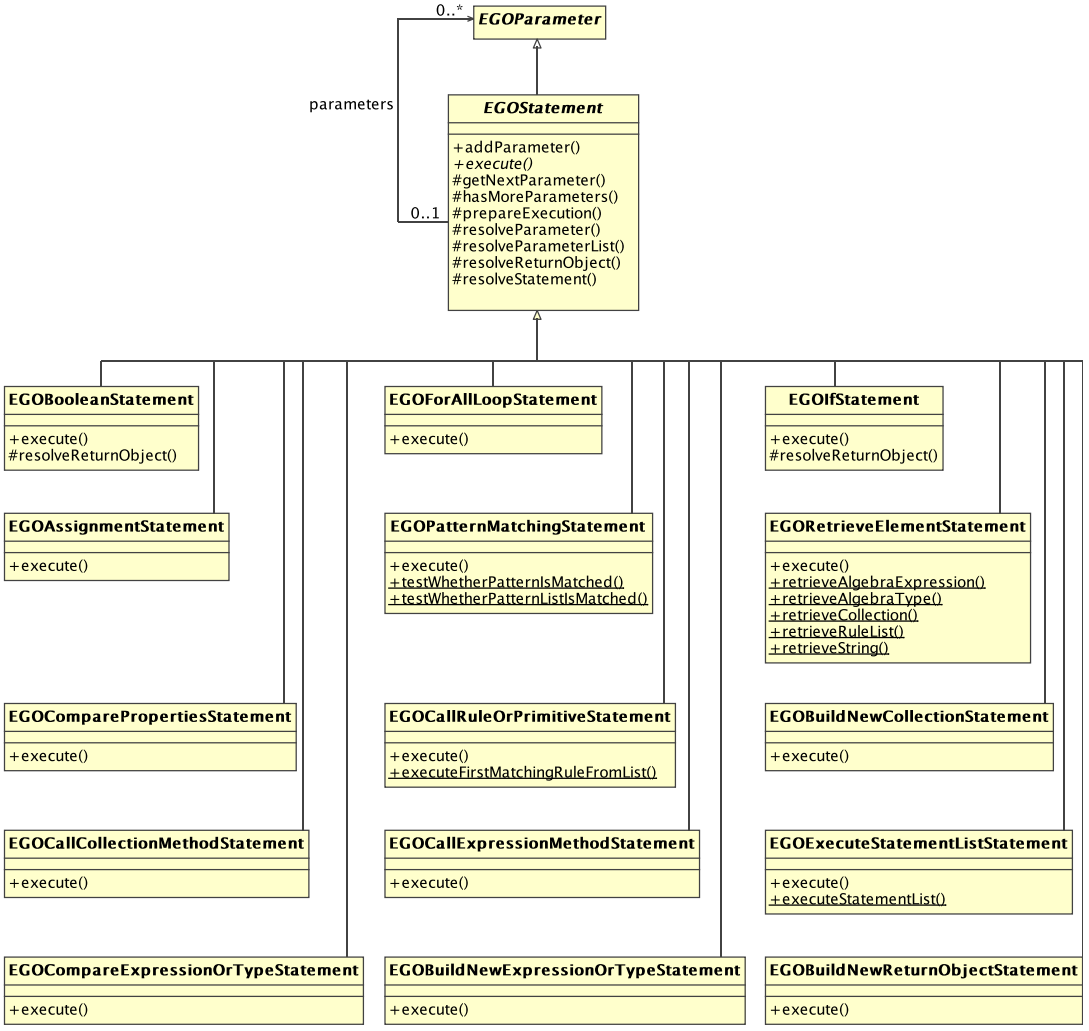


Abbildung 5.7 Klassen zur Repräsentation von Anweisungen in EGO



Instanzen der Klasse `EGOCollectionType` wird auch der Typ der in einer Kollektion enthaltenen Objekte vermerkt, um auch hier eine Typprüfung vornehmen zu können. Die durch alle Klassen realisierte Methode `isCorrespondingElement` überprüft für ein übergebenes Objekt mit Obertyp `EGOElement`, ob es den durch das aktuelle Elementtypobjekt repräsentierten Typ besitzt. Für Kollektionsobjekte wird entsprechend der Definition von `EGOCollectionType` auch der Typ aller enthaltenen Elemente betrachtet.

Anweisungen

Anweisungen in Optimierungsregeln werden intern durch Instanzen mit Oberklasse `EGOStatement` realisiert. Für jeden Anweisungstyp existiert zu diesem Zweck eine spezielle Klasse; alle derartigen Klassen

Tabelle 5.3 Aufgaben der Anweisungsklassen in EGO

Anweisungsklasse	Funktionalität
EGOAssignmentStatement	Durchführung von Zuweisungen
EGOBooleanStatement	Auswertung boolescher Ausdrücke
EGOBuildNewCollectionStatement	Erzeugung einer Kollektion
EGOBuildNewExpressionOrTypeStatement	Erzeugung eines Algebraausdrucks oder Algebratyps
EGOBuildNewReturnObjectStatement	Erzeugung eines Rückgabeobjekts
EGOCallCollectionMethodStatement	Aufruf einer Methode für ein Kollektionsobjekt
EGOCallExpressionMethodStatement	Aufruf einer Methode für ein Algebraausdrucksobjekt
EGOCallRuleOrPrimitiveStatement	Aufruf einer Regel oder eines Primitives
EGOCompareExpressionOrTypeStatement	Vergleich von Algebraausdrücken oder Algebratypen
EGOComparePropertiesStatement	Vergleich der Eigenschaftswerte von Algebraausdrücken
EGOExecuteStatementListStatement	Abarbeitung einer Anweisungsliste
EGOForAllLoopStatement	Abarbeitung einer Schleife
EGOIfStatement	Abarbeitung einer bedingten Verzweigung
EGOPatternMatchingStatement	Vergleich einer Musterliste mit Parametern
EGORetrieveElementStatement	Auslesen benannter Elemente aus dem Regelkontext

werden in Abbildung 5.7 dargestellt. Weil sich die Funktionalität dieser Klassen wesentlich auf den durch `EGOStatement` bereitgestellten Methoden abstützt, werden diese zuerst beschrieben. Anschließend erfolgt eine Darstellung der einzelnen Anweisungsklassen.

Die Klasse `EGOStatement` besitzt eine abstrakte Methode `execute`, die von allen konkreten Anweisungsklassen mit der jeweiligen Funktionalität zur Ausführung einer entsprechenden Anweisung überschrieben wird. Als Parameterwert erhält diese Methode beim Aufruf ein Regelobjekt, sodass der aktuelle Kontext während der Ausführung einer Anweisung bekannt ist. Alle weiteren Methoden der Klasse `EGOStatement` dienen der Verwaltung der zu jeder Anweisung gehörenden Parameter; so können beispielsweise während des Einlesens der Optimierungsregeln jedem Anweisungsobjekt neue Argumentwerte mittels der Methode `addParameter` übergeben werden.

Die verbleibenden Methoden zur Verwaltung der Argumente einer Anweisung kommen bei ihrer Ausführung im Rahmen des Optimierungsprozesses zum Einsatz. Die Ausführung einer Anweisung beginnt jeweils mit dem Aufruf der Methode `prepareExecution` innerhalb der Operation `execute` des entsprechenden Anweisungsobjekts. Hier werden insbesondere alle übergebenen Parameterwerte durch die Methoden `resolveParameterList` und `resolveParameter` betrachtet. Diese Untersuchung wird im Folgenden genauer dargestellt.

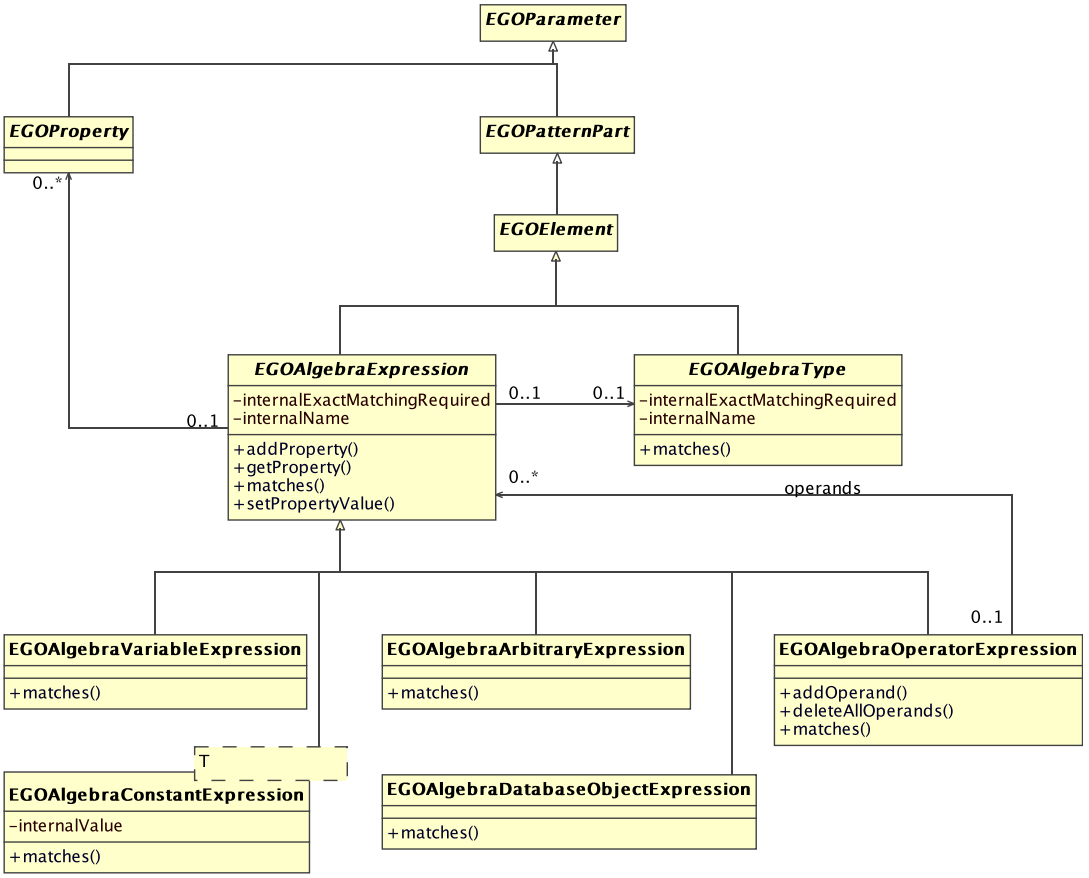
Sollten Anweisungen geschachtelt sein, sodass als Argument ein weiteres Anweisungsobjekt vorliegt, so wird zuerst diese Anweisung durch Aufruf der Methode `resolveStatement` ausgeführt, um deren Ergebnis als Parameterwert für die aktuelle Anweisung zu erhalten.

Weiterhin kann es vorkommen, dass als Parameterwert das Resultat des Aufrufs einer Optimierungsregel – also ein Objekt vom Typ `EGOReturnObject` – auftritt. Die meisten Anweisungsobjekte sind nur am eigentlichen Ergebnis – beispielsweise einem Algebraausdruck oder einer Kollektion von Algebraausdrücken – interessiert, sodass die Methode `resolveReturnObject` dafür sorgt, dass der implizite boolesche Rückgabewert ignoriert und stattdessen nur die eigentliche Rückgabe betrachtet wird. Nur zwei Klassen – `EGOBooleanStatement` und `EGOIfStatement` – überschreiben diese Methode, da ihre zugehörigen Objekte zur Erfüllung ihrer Aufgabe entscheiden müssen, ob eine Regelanwendung erfolgreich war. Dazu ist es erforderlich, dass sie den impliziten Rückgabewert dieser Regelanwendung auswerten und stattdessen ihr eigentliches Ergebnis nicht betrachten.

Eine Übersicht über die Aufgaben der bereits in Abbildung 5.7 dargestellten konkreten Anweisungsklassen liefert Tabelle 5.3. Alle dort gezeigten Klassen realisieren die in `EGOStatement` nur abstrakt definierte Methode `execute`, sodass die zugehörigen Objekte bei Aufruf dieser Methode ihrer jeweiligen Aufgabe nachgehen. So besitzt beispielsweise jede Instanz der Klasse `EGOCallCollectionMethodStatement` ein Kollektionsobjekt, einen Methodennamen und gegebenenfalls Parameter für diesen Methodenaufruf als Argumentwerte, sodass sie beim Aufruf ihrer Methode `execute` den repräsentierten Methodenaufruf für das assoziierte Kollektionsobjekt ausführt.

Die Funktionalität der weiteren Klassen ist entsprechend realisiert. Dabei werden Teilaufgaben, die auch von anderen Klassen durchzuführen sind, von statischen Methoden gelöst. So stellt beispielsweise

Abbildung 5.8 Klassen zur Repräsentation generischer Algebraausdrücke in EGO



die Klasse `EGOExecuteStatementListStatement` die Methode `executeStatementList` bereit, damit auch Instanzen anderer Klassen – etwa `EGOForAllLoopStatement` oder `EGOIfStatement` – in die Lage versetzt werden, Anweisungslisten abzuarbeiten.

Auf eine weitergehende Darstellung der Anweisungsklassen wird an dieser Stelle – beispielsweise im Gegensatz zu der in Abschnitt 3.2.3 für Knotentypen und textuelle Ausführungspläne erfolgten ausführlichen Darstellung – verzichtet, weil selbst Systemprogrammierer nur über die Formulierung textueller Optimierungsregeln mit diesem Teil von EGO in Kontakt kommen.

Generische Algebraausdrücke

In Abbildung 5.8 werden die Klassen gezeigt, deren Instanzen der Repräsentation generischer Algebraausdrücke dienen. Ein Systemprogrammierer verwendet diese Instanzen zur Darstellung konkreter Algebraausdrücke eines speziellen Datenbankmanagementsystems innerhalb von EGO. Für die Transformation der konkreten Algebraausdrücke in das hier vorgestellte generische interne Format ist ein Objekt mit Oberklasse `ExpressionConverter` verantwortlich (siehe auch Abschnitt 5.3.1 sowie die Beschreibung von Abbildung 5.10).

Damit generische Algebraausdrücke eine möglichst einfache Struktur besitzen, aber trotzdem einen problemlosen Einsatz von EGO in vielen Datenbankmanagementsystem gewährleisten, stellt EGO fünf verschiedene Klassen zur Repräsentation wesentlicher Komponenten eines konkreten Algebraausdrucks im internen generischen Format bereit.

Bezeichner: Bezeichner in konkreten Algebraausdrücken – etwa Attributnamen – können innerhalb von EGO durch Objekte vom Typ `EGOAlgebraVariableExpression` repräsentiert werden.

Datenbankobjekte: Instanzen der Klasse `EGOAlgebraDatabaseObjectExpression` dienen der Darstellung atomarer Datenbankobjekte. Hierunter fallen im relationalen Modell beispielsweise Attributwerte einzelner Tupel in Relationen. Sofern die Struktur eines Datenbankobjekts – etwa einer Relation –

nicht von Bedeutung ist, kann es ebenfalls als atomar angesehen und durch eine Instanz dieser Klasse repräsentiert werden.

Konstanten: Konstanten des Typs `T` der konkreten Algebra können innerhalb von EGO auf Objekte vom Typ `EGOAlgebraConstantExpression<T>` abgebildet werden. Durch die explizite Speicherung des Objekts als Attributwert werden auch komplexe anwendungsspezifische Objekte als Konstanten im Optimierungsprozess berücksichtigt, indem EGO diese für ein konkretes Datenbankmanagementsystem spezifischen Konstanten kapselt.

Operatoren: Algebraoperatoren werden durch Instanzen der Klasse `EGOAlgebraOperatorExpression` repräsentiert. Dabei sind beliebig viele weitere Algebraausdrücke als Operanden zulässig.

Weitere und unbekannte Komponenten: Tritt in Algebraausdrücken ein Platzhalter auf – beispielsweise *r* für eine beliebige (berechnete) Relation im relationalen Modell –, so lässt sich dieser auf ein Objekt des Typs `EGOAlgebraArbitraryExpression` abbilden. Auch alle weiteren bisher nicht berücksichtigten Komponenten eines konkreten Algebraausdrucks lassen sich durch Instanzen dieser Klasse darstellen.

Diese fünf Klassen unterscheiden sich bei einer Verwendung als Repräsentation eines Musters auch hinsichtlich ihrer Überprüfung auf Übereinstimmung eines gegebenen Algebraausdrucks mit diesem Muster. Aus diesem Grund muss ein Systemprogrammierer bei der Auswahl einer geeigneten Klasse zur internen Repräsentation bestimmter Arten konkreten Algebraausdrücke auch diese Überprüfung berücksichtigen. Die fünf zuvor genannten Klassen werden alle von der abstrakten Oberklasse `EGOAlgebraExpression` abgeleitet. Daher wird diese Oberklasse im Folgenden als erstes vorgestellt, bevor im Anschluss auf die Besonderheiten der einzelnen Unterklassen eingegangen wird.

Jede Instanz einer Unterklasse von `EGOAlgebraExpression` besitzt einen Namen, durch den beispielsweise eine Repräsentation von Informationen über den ursprünglichen Ausdruck der konkreten Algebra des angebundenen Datenbankmanagementsystems möglich ist. So könnte etwa ein Operator **join** der Algebra eben diesen Namen in der internen Repräsentation erhalten, um zu verdeutlichen, dass eine entsprechende Instanz einer Unterklasse von `EGOAlgebraExpression` eben diesen Operator **join** darstellt. Der Wert des Attributs `internalExactMatchingRequired` gibt zudem an, ob dieser Name bei der Überprüfung auf Übereinstimmung mit einem Muster über die Methode `matches` betrachtet wird. So ist es beispielsweise möglich, Operatoren der konkreten Algebra auf ein generisches Algebraausdrucksobjekt mit dem entsprechenden Operatornamen abzubilden. Wird dieser Operator zudem als Teil eines Musters innerhalb einer Regel verwendet, soll er sicherlich in die Überprüfung auf Übereinstimmung mit einem gegebenen Algebraausdrucksobjekt mit einbezogen werden. Eine genauere Beschreibung dieser Überprüfung findet sich weiter unten.

Jedem Algebraausdrucksobjekt kann weiterhin ein Algebratypobjekt – also eine Instanz einer Unterklasse von `EGOAlgebraType` – zugewiesen werden. Dadurch ist es möglich, die Struktur eines konkreten Algebraausdrucks noch genauer im generischen Format darzustellen. Eine genauere Beschreibung dieser generischen Algebratypen findet sich in der Beschreibung von Abbildung 5.9.

Weiterhin können jedem Algebraausdruck – wie bereits oben beschrieben – Eigenschaften zugewiesen werden. Dazu muss ein Systemprogrammierer konkrete Klassen von `EGOProperty` ableiten, die jeweils eine Eigenschaft symbolisieren. Über die Methode `addProperty` wird einem Algebraausdrucksobjekt ein neues Eigenschaftsobjekt zusammen mit dem Namen dieser Eigenschaft übergeben. Durch die Methode `getProperty` wird die schon bei der Beschreibung der Optimierungsregeln erläuterte Funktionalität zum Auslesen eines Eigenschaftsobjekts – beispielsweise zum Vergleich mit einem anderen Eigenschaftsobjekt oder einer äquivalenten textuellen Darstellung des repräsentierten Eigenschaftswerts – realisiert. Die Methode `setPropertyValue` ermöglicht schließlich eine Veränderung des Werts einer Eigenschaft durch Angabe des entsprechenden Namens sowie einer äquivalenten textuellen Darstellung des Eigenschaftswertes.

Die fünf verschiedenen Unterklassen von `EGOAlgebraExpression` lassen sich neben ihren unterschiedlichen Repräsentationsaufgaben für konkrete Algebraausdrücke vor allem durch die verschiedenen Schritte während der Überprüfung auf Übereinstimmung mit einem Muster – also durch die jeweilige Funktionalität der Methode `matches` – voneinander abgrenzen. Ein Systemprogrammierer muss diese Funktionalität gut kennen, um die gewünschten Ergebnisse beim Vergleich von Algebraausdrucksobjekten zu erzielen. Daher wird die Funktionalität der Methode `matches` für die Klasse `EGOAlgebraExpression` und ihre Unterklassen im Folgenden beschrieben. Der hier dargestellte Vergleich geht immer davon aus, dass ein gegebenes Muster überprüft, ob ein übergebener Algebraausdruck mit dem Muster übereinstimmt.

Ein Vorteil dieser Betrachtungsweise liegt im einfachen Verlauf der Überprüfung. Insbesondere kann jedes ein Muster darstellendes Objekt ohne Kenntnis anderer konkreter Klassen zur Repräsentation von Algebraausdrücken entscheiden, ob ein übergebener Algebraausdruck mit diesem Muster übereinstimmt. Würde ein Algebraausdruck testen, ob er auf ein übergebenes Muster passt, so müsste er zwischen den Klassen unterscheiden. Ein konkretes Algebraausdrucksobjekt vom Typ `EGOAlgebraOperatorExpression` kann beispielsweise mit einem übergebenen Musterobjekt vom Typ `EGOAlgebraArbitraryExpression` oder `EGOAlgebraOperatorExpression` übereinstimmen, allerdings nicht mit einer Instanz der Klasse `EGOAlgebraDatabaseObjectExpression`. Ist das Musterobjekt für die Überprüfung auf Übereinstimmung mit einem Algebraausdruck verantwortlich, so muss das übergebene Algebraausdrucksobjekt genau den gleichen – oder im Fall eines Musterobjekts vom Typ `EGOAlgebraArbitraryExpression` einen beliebigen – Typ besitzen. Die Einzelheiten dieser Überprüfung werden im Folgenden näher erläutert.

`EGOAlgebraExpression`: Die gemeinsamen Aufgaben beim Vergleich zweier Algebraausdrucksobjekte werden in der Methode `matches` dieser Klasse zusammengefasst. Sollte eine exakte Übereinstimmung der Namen der Algebraausdrücke erforderlich sein, wird dieses überprüft. Dieses ist genau dann der Fall, wenn das Attribut `internalExactMatchingRequired` des vorliegenden Musterobjekts den Wert `true` besitzt. Anschließend werden gegebenenfalls die assoziierten Algebratypobjekte auf Übereinstimmung überprüft. Verlaufen die durchgeführten Tests erfolgreich, wird ein positives Resultat als Ergebnis des Methodenaufrufs zurückgeliefert. Sollte keine exakte Übereinstimmung der Namen erforderlich gewesen sein, wird das übergebene Algebraausdrucksobjekt unter dem Bezeichner des vorliegenden Objekts über die Methode `EGORule::insertExpression` in den Regelkontext eingefügt.

`EGOAlgebraArbitraryExpression`: Objekte dieser Klasse stellen beim Test auf Übereinstimmung keine Anforderungen an das übergebene Algebraausdrucksobjekt und liefern als Ergebnis das Resultat der Überprüfung durch die entsprechende Methode der Oberklasse `EGOAlgebraExpression` zurück.

`EGOAlgebraConstantExpression<T>`: Ein Objekt dieser Klasse überprüft als erstes, ob das übergebene Algebraausdrucksobjekt ebenfalls den Typ `EGOAlgebraConstantExpression<T>` besitzt. Falls eine exakte Übereinstimmung mit diesem Argument erforderlich ist, wird zudem ein Gleichheitstest für die gekapselten Objekte des Typs `T` durchgeführt. Bei positivem Ausgang wird anschließend die Überprüfung aus `EGOAlgebraExpression` aufgerufen, um das endgültige Ergebnis zu bestimmen.

`EGOAlgebraDatabaseObjectExpression`: Objekte dieses Typs überprüfen nur, ob das übergebene Algebraausdrucksobjekt ebenfalls eine Instanz dieser Klasse darstellt, bevor im Erfolgsfall die Methode `EGOAlgebraExpression::matches` zur Bestimmung des Resultats ausgeführt wird.

`EGOAlgebraOperatorExpression`: Falls das aktuelle Algebraoperatorobjekt weitere Algebraausdrucksobjekte als Operanden assoziiert, so muss deren Anzahl mit der Anzahl der Operanden des übergebenen Algebraausdrucksobjekts – welches ebenfalls den Typ `EGOAlgebraOperatorExpression` besitzen muss – übereinstimmen. Weiterhin muss die Methode `matches` für alle Operanden ein positives Resultat ergeben, bevor abschließend die Methode `matches` der Oberklasse aufgerufen wird.

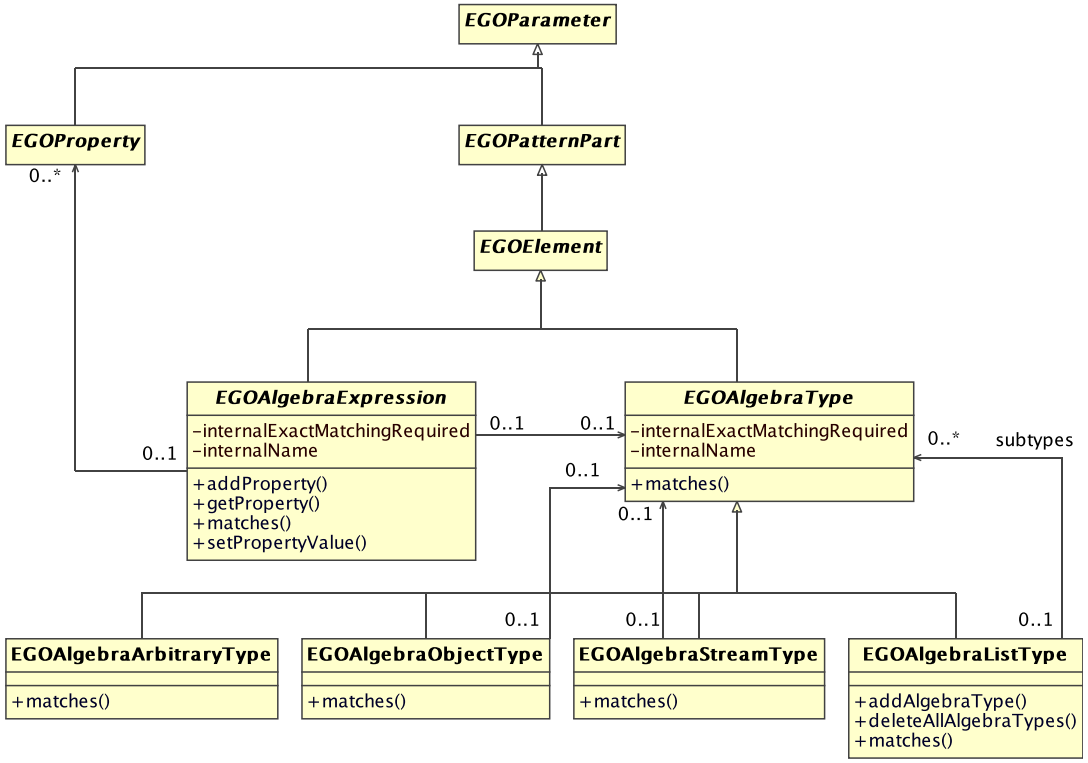
`EGOAlgebraVariableExpression`: Objekte dieser Klasse verhalten sich ähnlich wie Instanzen der Klasse `EGOAlgebraDatabaseObjectExpression`. Repräsentiert das übergebene Algebraausdrucksobjekt ebenfalls einen Bezeichner, so wird als Ergebnis das Resultat der Methode `matches` der Oberklasse zurückgegeben.

An dieser Stelle soll nicht über die allgemeine Beschreibung der Funktionalität der Algebraausdrucksobjekte hinausgegangen werden. Beispiele für die zuvor beschriebenen Eigenschaften finden sich etwa in Abschnitt 5.2.3.

Generische Algebratypen

Abbildung 5.9 stellt die vier von `EGOAlgebraType` abgeleiteten Klassen vor, die EGO zur Abbildung konkreter Algebratypen bereit hält. Der Typ eines Algebraausdrucks lässt sich häufig in vier wesentliche Kategorien einordnen. Dabei wird deutlich, dass ein Systemprogrammierer für ein bestimmtes Szenario – etwa die Abbildung des Typs einer Relation im relationalen Modell – unterschiedliche Möglichkeiten besitzt, um die von EGO bereit gestellten Klassen bei der Anbindung an ein konkretes Datenbankmanagementsystem einzusetzen.

Abbildung 5.9 Klassen zur Repräsentation generischer Algebratypen in EGO



Listen: Listen und Aufzählungen treten häufig als Algebratypen auf, wenn die Struktur eines Rückgabeobjekts näher betrachtet wird. So können im relationalen Modell eine Relation beispielsweise als Liste von Tupeln und diese wiederum als Listen von Attributwerten aufgefasst werden. Innerhalb von EGO können Instanzen der Klasse `EGOAlgebraListType` diesen Sachverhalt darstellen.

Objekte: Vor allem atomare Daten wie beispielsweise Attributwerte besitzen den Algebratyp Objekt. Aber auch Relationen oder Tupel im relationalen Modell können als Objekt aufgefasst werden, falls eine weitere Verfeinerung nicht gewünscht ist. EGO verwendet zur Repräsentation dieses Typs Instanzen der Klasse `EGOAlgebraObjectType`.

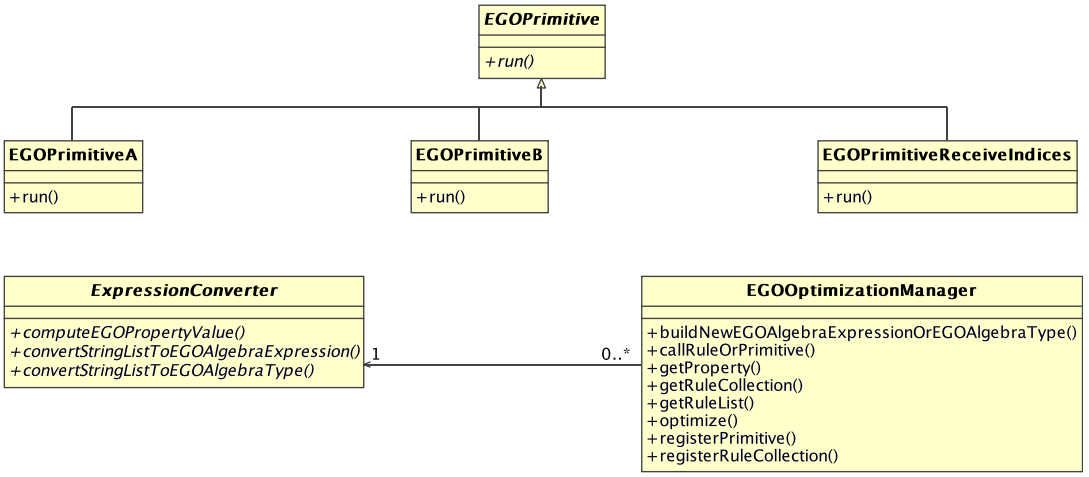
Ströme: Wie bereits in Kapitel 3 ausführlich dargestellt wurde, werden Zwischenergebnisse und Endresultate im Rahmen der Abarbeitung eines Ausführungsplans häufig als Ströme von Objekten aufgefasst. Daher liegt es nahe, Ströme als Algebratyp für die algebraische Repräsentation derartiger Ausführungspläne zu verwenden. Bei einer Verallgemeinerung dieses Konzepts liefern auch Algebraausdrücke zur Repräsentation einer vom Benutzer gestellten Anfrage einen Strom als Ergebnis. So lassen sich im relationalen Modell beispielsweise Relationen, die einem Operator als Operanden dienen, als Strom auffassen. Innerhalb von EGO kann dieser Zusammenhang durch Instanzen der Klasse `EGOAlgebraStreamType` dargestellt werden.

Unbekannte und sonstige Typen: Sofern der Algebratyp nicht von Interesse ist oder sich keine der zuvor beschriebenen Klassen zur Repräsentation des konkreten Algebratyps innerhalb von EGO anbietet, kann eine Instanz der Klasse `EGOAlgebraArbitraryType` zur Erzeugung eines entsprechenden generischen Algebratypobjekts verwendet werden.

Die Funktionalität der Klassen zur Darstellung generischer Algebratypen ähnelt stark den zuvor präsentierten Klassen zur Repräsentation generischer Algebraausdrücke, daher wird an dieser Stelle nur die Funktionalität der Methode `matches` zum Test auf Übereinstimmung zweier Algebratypobjekte näher thematisiert:

`EGOAlgebraType`: Eine Instanz dieser Klasse überprüft bei der Notwendigkeit einer genauen Übereinstimmung, ob das übergebene Objekt den gleichen Namen besitzt. Dieses Resultat stellt das Ergebnis des Vergleichs der beiden Objekte dar. Wird keine genaue Übereinstimmung gefordert, so ist das Resultat dieses Tests auf Übereinstimmung immer positiv.

Abbildung 5.10 Klassen für übergreifende Aufgaben in EGO



`EGOAlgebraArbitraryType`: Eine Instanz zur Repräsentation eines beliebigen Algebratyps stellt keine Anforderung an das übergebene Algebratypobjekt und liefert daher das Ergebnis des entsprechenden Methodenaufrufs der Oberklasse als Rückgabe.

`EGOAlgebraListType`: Für zwei Instanzen dieser Klasse wird überprüft, ob sie die gleiche Anzahl von Listenelementen besitzen und ob diese Listenelemente zudem jeweils übereinstimmen. Im Erfolgsfall wird außerdem `matches` der Oberklasse ausgeführt.

`EGOAlgebraObjectType`: Eine Instanz dieser Klasse überprüft für ein übergebenes Algebratypobjekt, ob es ebenfalls einen Algebraobjekttyp darstellt. Kapselt die aktuelle Instanz zudem ein weiteres Algebratypobjekt, so muss dieses auch für die übergebene Instanz der Klasse `EGOAlgebraObjectType` gelten und die beiden gekapselten Objekte müssen übereinstimmen. Zur endgültigen Bestimmung der Rückgabe dient abschließend der Aufruf der entsprechenden Methode der Oberklasse.

`EGOAlgebraStreamType`: Eine Instanz dieser Klasse führt die erforderliche Überprüfung ähnlich der in Instanzen der Klasse `EGOAlgebraObjectType` erfolgenden Überprüfung durch. Der einzige Unterschied besteht darin, dass die übergebene Instanz zur Klasse `EGOAlgebraStreamType` gehören muss.

Auch hier soll nicht näher auf die Funktionalität der Klassen zur Darstellung generischer Algebratypen oder deren Einsatz im Rahmen der Anfrageoptimierung eingegangen werden, weil die zuvor erfolgte Beschreibung für die Darstellung der Möglichkeiten von EGO sowie für das weitere Verständnis dieser Arbeit ausreicht. Zudem finden diese Klassen zur Darstellung generischer Algebratypen bei der Einbindung von EGO in GOODAC (siehe auch Abschnitt 5.3.1) keine Verwendung.

Klassen für übergreifende Aufgaben

Im Folgenden soll die bisher noch nicht beschriebene Funktionalität von EGO vorgestellt werden. Diese wird im Wesentlichen durch die in Abbildung 5.10 gezeigten Klassen realisiert.

Die bereits oben erläuterten Primitive werden intern durch Instanzen mit Oberklasse `EGOPrimitve` dargestellt. Jede derartige Klasse muss die Methode `run` implementieren, die die Aufgabe eines Primitivs erledigt. Als Parameter dieser Methode dient eine Liste vom Typ `EGOList<EGOParameter>`. Abbildung 5.10 zeigt eine Klasse `EGOPrimitiveReceiveIndices`, deren Aufgabe darin besteht, zu einem gegebenen Objekt vom Typ `EGOAlgebraDatabaseObjectExpression` alle für einen Zugriff auf dieses Datenbankobjekt geeigneten Indexstrukturen zu bestimmen. Die zusätzlich dargestellten Klassen `EGOPrimitiveA` und `EGOPrimitveB` dienen nur der Visualisierung der grundsätzlichen Struktur zur Einbettung weiterer Primitive in EGO und realisieren eine hier nicht näher festgelegte Funktionalität über ihre Methode `run`.

Die Konvertierung von konkreten Algebraausdrücken in das generische interne Format – und ebenso umgekehrt – wird durch eine vom Systemprogrammierer zu entwickelnde Unterklasse der in Abbildung 5.10 dargestellten Klasse `ExpressionConverter` erreicht. Diese Unterklasse muss die dort deklarierten Methoden überschreiben, um eine korrekte Transformation zwischen einer textuellen Beschreibung

von konkreten Algebraausdrücken des vorliegenden Datenbankmanagementsystems und generischen Algebraausdrücken im internen Format von EGO zu gewährleisten. Ebenso müssen sich durch die Methoden der von `ExpressionConverter` für ein konkretes Datenbankmanagementsystem abgeleiteten Unterklasse textuelle Beschreibungen konkreter Algebratypen in generische Algebratypen überführen lassen. Diese Methoden werden zum Beispiel im Rahmen der Ausführung von Anweisungen – also dem Aufruf der Methode `execute` von Instanzen mit Oberklasse `EGOStatement` – benötigt. Beispielsweise werden konkrete Algebraausdrücke, die in Mustern in der textuellen Beschreibung von Optimierungsregeln vorkommen, über die Methode `convertStringListToEGOAlgebraExpression` in eine objektbasierte Darstellung im generischen internen Format überführt.

Ein Systemprogrammierer muss bei der Anbindung von EGO an ein konkretes Datenbankmanagementsystem und der damit verbundenen Bereitstellung einer konkreten Unterklasse von `ExpressionConverter` vor allem beachten, dass er neben der korrekten Transformation von Algebraausdrücken vom konkreten ins generische und vom generischen ins konkrete Format auch die Semantik seiner Optimierungsregeln berücksichtigt. Enthält eine Regel etwa ein Muster der Form

```
{ leftOperand operator rightOperand } : Expr
```

und soll dieser konkrete Algebraausdruck andeuten, dass jeder Operator mit genau zwei Operanden für eine Übereinstimmung mit diesem Muster sorgt, so zählt es zu den Aufgaben des Systemprogrammierers, dieses Verhalten auch bei Durchführung der Konvertierung von einem konkreten zu einem generischen Algebraausdruck zu gewährleisten. Hierzu muss er die oben beschriebenen Klassen zur Repräsentation generischer Algebraausdrücke geschickt einsetzen, damit die Überprüfung auf Übereinstimmung mit einem Muster genau in den gewünschten Fällen zum Erfolg führt.

Falls – wie im zuvor angeführten Beispiel – Namen für Platzhalter im Rahmen eines Musters bei der textuellen Beschreibung konkreter Algebraausdrücke verwendet werden, kann der entsprechende Teil eines übergebenen generischen Algebraausdrucks beim Test auf Übereinstimmung unter diesem Namen im aktuellen Regelkontext gespeichert werden. Eine derartige Speicherung erfolgt immer dann, wenn keine genaue Übereinstimmung der Namen für generische Algebraausdrücke erforderlich ist. Tritt nun dieser Name des Platzhalters während der weiteren Abarbeitung der vorliegenden Optimierungsregel erneut auf, so ist eine Instanz der vom Systemprogrammierer realisierten Unterklasse von `ExpressionConverter` in der Lage, über diesen Namen die ursprünglichen Algebraausdrücke aus dem Regelkontext auszulesen. So können beispielsweise innerhalb der Anweisung

```
return { rightOperand operator leftOperand };
```

über die im Algebraausdruck auftretenden Namen von Platzhaltern die ursprünglichen Operanden und der eingangs übergebene Operator aus dem Regelkontext ausgelesen werden, um einen neuen generischen Algebraausdruck zu erzeugen, der dem alten generischen Algebraausdruck bis auf Vertauschung seiner Operanden genau entspricht.

Die bisher nicht erwähnte Methode `computeEGOPropertyValue` der Klasse `ExpressionConverter` sorgt dafür, dass ein Eigenschaftswert für eine gegebene Instanz mit Oberklasse `EGOAlgebraExpression` und einen Eigenschaftsnamen berechnet wird, sofern dieser Eigenschaftswert nicht schon im Rahmen der ursprünglichen Konvertierung dem Algebraausdrucksobjekt über seine Methode `addProperty` hinzugefügt wurde. Daher muss der Systemprogrammierer dafür sorgen, dass alle Eigenschaften, die in Optimierungsregeln verwendet werden, bereits während der Konvertierung oder aber über die Methode `getProperty` für jeden auftretenden generischen Algebraausdruck bestimmt werden können.

Die Klasse `EGOOptimizationManager` stellt schließlich Instanzen zur Steuerung des Optimierungsprozesses bereit. Die beiden Methoden `registerPrimitive` und `registerRuleCollection` sorgen im Vorfeld des Optimierungsprozesses – also beim Einlesen der Optimierungsregeln – dafür, dass die verfügbaren Primitive und die angegebenen Optimierungsregeln bei der Optimierung von Anfragen Anwendung finden können.

Die Methode `optimize` sorgt für den Start des Optimierungsprozesses. Sie erhält einen generischen Algebraausdruck als Parameter, auf den sie eine unter dem reservierten Namen `main` verfügbare Optimierungsregel anwendet. Das Ergebnis dieser Regelanwendung stellt zugleich das Resultat des Optimierungsprozesses – gewöhnlich einen weiteren generischen Algebraausdruck – dar.

Die übrigen Methoden dieser Klasse werden üblicherweise während des Optimierungsprozesses benötigt. So sorgt etwa `buildNewEGOAlgebraExpressionOrEGOAlgebraType` dafür, dass aus einer textuellen Darstellung eines konkreten Algebraausdrucks oder eines konkreten Algebratyps durch das assoziier-

te Objekt mit Oberklasse `ExpressionConverter` eine entsprechende Darstellung im generischen internen Format erzeugt wird. Weiterhin tauchen in Optimierungsregeln Aufrufe weiterer Regeln oder Primitive auf. Diese werden mit Hilfe ihres Namens über die Methode `getRuleOrPrimitive` aus der aktuellen Instanz der Klasse `EGOOptimizationManager` ausgelesen. In diesem Zusammenhang müssen unter Umständen die im Vorfeld registrierten Optimierungsregeln und Kollektionen von Regeln über die Methoden `getRuleList` – für alle Regeln mit einem gegebenen Namen – oder `getRuleCollection` – für eine Kollektion von Regeln, also beispielsweise eine ursprüngliche Regelgruppe – aufgerufen werden. An dieser Stelle müssen Listen von Regeln berücksichtigt werden, weil im Allgemeinen mehrere Regeln den gleichen Namen besitzen können (siehe auch Abschnitt 5.2.3); in diesem Fall erfolgt die Auswahl einer passenden Regel durch sequentielle Überprüfung der initialen Musterliste jeder Regel mit den für den Regelaufruf bereitgestellten Argumenten. Die Methode `getProperty` sorgt schließlich für den Aufruf der Methode `computeEGOPropertyValue` des assoziierten Objekts mit Oberklasse `ExpressionConverter`, um für einen generischen Algebraausdruck einen bestimmten Eigenschaftswert zu berechnen.

5.2.3 Beispiele für die Funktionsweise von EGO

Nachdem zuvor sowohl die Schnittstellen als auch die interne Realisierung von EGO vorgestellt worden sind, sollen diese Konzepte im Folgenden zum besseren Verständnis an Beispielen erläutert werden. Dazu wird als erstes das Einlesen der in textueller Form vorliegenden Optimierungsregeln und ihre interne objektbasierte Darstellung thematisiert, bevor näher auf die Anwendung der Optimierungsregeln während des Optimierungsprozesses eingegangen wird.

Einlesen von Optimierungsregeln und ihre objektbasierte Darstellung

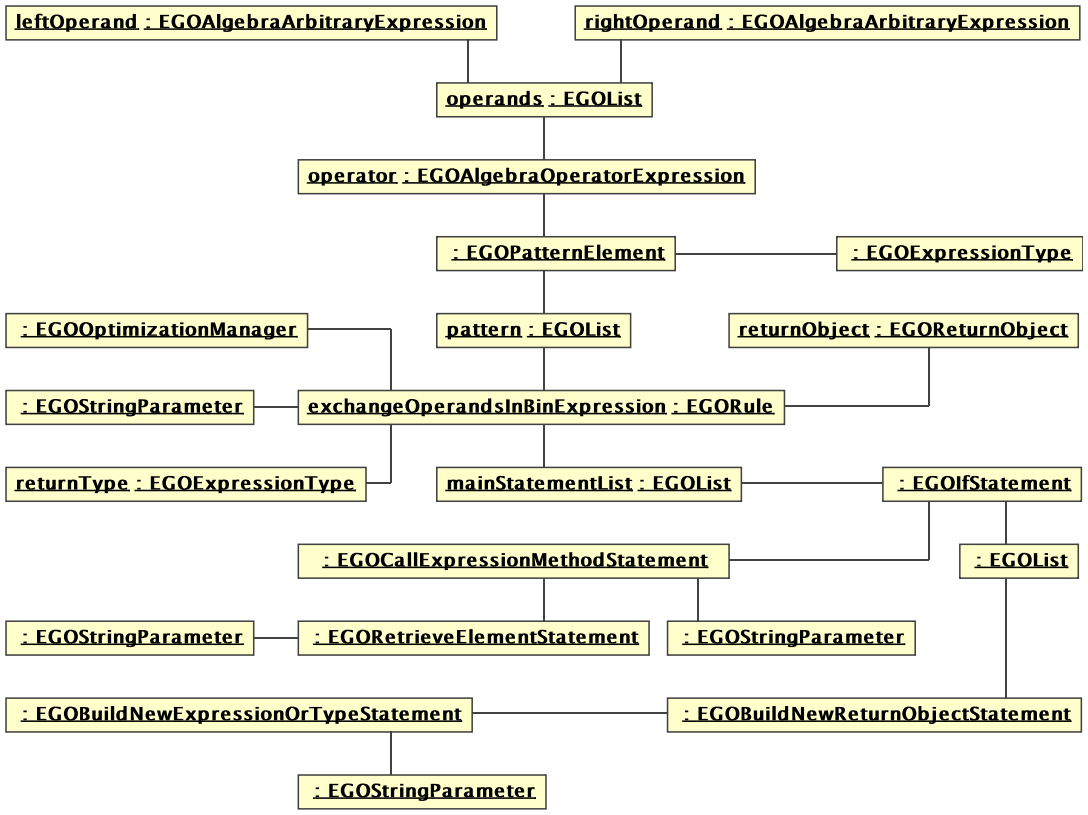
Anwendungsentwickler und Systemprogrammierer stellen Optimierungsregeln in textueller Form bereit. In Abschnitt 5.2.1 wurde bereits der grundlegende Aufbau dieser Optimierungsregeln erläutert, während durch Abschnitt 5.2.2 die interne Darstellung der Optimierungsregeln vorgestellt wurde. Die Aufgabe beim Einlesen von Optimierungsregeln besteht nun im Wesentlichen darin, ihre textuelle Darstellung in das interne Format für Optimierungsregeln zu überführen. Die technischen Grundlagen der hierbei eingesetzten lexikalischen und syntaktischen Analyse sind bereits bei der Beschreibung der Verarbeitung textueller Ausführungspläne in Abschnitt 3.2.4 zu finden, daher wird an dieser Stelle vor allem Wert auf die detaillierte Erläuterung der internen objektbasierten Struktur nach dem Einlesen textueller Optimierungsregeln gelegt.

So wird beispielsweise auf Seite 87 in Beispiel 5.1 die textuelle Beschreibung einer einfachen Optimierungsregel `exchangeOperandsInBinExpression` gezeigt, die eine Vertauschung der Operanden eines kommutativen Operators beschreibt. Während des Einlesens dieser Optimierungsregel wird die in Abbildung 5.11 gezeigte objektbasierte Repräsentation dieser Optimierungsregel erzeugt.

Als Ausgangspunkt dient das Regelobjekt `exchangeOperandsInBinExpression`. Es assoziiert neben einem Objekt des Typs `EGOStringParameter` zur Repräsentation des Regelnamens auch eine Instanz der Klasse `EGOOptimizationManager`, um Zugriff auf weitere Regelobjekte, Primitive und eine konkrete Konvertierungskomponente zur Berechnung der Eigenschaften von Algebraausdrücken zu erhalten. Weiterhin gibt das Objekt `returnType` den Rückgabetyp der Regel – in diesem Beispiel ein Algebraausdrucksobjekt – für den Fall ihrer Ausführung an. Die eingehende Musterliste enthält nur einen Bestandteil des Typs `EGOPatternElement`. Dieses gibt über eine Instanz der Klasse `EGOExpressionType` an, dass es sich bei diesem Muster um einen Algebraausdruck im internen generischen Format handelt. Die Darstellung des repräsentierten konkreten Algebraausdrucks in der textuellen Beschreibung der Optimierungsregel wird beim Einlesen dieser Regel in die gezeigte objektbasierte generische Repräsentation überführt. Das geschieht durch eine Instanz der vom Systemprogrammierer von der Klasse `ExpressionConverter` abgeleiteten konkreten Unterklasse. Das assoziierte Objekt vom Typ `EGOReturnObject` sorgt dafür, dass auch bei einer nicht erfolgreichen Regelanwendung zumindest der implizite boolesche Rückgabewert ausgelesen werden kann.

Schließlich assoziert das Regelobjekt `exchangeOperandsInBinExpression` noch die Anweisungsliste `mainStatementList`, die bei Übereinstimmung eines übergebenen Objekts vom Typ `EGOElement` mit dem zuvor beschriebenen Muster der Regel zur Anwendung kommt. Diese Anweisungsliste enthält nur ein Objekt vom Typ `EGOIfStatement`, dass die bedingte Verzweigung aus der textuellen Beschreibung der Optimierungsregel darstellt. Als Bedingung kommt eine Instanz der Klasse `EGOCallExpressionMethod` zum Einsatz, die bei Ausführung der Regel den Wert der Eigenschaft `commutative` bestimmt. Dazu erhält

Abbildung 5.11 Interne Darstellung der eingelesenen Optimierungsregel aus Beispiel 5.1



diese Instanz diesen Eigenschaftsnamen, den Methodennamen `getProperty` und ein Objekt vom Typ `EGORetrieveElementStatement`, das das unter dem Namen `operator` im Regelkontext gespeicherte Element zurückliefert, als Parameter.

Die bei Zutreffen der Bedingung auszuführenden Anweisungen werden in der von der Instanz der Klasse `EGOIfStatement` assoziierten Liste durch entsprechende Anweisungsobjekte bereit gehalten. In diesem Beispiel enthält die Liste nur ein Objekt vom Typ `EGOBuildNewReturnObjectStatement`, das das Rückgabeobjekt mit Hilfe einer Instanz der Klasse `EGOBuildNewExpressionOrTypeStatement` erzeugt. Dabei wird der in der textuellen Beschreibung enthaltene konkrete Algebraausdruck bei Ausführung der repräsentierten Anweisung in einen neuen generischen Algebraausdruck transformiert. Hierfür ist ebenfalls die bereits oben erwähnten Instanz der vom Systemprogrammierer von der Klasse `ExpressionConverter` abgeleiteten konkreten Unterklasse verantwortlich. Hierzu werden die zu Beginn der Regelausführung bei Überprüfung der Musterliste im aktuellen Regelkontext gespeicherten Teile des Algebraausdrucks über ihre Namen `leftOperand`, `operator` und `rightOperand` berücksichtigt, sodass als Ergebnis das ursprüngliche Algebraausdrucksobjekt mit vertauschter Reihenfolge der beiden Operanden zurückgeliefert wird. Sollte diese Rückgabeanweisung bei der späteren Anwendung der vorliegenden Optimierungsregel ausgeführt werden, so wird zusätzlich zur Rückgabe dieses Ergebnisses auch der implizite Rückgabewert der vom Regelobjekt assoziierten Instanz des Typs `EGOReturnObject` auf `true` gesetzt.

Ausgehend vom zuvor beschriebenen Beispiel lässt sich die interne objektbasierte Struktur auch für weitere Optimierungsregeln darstellen. Da jedoch Systemprogrammierer und Anwendungsentwickler mit dieser internenen Struktur der Optimierungsregeln nicht in Berührung kommen und das Konzept der Optimierungsregeln schon ausführlich dargestellt wurde, findet an dieser Stelle keine weitere Betrachtung der internen Struktur von Optimierungsregeln statt.

Anwendung von Optimierungsregeln

Die Anwendung von Optimierungsregeln gliedert sich im Wesentlichen in vier Schritte:

Aufruf der Regel: Eine Optimierungsregel wird über die Angabe ihres Namens oder über das Auslesen aus einem Kollektionsobjekt – das etwa beim Einlesen der Optimierungsregeln aus den Regeln einer

Regelgruppe entstanden ist – aufgerufen. Beim Aufruf einer Optimierungsregel über die Angabe ihres Namens innerhalb einer weiteren Optimierungsregel aktiviert diese die Methode `getRuleList` der assoziierten Instanz der Klasse `EGOptimizationManager`. Diese liefert alle Regeln mit dem angegebenen Namen zurück. Gewöhnlich handelt es sich dabei nur um eine einzige Optimierungsregel. Um aber abhängig von den übergebenen Argumenten unterschiedliche Funktionalitäten der Regelanwendung zu gewährleisten, besteht die Möglichkeit, Polymorphismus durch Angabe mehrerer Regeln gleichen Namens mit unterschiedlichen Musterlisten zu realisieren. Wird eine Optimierungsregel aus einem Kollektionsobjekt – über dessen Methode `get` – ausgelesen, sind keine weiteren Schritte zur Vorbereitung der Ausführung erforderlich.

Überprüfung der Musterliste: Bevor eine Regel zur Anwendung kommen kann, werden die übergebenen Argumente auf Übereinstimmung mit der Musterliste geprüft. Dazu wird die statische Methode `testWhetherPatternListIsMatched` der Klasse `EGOPatternMatchingStatement` aufgerufen, die wiederum mit Hilfe der statischen Methode `testWhetherPatternIsMatched` der gleichen Klasse diese Überprüfung vornimmt. Für generische Algebraausdrücke und generische Algebratypen wurden Einzelheiten dieser Überprüfung weiter oben genauer beschrieben. Sollte der Methodenaufruf ein positives Ergebnis zurückliefern, wird die Anweisungsliste der Regel ausgeführt, ansonsten kommt die Optimierungsregel nicht zur Anwendung. Wurde die Regel über ihren Namen aufgerufen und existieren weitere Regeln dieses Namens, so wird solange nach einer passenden Regel gesucht, bis die Überprüfung auf Übereinstimmung zwischen übergebenen Argumenten und Musterliste der Optimierungsregel erfolgreich verläuft.

Ausführung der Anweisungsliste: Jede Optimierungsregel enthält in ihrer objektbasierten Darstellung unter dem Attributnamen `mainStatementList` eine Anweisungsliste, die bei Anwendung der Regel zur Ausführung kommt. Eine derartige Anweisungsliste besteht im Wesentlichen aus Anweisungsobjekten, die unter Umständen weitere Anweisungslisten oder andere Anweisungen als Parameter besitzen, sodass beispielsweise bedingte Verzweigungen oder geschachtelte Anweisungen aus der textuellen Beschreibung von Optimierungsregeln korrekt in der objektbasierten Darstellung abgebildet werden. Eine Anweisungsliste wird über die statische Methode `executeStatementList` der Klasse `EGOExecuteStatementListStatement` abgearbeitet. Eine genauere Beschreibung der Abarbeitung einzelner Anweisungen über Aufruf der Methode `execute` des entsprechenden Anweisungsobjekts findet sich bereits auf Seite 95.

Erzeugung eines Rückgabeobjekts: Sollte bei der Abarbeitung einer Anweisungsliste eine Rückgabeanweisung erreicht werden, so wird die Ausführung der Anweisungsliste damit beendet. Das zurückzugebende Element – eine Instanz mit Oberklasse `EGOElement` – wird in einem Rückgabeobjekt – also einem Objekt des Typs `EGOReturnObject` – gekapselt. Zusätzlich wird der implizite Rückgabewert der Regelanwendung auf `true` gesetzt, weil eine Rückgabeanweisung erreicht wurde und die Regelanwendung somit als erfolgreich abgeschlossen gilt. Dieser implizite Rückgabewert wird ebenfalls in dem zuvor erzeugten Rückgabeobjekt gespeichert.

Im Folgenden soll die Anwendung von Optimierungsregeln am Beispiel der Ausführung der auch zuvor angeführten Optimierungsregel `exchangeOperandsInBinExpression` veranschaulicht werden. Dazu werden insbesondere die letzten beiden der zuvor beschriebenen vier Schritte am Beispiel dieser Optimierungsregel erläutert. Die verwendeten Sequenzdiagramme verzichten hierbei aus Platzgründen auf die Darstellung der Parameter der Methodenaufrufe, auf die Nennung von Listenobjekten sowie auf die Kennzeichnung trivialer Rückgabewerte.

Als erstes wird die Regel `exchangeOperandsInBinExpression` aufgerufen. Dieser Aufruf kann beispielsweise aus der Anwendung einer anderen Optimierungsregel heraus über ein dort erzeugtes Objekt vom Typ `EGOCallRuleOrPrimitiveStatement` erfolgen. Stimmt das übergebene Argument mit dem Muster dieser Optimierungsregel überein, so kann die Regel `exchangeOperandsInBinExpression` angewendet werden, sodass insbesondere ihre Anweisungsliste ausgeführt wird.

Abbildung 5.12 zeigt den Beginn der Ausführung dieser Anweisungsliste. Nachdem ein Objekt des Typs `EGOCallRuleOrPrimitiveStatement` sichergestellt hat, dass die Musterliste dieser Optimierungsregel mit den übergebenen Argumenten übereinstimmt, ruft es die Methode `execute` des ersten Anweisungsobjekts – im Beispiel eine Instanz der Klasse `EGOIfStatement` – auf. Diese bereitet über `prepareExecution` ihre Ausführung vor und vereinfacht vor allem die Parameter des Anweisungsobjekts über die Methode `resolveParameterList`, indem sie dafür sorgt, dass als Argument dienende Anweisungen ausgeführt

und in der Parameterliste durch ihre Ergebnisse ersetzt werden. Dazu werden alle Bestandteile der Parameterliste durch `resolveParameter` und im Fall eines Anweisungsobjekts durch `resolveStatement` näher betrachtet. Diese geschachtelten Methodenaufrufe werden im Folgenden bei der Ausführung weiterer Anweisungsobjekte aus Übersichtlichkeitsgründen nicht mehr im Detail dargestellt.

Um den vom Anweisungsobjekt des Typs `EGOCallExpressionMethodStatement` zu erzeugenden Parameterwert zu erhalten, wird erneut `execute` aufgerufen. Auch hier liegen wiederum nicht alle Parameter in ihrer benötigten Form vor. Daher ist eine Instanz der Klasse `EGORetrieveElementStatement` dafür verantwortlich, das während der Betrachtung der Musterliste dieser Optimierungsregel unter dem Namen `operator` gespeicherte generische Algebraausdrucksobjekt aus dem aktuellen Regelkontext auszulesen. Dieser Regelkontext wird jeweils als veränderbares Argument an die Methode `execute` übergeben, sodass während der Ausführung von Anweisungen alle benötigten Informationen über an Bezeichner gebundene Elemente – wie hier das unter dem Namen `operator` erreichbare Algebraausdrucksobjekt – verfügbar sind. Die aktuelle Instanz der Klasse `EGORetrieveElementStatement` muss keine wesentlichen Aufgaben zur Vorbereitung der Ausführung ihrer eigentlichen Anweisung verrichten, sodass innerhalb der Methode `prepareExecution` für diese Instanz nichts Entscheidendes passiert. Anschließend ruft diese Instanz ihre Methode `retrieveAlgebraExpression` auf. Damit wird aus dem aktuellen Regelkontext über Aktivierung der Methode `getAlgebraExpression` des Objekts vom Typ `EGORule` das unter dem Bezeichner `operator` gespeicherte Algebraausdrucksobjekt zurückgeliefert. Die aktuelle Instanz der Klasse `EGORetrieveElementStatement` reicht dieses Element an das übergeordnete Objekt weiter und beendet damit ihre Arbeit.

Dieses übergeordnete Objekt des Typs `EGOCallExpressionMethodStatement` hat damit die Vorbereitungen seiner Ausführung beendet und kann jetzt mit Hilfe der Methode `getProperty` den Wert der Eigenschaft `commutative` beim erhaltenen Algebraausdrucksobjekt erfragen. Für den weiteren Verlauf dieses Beispiels wird angenommen, dass dieser Eigenschaftswert dem booleschen Wert `true` entspricht.

In Abbildung 5.13 ist der weitere Verlauf dieser Regelanwendung zu sehen. Da die Auswertung der Bedingung zu einem positiven Ergebnis geführt hat, ruft die Instanz der Klasse `EGOIfStatement` das erste und einzige Anweisungsobjekt der zugehörigen Anweisungsliste über dessen Methode `execute` auf. Dieses sorgt wieder durch `prepareExecution` für eine Vorbereitung der eigentlichen Ausführung. Hier muss insbesondere das zurückzugebende Element – ein generisches Algebraausdrucksobjekt – von einer Instanz der Klasse `EGOBuildNewExpressionOrTypeStatement` erzeugt werden. Die Verwendung der von EGO bereitgestellten Klassen zur Erzeugung generischer Algebraausdrücke und damit auch die Struktur der erzeugten generischen Algebraausdrücke ist vom konkreten Datenbankmanagementsystem, in das EGO eingebunden wird, abhängig. Deswegen muss ein Objekts vom Typ `EGOOptimizationManager` mit Hilfe einer Instanz der vom Systemprogrammier für das konkrete Datenbankmanagementsystem erzeugten Unterklasse zur abstrakten Klasse `ExpressionConverter` durch die Verwendung ihrer Methode `convertStringListToEGOAlgebraExpression` dafür sorgen, dass ein entsprechendes generisches Algebraausdrucksobjekt – also eine Instanz einer Unterklasse von `EGOAlgebraExpression` – entsteht. Diese Instanz wird in Abbildung 5.13 mit `e` bezeichnet.

Mit diesen Schritten ist die Vorbereitung der Ausführung der eigentlichen Aufgabe des vorliegenden Objekts vom Typ `EGOBuildNewReturnObject` abgeschlossen, sodass es mit seiner eigentlichen Arbeit beginnen kann. Als erstes setzt es dazu den impliziten Rückgabewert eines neu erzeugten Rückgabeobjekts auf `true`, weil eine Rückgabeanweisung während der Regelanwendung erreicht wurde. Weiterhin setzt es das explizite Ergebnis `e` dieser Regelanwendung über die Methode `setEGOElement`, bevor das Rückgabeobjekt in den aktuellen Regelkontext eingefügt wird.

Damit ist die Anwendung dieser Regel beendet. Die weiteren Aktionen finden sich nicht mehr in den zuvor beschriebenen Sequenzdiagrammen. Als erstes beenden alle noch aktiven Anweisungsobjekte ihre Arbeit, bevor die Konktrolle auf das aufrufende Objekt des Typs `EGOCallRuleOrPrimitiveStatement` zurückfällt. Dieses wurde – wie bereits oben beschrieben – beispielsweise im Rahmen einer übergeordneten Regelanwendung aktiviert, sodass es das Ergebnis der zuvor ausführlich beschriebenen Regelanwendung an diese urspünglich ausgeführte Optimierungsregel zurückgeben kann.

5.3 Anfrageoptimierung in GOODAC – Eine exemplarische Anwendung von EGO

Wie bereits oben erwähnt wurde, sollen in dieser Arbeit weder eine möglichst gute Optimierungsstrategie noch ein passendes Kostenmodell für den Einsatz im Rahmen der Anfrageoptimierung in GOODAC entwi-

Abbildung 5.12 Anwendung einer Optimierungsregel, Teil 1

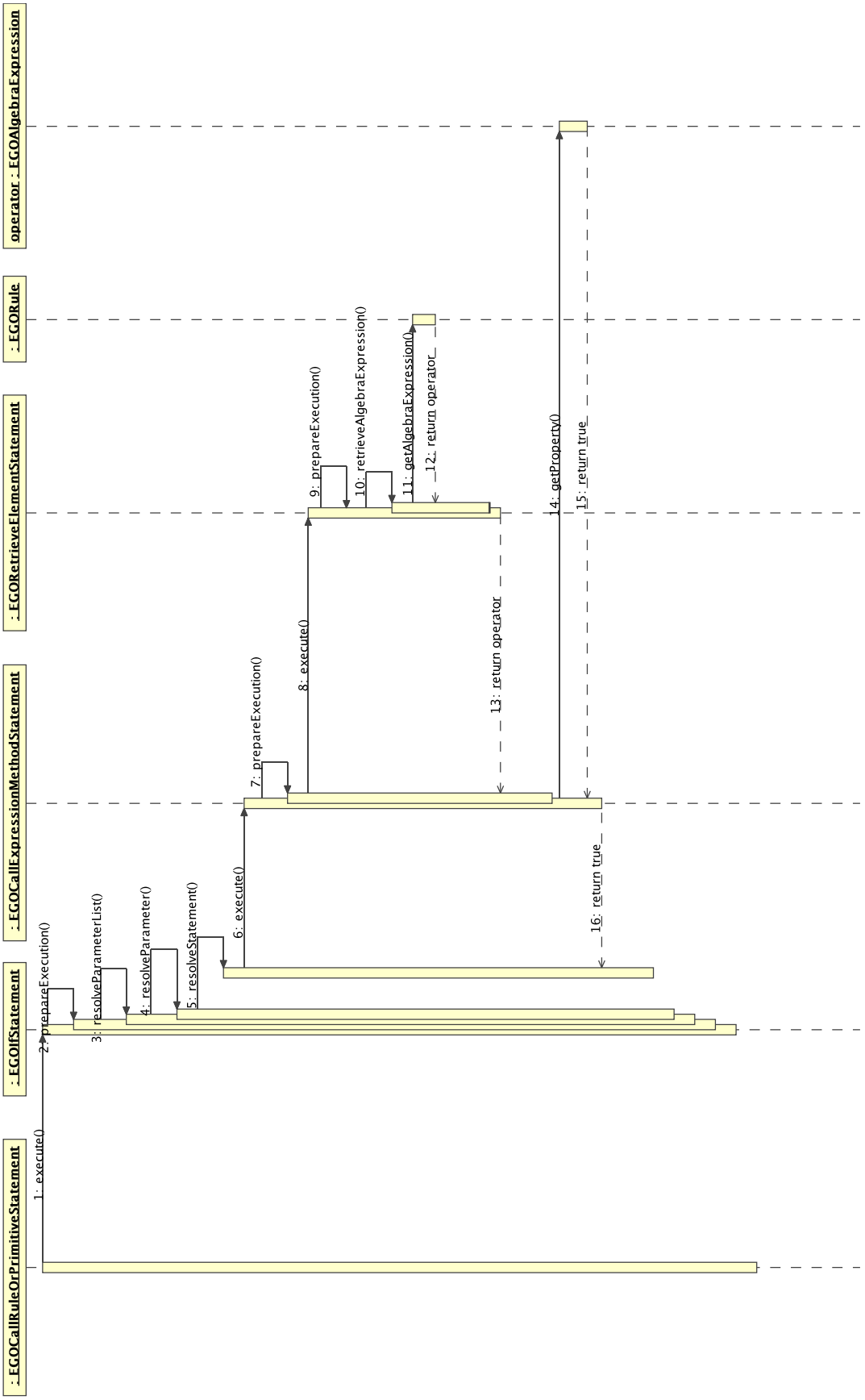
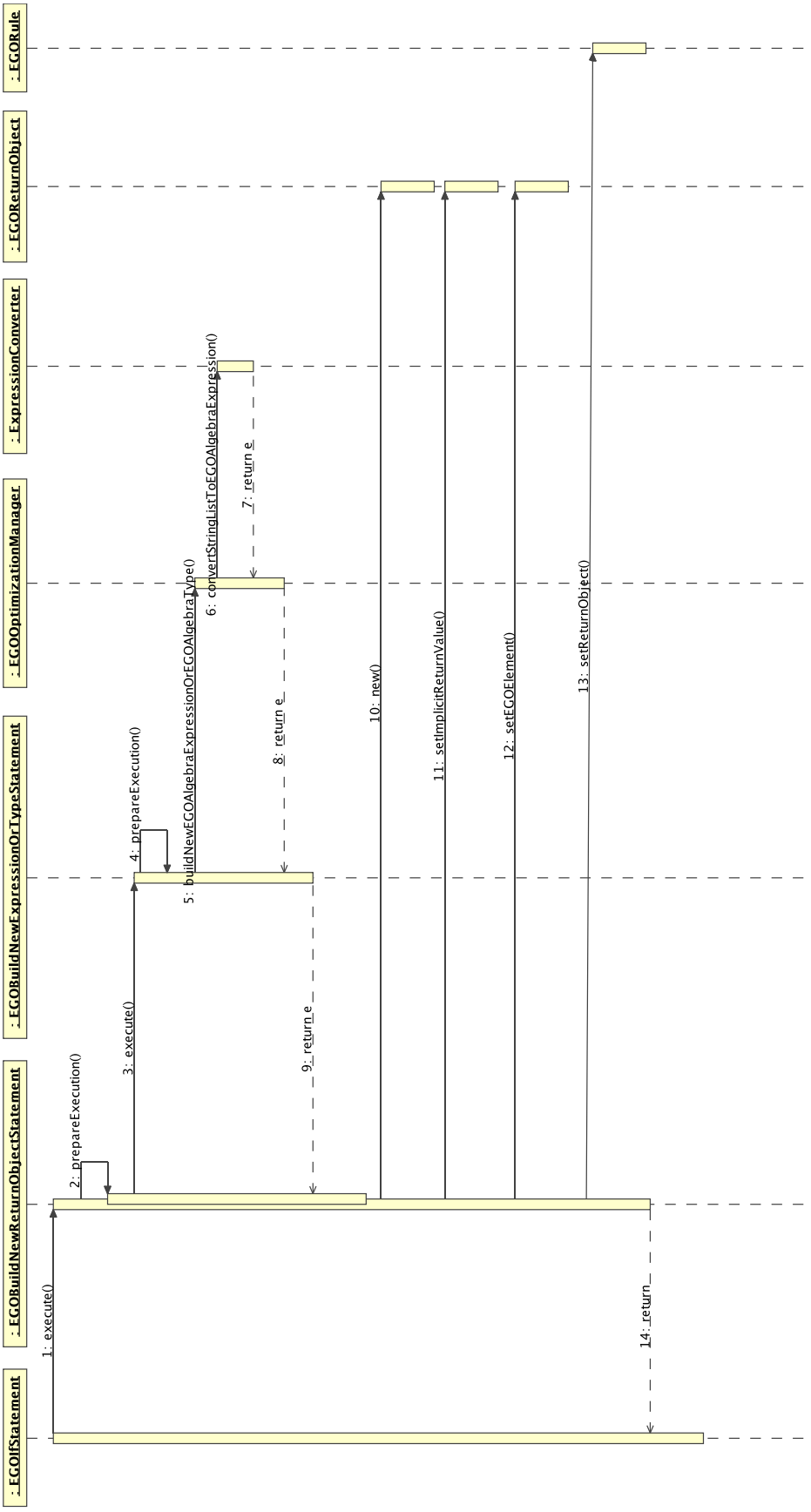


Abbildung 5.13 Anwendung einer Optimierungsregel, Teil 2



ckelt werden. Stattdessen liegt das Ziel dieser Arbeit in der Bereitstellung einer Umgebung, die es einem Systemprogrammierer in GOODAC ermöglicht, unterschiedliche Optimierungsstrategien und Kostenmodelle ohne Schwierigkeiten miteinander zu vergleichen und zu verändern. Dazu müssen sich insbesondere neue Algebraoperatoren, zusätzliche Datentypen sowie weitere Indexsstrukturen problemlos in den Optimierungsprozess eingliedern lassen. Auch die einfache Anpassung der Optimierungsstrategie ist in diesem Kontext von wesentlicher Bedeutung. Schließlich stellt GOODAC nur ein GIS-Kernsystem dar, das für den Einsatz in einem bestimmten Anwendungsgebiet gewöhnlich um spezielle Konzepte der Anwendungsdomäne erweitert werden muss. Daher kann eine für GOODAC entwickelte Optimierungsstrategie nicht alle späteren Einsatzgebiete abdecken. Auch die Bereitstellung mehrerer unterschiedlicher Optimierungsstrategien führt voraussichtlich zu keinem zufriedenstellenden Ergebnis, da auch in diesem Fall die Besonderheiten zukünftiger Erweiterungen nur sehr schwer berücksichtigt werden können, sodass sich hier erneut die Notwendigkeit für spätere Erweiterungen der Optimierungsstrategie zeigt.

Die zuvor beschriebene Komponente EGO zur erweiterbaren generischen Anfrageoptimierung erfüllt alle diese Anforderung. Durch das verwendete regelbasierte Konzept und die Möglichkeit, die Menge der eingesetzten Optimierungsregeln ohne nachfolgende Neuübersetzung von Teilen dieser Komponente zur Anfrageoptimierung zu verändern, erleichtert EGO vor allem Systemprogrammierern die Arbeit, die GOODAC in einem konkreten Anwendungskontext einsetzen und beispielsweise um weitere Operationen auf der ausführbaren Ebene – also Knotentypen für ausführbare Anfragegraphen (siehe auch Abschnitt 3.1.1 und Abschnitt 3.2.3) – erweitern. Ein Systemprogrammierer muss in einem derartigen Fall für die Berücksichtigung einer neuen Operation im Optimierungsprozess nur einige weitere Optimierungsregeln zum von EGO einzulesenden Regelsatz hinzufügen. Auch leichte Veränderugen der eingesetzten Optimierungsstrategie können durch kleine Veränderungen einiger Optimierungsregeln erfolgen. Zusätzliche Erweiterungsmöglichkeiten von EGO und den übrigen an der Anfragebearbeitung in GOODAC beteiligten Komponenten werden in Abschnitt 6.2 aufgeführt.

Weil EGO – wie zuvor geschildert – die Anforderungen an eine Komponente zur Anfrageoptimierung in GOODAC erfüllt, wird EGO in den Optimierungsprozess in GOODAC eingebunden. Bei der Entwicklung einer passenden Optimierungsstrategie kann ein Systemprogrammierer beispielsweise auf die Arbeit von Gaede [Gae96] zurückgreifen. Dort werden bereits einige wesentliche Anätze zur erweiterbaren Anfrageoptimierung in räumlichen Datenbanksystemen vorgestellt. Gaede untersucht zum Beispiel die Auswertung benutzerdefinierter Funktionen und Methoden im Rahmen der Anfragebearbeitung. Zudem liefert er ein Kostenmodell für die Berechnung von Verbundoperationen und präsentiert ausgehend von seinen Überlegungen einen Rahmen zur Entwicklung einer konkreten Komponente zur Anfragebearbeitung.

Im folgenden Abschnitt 5.3.1 wird zuerst die Einbindung von EGO in GOODAC beschrieben. Im Anschluss erläutert Abschnitt 5.3.2 die verwendete Optimierungsstrategie, bevor sich abschließend noch einige Beispiele zur Erläuterung der Funktionsweise der Anfrageoptimierung in GOODAC (Abschnitt 5.3.3) finden. Daneben liefert Schmidt [Sch04] eine weitere Beschreibung der Anfrageoptimierung in GOODAC unter besonderer Berücksichtigung der Erzeugung textueller Ausführungspläne.

5.3.1 Einbindung von EGO in GOODAC

Um EGO als Komponente zur Anfrageoptimierung im Rahmen der Anfragebearbeitung in GOODAC zu verwenden, muss GOODAC im Wesentlichen nur eine Voraussetzung erfüllen: Sowohl vom Benutzer gestellte OOGQL-Anfragen als auch textuelle Ausführungspläne müssen eine algebraische Repräsentation besitzen, die sich zudem sowohl textuell als auch objektbasiert darstellen lässt. Diese algebraische Repräsentation wurde bereits in Kapitel 4 näher beschrieben, sodass an dieser Stelle nur die Einbindung von EGO in GOODAC thematisiert wird.

Algebraausdrücke

Algebraausdrücke können in GOODAC durch eine objektbasierte Struktur dargestellt werden (siehe auch Abschnitt 4.3.3). EGO stellt keine Anforderungen an diese Klassen, da Algebraausdrucksobjekte vor ihrer Verwendung im Optimierungsprozess in das interne generische Format von EGO konvertiert werden, sodass EGO nur auf dieser internen Repräsentation arbeitet.

Im Rahmen des Optimierungsprozesses von Algebraausdrücken werden keine Typen dieser für GOODAC spezifischen Algebraausdrücke berücksichtigt. Die Typinformationen, die in der objektbasierten Darstellung von für GOODAC spezifischen Algebraausdrücken existieren, werden also nicht auf das interne

generische Format von EGO übertragen. Zudem lässt sich zu jedem Algebraausdruck in GOODAC problemlos sein zugehöriger Typ bestimmen, sodass Algebratypen an dieser Stelle nicht näher betrachtet werden. Im Rahmen der Anfrageoptimierung von für GOODAC spezifischen Algebraausdrücken werden somit nur die eigentlichen Algebraausdrücke ohne ihre Typen berücksichtigt.

Konvertierung von Algebraausdrücken

Um Algebraausdrücke aus GOODAC in das interne generische Format konvertieren zu können, wurde eine neue Klasse `GOODACEGOExpressionConverter` als konkrete Unterklasse zu `ExpressionConverter` (siehe auch die Beschreibung von Abbildung 5.10) entwickelt [Sch04, 4]. Die dort implementierten Methoden sorgen dafür, dass sich Algebraausdrücke in der objektbasierten Darstellung aus GOODAC in das interne generische Format konvertieren lassen. Ebenso lassen sich zu Algebraausdrücken im internen generischen Format Eigenschaftswerte bestimmen und die entsprechenden konkreten Algebraausdrücke der deskriptiven und ausführbaren Algebra für GOODAC erzeugen.

Dabei stützt sich die Implementation der Methoden `convertStringListToEGOAlgebraExpression` und `convertConcreteAlgebraExpressionToEGOAlgebraExpression` im Wesentlichen auf entsprechende Methoden der einzelnen Unterklassen von `AlgebraThings` ab, die eine passende Konvertierung der zugehörigen Instanzen gewährleisten. Durch diese Verwendung des dynamischen Bindens wird sichergestellt, dass beim Hinzufügen weiterer Algebraausdrucksklassen oder bei der Modifikation der bestehenden Klassenhierarchie keine Änderungen an diesen Methoden der Klasse `GOODACEGOExpressionConverter` erforderlich sind.

Die Methode `convertEGOAlgebraExpressionToConcreteAlgebraExpression` kann kein dynamisches Binden verwenden, da sich die von EGO bereitgestellten Klassen zur Repräsentation generischer Algebraausdrücke weder modifizieren noch erweiteren lassen. Aus diesem Grund muss diese Methode einen übergebenen generischen Algebraausdruck direkt auswerten und für die Umwandlung in einen für GOODAC spezifischen objektbasierten Algebraausdruck sorgen.

Optimierungsregeln

Schließlich sind noch Optimierungsregeln in textueller Darstellung erforderlich, die für eine Optimierung von Anfragen sorgen. Diese Optimierungsregeln beinhalten neben der Beschreibung der anzuwendenden Optimierungsstrategie auch für GOODAC spezifische Algebraausdrücke in ihrer textuellen Darstellung. Wie bereits oben beschrieben, werden die Optimierungsregeln im Vorfeld des Optimierungsprozesses von EGO eingelesen. Hierbei wird vor allem eine interne Darstellung dieser bisher nur textuell definierten Optimierungsregeln erzeugt, die insbesondere Algebraausdrücke enthält, die zuvor durch die Methode `convertStringListToEGOAlgebraExpression` der Klasse `GOODACEGOExpressionConverter` in das interne generische Format übersetzt wurden.

5.3.2 Verwendete Optimierungsstrategie

Die während der Anfrageoptimierung in GOODAC zum Einsatz kommende Optimierungsstrategie ist sehr einfach aufgebaut. Im Wesentlichen besteht das Ziel dieser Strategie darin, deskriptive Algebraausdrücke – die OOGQL-Anfragen repräsentieren – auf ausführbare Algebraausdrücke – zur Repräsentation textueller Ausführungspläne – abzubilden. Zudem werden Selektionen möglichst früh ausgeführt, weil sich durch die Berücksichtigung dieser Heuristik die geschätzte Ausführungszeit verringert. Außerdem werden ein Kreuzprodukt und eine direkt nachfolgende Selektion zu einer Verbundoperation zusammengefasst. Schließlich findet eine Betrachtung von Alternativen, um die geschätzte Ausführungszeit zu verringern statt. Schmidt [Sch04, 5] beschreibt die verwendete Optimierungsstrategie im Rahmen seiner Arbeit und formuliert sie zudem durch Optimierungsregeln unter Verwendung von EGO.

Für den einfachen Aufbau der in GOODAC eingesetzten Optimierungsstrategie gibt es im Wesentlichen zwei Gründe. Zum einen handelt es sich bei GOODAC – wie bereits oben beschrieben – nur um ein GIS-Kernsystem, sodass keine für alle möglichen Anwendungsbereiche von GOODAC optimale Strategie zur Optimierung von Anfragen entwickelt werden kann. Zum anderen stellt das Gebiet der Entwicklung und Analyse möglicher Optimierungsstrategien ein sehr weites Feld dar, dessen Betrachtung deutlich über den Rahmen dieser Arbeit hinausgehen müsste, um eine zufriedenstellende Einschätzung zu gewährleisten. Insbesondere gestaltet sich die Spezifikation einer geeigneten Optimierungsstrategie wesentlich schwieriger als ihre eigentliche Umsetzung in einem Datenbankmanagementsystem. Allerdings unterstützt GOODAC

einen Systemprogrammierer bei der Spezifikation und der Ermittlung einer passenden Optimierungsstrategie durch die Verwendung von EGO und die damit verbundene einfache Darstellung durch Optimierungsregeln, die sich zudem leicht modifizieren lassen. Dadurch wird auch eine leichte Anpassung der Optimierungsstrategie im Fall einer Erweiterung von GOODAC sowie bei einem Einsatz von GOODAC in einem konkreten Anwendungsprogramm sichergestellt (siehe auch Abschnitt 6.2).

5.3.3 Beispiele für die Funktionsweise der Anfrageoptimierung in GOODAC

In diesem Abschnitt werden nun einige Beispiele für den Ablauf der Anfrageoptimierung in GOODAC gegeben. Diese zeigen einige Möglichkeiten für Systemprogrammierer, um ausgehend von einer für GOODAC spezifischen Konvertierungskomponente für Algebraausdrücke (siehe auch Abschnitt 5.3.1) eine Optimierungsstrategie durch textuelle Optimierungsregeln zu formulieren. Im Rahmen der nachfolgenden Beschreibung wird weitgehend auf eine Darstellung der Implementation sowie der beteiligten Komponenten verzichtet, um die Verwendung der textuellen Optimierungsregeln nach erfolgter Bereitstellung der für GOODAC spezifischen Konvertierungskomponente für Algebraausdrücke (siehe auch Abschnitt 5.3.1) vereinfacht zu erläutern.

Innerhalb von Algebraausdrücken, die in diesen Optimierungsregeln auftreten, kommen so genannte *Platzhalter* zum Einsatz. Diese repräsentieren konkrete Algebraausdrücke eines bestimmten Typs, um eine Optimierungsregel für verschiedene konkrete Algebraausdrücke anwendbar zu machen. Dieses Konzept der Platzhalter für Algebraausdrücke wird daher im Folgenden zuerst eingeführt. Bei diesen Erläuterungen wird zur Vereinfachung der Darstellung nicht mehr auf die interne Repräsentation von Algebraausdrücken eingegangen. Stattdessen werden alle auftretenden Algebraausdrücke im Folgenden nur noch in ihrer textuellen Darstellung gezeigt, obwohl intern hauptsächlich objektbasierte Algebraausdrücke im für GOODAC spezifischen Format oder im internen generischen Format von EGO zum Einsatz kommen. Die Beschreibung dieser Objekthierarchien würde jedoch nicht dem weiteren Verständnis der vorliegenden Arbeit dienen.

Platzhalter

In Kapitel 4 wurden die in GOODAC verwendeten Algebren und zugehörige Ausdrücke zur Repräsentation von OOGQL-Anfragen und textuellen Ausführungsplänen vorgestellt. Derartige Algebraausdrücke kommen auch innerhalb der Optimierungsregeln von GOODAC zum Einsatz, um die Transformation dieser Algebraausdrücke im Rahmen des Optimierungsprozesses darstellen zu können. Allerdings ist es dort nicht sinnvoll, nur Algebraausdrücke in der aus Kapitel 4 bekannten Form – so genannte *Algebraausdrücke ohne Platzhalter* – zu verwenden.

Soll beispielsweise eine Regel `exchangeOperandsInProduct` entwickelt werden, die für eine Vertauschung der beiden Operanden des Operators **product** sorgt, so sind genaue Kenntnisse dieser Operanden zum Definitionszeitpunkt der Optimierungsregel weder gegeben noch erforderlich. Einzig und allein das Auftreten dieses Operators mit seinen beiden Operanden – zwei Objektströmen – ist von Interesse. Daher wurden die Algebren dahingehend ergänzt, dass Platzhalter in Algebraausdrücken zugelassen werden. So repräsentieren beispielsweise Ausdrücke der Form *streamA* – angedeutet durch den Präfix *stream* – einen Strom von Objekten. Werden Sie in einem Musters innerhalb einer Optimierungsregel verwendet, so wird der passende Teil eines Algebraausdrucks ohne Platzhalter an einen solchen Platzhalter gebunden und in den Regelkontext eingefügt. Treten Platzhalter außerhalb eines Musters auf, so wird an ihrer Stelle der im Regelkontext gespeicherte Algebraausdruck ohne Platzhalter während der Ausführung der Optimierungsregel verwendet.

Dieses Konzept soll im Folgenden ausgehend von der in Beispiel 5.4 gezeigten Optimierungsregel `exchangeOperandsInProduct` und dem in Beispiel 5.5 gezeigten deskriptiven Algebraausdruck verdeutlicht werden. Die Optimierungsregel `exchangeOperandsInProduct` besteht im Wesentlichen nur aus einem Muster – einem deskriptiven Algebraausdruck – und einer Rückgabeanweisung, die einen gegenüber dem Muster leicht veränderten Algebraausdruck enthält. Diese beiden Algebraausdrücke enthalten jeweils die Platzhalter *streamA* und *streamB* zur Repräsentation von zwei Objektströmen, allerdings in einer unterschiedlichen Reihenfolge. Wird nun diese Optimierungsregel für einen Algebraausdruck ohne Platzhalter angewendet, so wird beim Vergleich auf Übereinstimmung mit dem Muster der vorliegenden Optimierungsregel überprüft, ob der übergebene Algebraaausdruck aus einem Operator **product** sowie zwei Operanden, die jeweils einen Objektstrom repräsentieren, besteht. In diesem Fall kommt die Regel zur Anwendung und die beiden Objektströme werden unter den Bezeichnern *streamA* und *streamB* in den aktuellen Regelkontext eingefügt. Als einzige Anweisung tritt nachfolgend bereits die Rückgabeanweisung auf, die einen

Beispiel 5.4 Eine Optimierungsregel zur Vertauschung der Operanden des Operators **product**

```
RULE
exchangeOperandsInProduct : Expr
{streamA streamB product} : Expr;
return {streamB streamA product};
```

Beispiel 5.5 Ein deskriptiver Algebraausdruck zur Erzeugung eines Kreuzprodukts

```
City("ci")
Castle("ca") select [fun(x: objectTuple(<("ca", oid("Castle"))>)) applyobjmethod("getCity", <>, ca(x))]
    product
```

Beispiel 5.6 Resultat einer Anwendung der in Beispiel 5.4 gezeigten Optimierungsregel

```
Castle("ca") select [fun(x: objectTuple(<("ca", oid("Castle"))>)) applyobjmethod("getCity", <>, ca(x))]
City("ci")
    product
```

neuen Algebraausdruck zurückliefert, bei dem diese Operanden vertauscht sind. Anstelle der beiden Platzhalter *streamA* und *streamB* enthält dieser neue Algebraausdruck die zuvor in den Regelkontext eingefügten Operanden.

Die Anwendung dieser Optimierungsregel `exchangeOperandsInProduct` lässt sich auch ausgehend von dem in Beispiel 5.5 gezeigten deskriptiven Algebraausdruck beschreiben. Dieser repräsentiert eine Anfrage, die das Kreuzprodukt von in der Datenbank gespeicherten Objekten des Typs `City` sowie von Ergebnissen des Methodenaufrufs `getCity` der in der Datenbank gespeicherten Instanzen der Klasse `Castle` liefert. Wird nun die Optimierungsregel `exchangeOperandsInProduct` auf diesen Algebraausdruck angewendet, so wird eine Übereinstimmung zwischen dem Muster dieser Optimierungsregel und dem Algebraausdruck festgestellt. Daher wird der Teil des Algebraausdrucks, der dem ersten Operanden des Operators **product** und damit der ersten Zeile in Beispiel 5.5 entspricht, unter dem Bezeichner *streamA* in den Regelkontext eingefügt. Ebenso wird der zweite Operand, der der zweiten Zeile in Beispiel 5.5 entspricht, unter dem Bezeichner *streamB* in den Regelkontext aufgenommen. Bei Ausführung der nachfolgenden Rückgabeanweisung und der damit verbundenen Erzeugung eines neuen Algebraausdrucks werden diese Teilausdrücke über ihre Bezeichner aus dem Regelkontext ausgelesen, sodass schließlich der in Beispiel 5.6 gezeigte Algebraausdruck als Ergebnis der Regelanwendung zurückgegeben wird.

Neben diesen Platzhaltern für Objektströme existieren weitere Platzhalter für andere Ausdrücke und Elemente der deskriptiven und ausführbaren Algebra. So stehen etwa Platzhalter mit dem Präfix *fun* für Funktionen, während der Präfix *ref* eine Referenz auf ein Objekt andeutet. Eine vollständige Übersicht der verfügbaren Platzhalter findet sich bei Schmidt [Sch04, 4], sodass im Rahmen dieser Arbeit nur diejenigen Platzhalter näher erläutert werden, die in den hier vorgestellten Optimierungsregeln auftreten.

Definition der Optimierungsregeln

Nun sollen exemplarisch einige in GOODAC zum Einsatz kommende Optimierungsregeln sowie deren Anwendung im Optimierungsprozess beschrieben werden. Dazu werden in Beispiel 5.7 und in Beispiel 5.8 diejenigen Optimierungsregeln gezeigt, die erforderlich sind, um den deskriptiven Algebraausdruck aus Beispiel 4.9 in einen ausführbaren Algebraausdruck zu überführen, der den in Beispiel 3.19 dargestellten textuellen Ausführungsplan repräsentiert. Der deskriptive Algebraausdruck aus Beispiel 4.9 entsteht dabei durch Verarbeitung der in Beispiel 3.18 gezeigten OOGQL-Anfrage durch den OOGQL-Parser (siehe auch Abschnitt 2.3). Die Übersetzung von deskriptiven Algebraausdrücken in entsprechende ausführbare Algebraausdrücke ergibt sich schon aus der Darstellung der beiden Algebren in Kapitel 4, sodass diese Transformation später nur noch exemplarisch beschrieben, aber nicht mehr motiviert wird.

Die Optimierungsregeln sind in einer Regelgruppe `someRuleForGOODAC` zusammengefasst. Deswegen wird beim Einlesen dieser Regeln eine Kollektion gleichen Namens erzeugt, auf die während der Ausführung jeder Optimierungsregel zugegriffen werden kann (siehe auch Abschnitt 5.2.1). Innerhalb der hier gezeigten Optimierungsregeln werden andere Regeln jedoch nur über die Angabe ihres Namens aufgerufen, sodass diese Kollektion hier nicht zum Einsatz kommt. Im Folgenden werden zuerst alle Regeln vorgestellt und erläutet, bevor ihre exemplarische Anwendung auf den bereits zuvor erwähnten Algebraausdruck gezeigt wird. Während dieser exemplarischen Anwendung wird unter anderem ausführlich auf die

Beispiel 5.7 Ausgewählte Optimierungsregeln für die Anfrageoptimierung in GOODAC, Teil 1

```
GROUP
someRulesForGOODAC

RULE
refToExe : Expr
{streamToBag(streamA)} : Expr;
streamB : Expr = streamToExe({streamA});
return {streamToCollection(streamB, applyclassmethod("new", <>, "OOGDMBag"))};

RULE PROTECTED
moveSelection : Expr
{streamA streamB product select [fun(variableA: objectTuple(<(identA, refA), (identB, refB)>))
                                                    applyobjmethod("and", <refC>, refD)]}: Expr;
streamC : Expr = {streamA select [fun(variableA: objectTuple(<(identA, refA)>)) refC]};
streamD : Expr = {streamA select [fun(variableA: objectTuple(<(identA, refA)>)) refD]};
if (streamC.getProperty(isValid)) then
    return {streamC streamB product
                select [fun(variableA: objectTuple(<(identA, variableSOSrefA),(identB, refB)>)) refD]};
else
    if (streamD.getProperty(isValid)) then
        return {streamD streamB product
                        select [fun(variableA: objectTuple(<(identA, refA), (identB, refB)>)) refC]};
    endif;
endif;

RULE PROTECTED
exchangeOperandsInProductSelection : Expr
{streamA streamB product
                    select [fun(variableA: objectTuple(<(identA, refA), (identB, refB)>)) refC]} : Expr;
return {streamB streamA product
                        select [fun(variableA: objectTuple(<(identB, refB), (identA, refA)>)) refC]};

RULE
streamToExe : Expr
{streamA streamB product
                select [fun(variableA: objectTupleA) applyobjmethod("and", <refC>, refD)]} : Expr;
streamC : Expr = {streamA streamB product
                        select [fun(variableA: objectTupleA) applyobjmethod("and", <refC>, refD)]};
if (streamD : Expr = moveSelection(streamC)) then
    streamE : Expr = exchangeOperandsInProductSelection(streamD);
else
    streamE : Expr = exchangeOperandsInProductSelection(streamC);
endif;
if (streamF : Expr = moveSelection(streamE)) then
    streamG : Expr = exchangeOperandsInProductSelection(streamF);
else
    streamG : Expr = exchangeOperandsInProductSelection(streamE);
endif;
if (streamG.isEqual(streamC)) then
    streamH : Expr = streamToExe({streamA streamB
                        join [fun(variableA: objectTupleA) applyobjmethod("and", <refC>, refD)]});
else
    streamH : Expr = streamToExe(streamG);
endif;
return streamH;
```

Beispiel 5.8 Ausgewählte Optimierungsregeln für die Anfrageoptimierung in GOODAC, Teil 2

RULE
streamToExe : Expr
{*streamA streamB* **product select** [*funA*]} : Expr;
streamC : Expr = streamToExe({*streamA streamB* **join** [*funA*]});
return *streamC*;

RULE
streamToExe : Expr
{*streamA streamB* **join** [*funA*]} : Expr;
streamC : Expr = streamToExe({*streamA*});
streamD : Expr = streamToExe({*streamB*});
return {*streamC streamD* **nestedLoopJoin** [*funA*]};

RULE
streamToExe : Expr
{*streamA* **select** [*funA*]} : Expr;
streamB : Expr = streamToExe({*streamA*});
return {*streamB* **select** [*funA*]};

RULE
streamToExe : Expr
{*identifierA* (*identA*)} : Expr;
c : Coll<Expr> = PrimitiveRetrieveIndices({*identifierA*});
[*indexA*] : Coll<Expr> = c.get(1);
return {*indexA identA* **scan** [**fun**(*x*: *objectTuple*(<(*identA*, *oid*("*identifierA*"))>)) **refBoolOp**(true)]};

ENDGROUP

Verwendung der oben eingeführten Platzhalter eingegangen.

Die erste Regel mit dem Namen `refToExe` liefert einen Algebraausdruck – verdeutlicht durch die Typangabe `Expr` hinter dem Regelnamen – zurück. Sie sorgt nämlich für eine Übersetzung eines deskriptiven Algebraausdrucks, der den Operator **streamToBag** verwendet, in einen entsprechenden Ausdruck der ausführbaren Algebra, in dem der Operator **streamToCollection** eingesetzt wird. Zugleich wird eine entsprechende Übersetzung des auftretenden Operanden angestoßen. Dazu besitzt diese Optimierungsregel den Algebraausdruck

streamToBag(*streamA*)

als Muster. Sie kann also für alle Algebraausdrücke ohne Platzhalter zum Einsatz kommen, die den Operator **streamToBag** auf einen beliebigen Objektstrom anwenden. Im Fall eines Aufrufs dieser Optimierungsregel für einen derartigen Algebraausdruck wird genau dieser Objektstrom dieses Algebraausdrucks unter dem Bezeichner *streamA* in den Regelkontext eingefügt. Bereits in der nächsten Anweisung

streamB : Expr = streamToExe({*streamA*});

kommt dieser Platzhalter zum Tragen. Dort wird nämlich für den unter *streamA* im Regelkontext gespeicherten Algebraausdruck ohne Platzhalter eine Optimierungsregel mit dem Namen `streamToExe` aufgerufen. Der zurückgelieferte Algebraausdruck gehört aufgrund der Definition der aufgerufenen Regel zur ausführbaren Algebra und wird anschließend unter dem Bezeichner *streamB* in den Regelkontext eingefügt. Dieser kommt in der abschließenden Rückgabeanweisung zum Einsatz. Der dort genannte Algebraausdruck

streamToCollection(*streamB*, **applyclassmethod**("new", <>, "OOGDMBag"))

verwendet diesen Bezeichner, um einen Ausdruck der ausführbaren Algebra zu erzeugen, der den bereits in diese Algebra übersetzten Operanden über den Bezeichner *streamB* aus dem Regelkontext ausliest und als erstes Argument für den gezeigten Operator **streamToCollection** der ausführbaren Algebra verwendet. Dessen zweites Argument gibt den Typ der zu verwendenden Kollektion an, indem es eine Referenz auf ein Objekt vom Typ `OOGDMBag` repräsentiert.

Die nachfolgende Optimierungsregel `moveSelection` kann nur über ihren Namen aufgerufen werden, weil sie durch die Verwendung des Schlüsselworts `PROTECTED` als geschützt gegen anonymes Aufrufen – zum Beispiel durch Auslesen aus einer Kollektion – gekennzeichnet ist. Diese Regel liefert ebenfalls einen Algebraausdruck zurück. Ihr Ziel liegt in dem Vorziehen eines Selektionsprädikats. Wird nämlich eine Selektion nicht auf dem Ergebnis eines Kreuzprodukts, sondern bereits auf einem der Operanden durchgeführt, lässt sich die Zeit zur Berechnung des Anfrageresultats im Allgemeinen stark reduzieren, weil durch dieses Vorgehen die Anzahl der Elemente im Ergebnisstrom des Kreuzprodukts gewöhnlich deutlich verringert wird [SKS02, 14.3.1]. Dieses Vorgehen wird hier aus Platzgründen nur für über den booleschen Operator `and` verbundene Selektionsprädikate gezeigt, weil dieses für die später exemplarisch untersuchte Optimierung des in Beispiel 4.9 dargestelltenx Algebraausdrucks erforderlich ist. Weiterhin wird nur versucht, ein Selektionsprädikat auf den ersten Operanden anzuwenden. Der symmetrische Fall muss durch eine übergeordnete Optimierungsregel behandelt werden.

Die Optimierungsregel `moveSelection` besitzt mit

streamA streamB **product**
  select [**fun**(*variableA*: *objectTuple*(‹(*identA, refA*), (*identB, refB*)›))
    applyobjmethod(“and”, ‹*refC*›, *refD*)] : Expr

also ein Muster, das mit einem Algebraausdruck ohne Platzhalter übereinstimmt, der aus einem Kreuzprodukt mit zwei Operanden besteht, auf das eine Selektion folgt, deren Prädikat zwei Objektreferenzen mittels `and` miteinander verknüpft. Hier treten viele Platzhalter auf, damit die als Operanden dienenden Objektströme *streamA* und *streamB*, die Komponenten des Objekttupels (*identA*, *refA*) und (*identB, refB*), der zugehörige Bezeichner *variableA* sowie die Referenzen *refC* und *refD* an die entsprechenden Teile eines Algebraausdrucks ohne Platzhalter gebunden und in den Regelkontext eingefügt werden können. In den folgenden beiden Zeilen dieser Optimierungsregel wird durch

streamC : Expr = {*streamA* **select** [**fun**(*variableA*: *objectTuple*(‹(*identA, refA*)›)) *refC*]};
streamD : Expr = {*streamA* **select** [**fun**(*variableA*: *objectTuple*(‹(*identA, refA*)›)) *refD*]};

ein Operator **select** für den ersten Operanden des ursprünglichen Operators **product** eingeführt. Dieser erhält als Prädikat eine Funktion über dem passenden Objekttupel, das sich nun nur noch auf diesen ersten Operanden des Operators **product** beziehen darf. Daher war auch vorher die explizite Angabe des Aufbaus dieses Objekttupels erforderlich. Die Funktion verwendet zur Erzeugung der zurückzuliefernden Objektreferenz – bei Zuweisung an *streamC* – den unter dem Bezeichner *refC* im Regelkontext gespeicherten Algebraausdruck. Im zweiten Fall – bei Zuweisung an *streamD* – kommt der über *refD* auszulesende Algebraausdruck zum Tragen.

Es kann nun allerdings vorkommen, dass sich beispielsweise das durch *refC* dargestellte Prädikat auch auf den zweiten Operanden *streamB* des ursprünglichen Operators **product** bezieht. In diesem Fall kann die zuvor im unter *streamC* im Regelkontext abgelegten Algebraausdruck enthaltene Selektion nicht durchgeführt werden, weil für die Auswertung des Prädikats eben nicht der zweite Operand *streamB* zur Verfügung steht. Daher wird durch die Bedingung

streamC.getProperty(isValid)

der folgenden bedingten Verzweigung überprüft, ob der erzeugte Algebraausdruck gültig ist. Hierzu wird der Wert der Eigenschaft `isValid` für den unter *streamC* im Regelkontext gespeicherten Algebraausdruck ohne Platzhalter ausgewertet. Nur wenn dieser Eigenschaftswert `true` entspricht, kann die Selektion wie zuvor beschrieben vorgezogen werden. In diesem Fall wird der Algebraausdruck

streamC streamB **product**
  select [**fun**(*variableA*: *objectTuple*(‹(*identA, refA*), (*identB, refB*)›)) *refD*]

erzeugt und als Ergebnis der Regelanwendung zurückgegeben. Dieser enthält das durch *refC* beschriebene Selektionsprädikat bereits im neuen ersten Operanden *streamC* des Operators **product**. Die Korrektheit dieses Operanden wurde durch die zuvor beschriebene Verwendung der Eigenschaft `isValid` sichergestellt. Das nach der Bildung des Kreuzprodukts angewendete Selektionsprädikat besteht nur noch aus dem unter *refD* im Regelkontext verfügbaren Algebraausdruck, weil der zweite Teil des ursprünglichen Prädikats – also der unter *refC* im Regelkontext verfügbare Algebraausdruck – wie bereits zuvor erwähnt innerhalb des von *streamC* repräsentierten Algebraausdrucks ausgewertet wird.

Sollte der vorherige Test auf Korrektheit des durch den Bezeichner *streamC* repräsentierten Algebraausdrucks nicht erfolgreich verlaufen sein, so darf er nicht verwendet werden. Daher wird in diesem Fall der unter *streamD* im Regelkontext verfügbare Algebraausdruck untersucht und gegebenenfalls – analog zu dem zuvor für *streamC* beschriebenen Vorgehen – zur Erzeugung eines neuen Algebraausdrucks als Rückgabe der Regelanwendung verwendet. Ist jedoch *streamD* ebenfalls kein gültiger Algebraausdruck, so liefert die Anwendung der aktuellen Optimierungsregel `moveSelection` kein Ergebnis. Insbesondere wird diese Tatsache durch den Wert `false` für das implizite Rückgabeergebnis dieses Regelaufrufs gekennzeichnet.

Die nächste in Beispiel 5.7 gezeigte Regel `exchangeOperandsInProductSelection` ist ebenfalls gegen einen anonymen Aufruf geschützt. Sie dient der Vertauschung der beiden Operanden in einem Algebraausdruck, der im Wesentlichen aus einer Selektion auf dem Ergebnis eines Kreuzprodukts besteht. Dazu werden beim Aufruf dieser Regel für einen Algebraausdruck ohne Platzhalter die beiden Operanden des Operators **product** über das Muster

streamA streamB **product**
select [**fun**(*variableA*: *objectTuple*(‹(*identA, refA*), (*identB, refB*)›)) *refC*] : Expr

unter den Bezeichnern *streamA* und *streamB* in den aktuellen Regelkontext eingefügt; ebenso sind dort nach Überprüfung des Musters auf Übereinstimmung mit diesem übergebenen Algebraausdruck das Selektionsprädikat *refC* sowie die einzelnen Bestandteile des Objekttupels der dort auftretenden Funktion zu finden. Die nächste Anweisung innerhalb dieser Optimierungsregel erzeugt bereits einen neuen Algebraausdruck

streamB streamA **product**
select [**fun**(*variableA*: *objectTuple*(‹(*identB, refB*), (*identA, refA*)›)) *refC*]

als Rückgabe eines Regelaufrufs. Dabei reicht es nicht aus, nur die Reihenfolge der Operanden *streamA* und *streamB* in *streamB* und *streamA* zu verändern, stattdessen muss diese Modifikation auch innerhalb des Objekttupels der Funktion zur Darstellung des Selektionsprädikats berücksichtigt werden. Schließlich führt eine Vertauschung der Operanden des Operators **product** dazu, dass sich der Typ des Resultats dieser Operatoranwendung und damit auch der Typ des Operanden des Operators **select** ändert.

In der Definition der nun folgenden Optimierungsregel `streamToExe` werden die beiden zuvor beschriebenen Regeln `moveSelection` und `exchangeOperandsInProductSelection` verwendet. Diese Optimierungsregel `streamToExe` setzt die beiden anderen Regeln ein, um auch mehrere durch `and` miteinander verknüpfte Selektionsprädikate an einem Operator **product** vorbeizuschieben und auf jeweils einen Operanden dieses Operators **product** anzuwenden. Dazu erwartet diese Optimierungsregel mit

streamA streamB **product**
select [**fun**(*variableA*: *objectTupleA*) **applyobjmethod**(“and”, ‹*refC*›, *refD*)] : Expr

ein ähnliches Muster wie schon `moveSelection`. Allerdings ist hier der Aufbau des eingesetzten Objekttupels nicht weiter von Interesse, sodass es durch den Platzhalter *objectTupleA* dargestellt werden kann. Bei Anwendung dieser Optimierungsregel `streamToExe` auf einen Algebraausdruck werden also erneut die aufgeführten Platzhalter verwendet, um Teile des übergebenen Algebraausdrucks in den aktuellen Regelkontext einzufügen. Mit der ersten Anweisung dieser Regel

streamC : Expr = {*streamA streamB* **product**
select [**fun**(*variableA*: *objectTupleA*) **applyobjmethod**(“and”, ‹*refC*›, *refD*)]};

wird dieser übergebene Algebraausdruck an den Bezeichner *streamC* gebunden, um ihn im Folgenden leicht verwenden zu können und den weiteren Aufbau der Regel übersichtlich zu gestalten. In der nun folgenden bedingten Verzweigung wird über die Bedingung

streamD : Expr = moveSelection(*streamC*)

festgestellt, ob eine Anwendung der Regel `moveSelection` auf diesen Algebraausdruck *streamC* erfolgreich verlaufen ist. Ein etwaiges Resultat wird unter dem Bezeichner *streamD* in den Regelkontext eingefügt. Die Regel `moveSelection` versucht – wie oben beschrieben –, einen Teil des Selektionsprädikats bereits auf den ersten Operanden des Kreuzprodukts anzuwenden. Im Erfolgsfall wird dieses Ergebnis verwendet, um durch die nachfolgende Anweisung

streamE : Expr = exchangeOperandsInProductSelection(*streamD*);

die Operanden des Operators **product** zu vertauschen, um danach zu versuchen, weitere Teile des Selektionsprädikat auf den anderen Operanden des Kreuzprodukts verlagern zu können. War der erste Versuch der Anwendung von `moveSelection` nicht erfolgreich, konnte auch keine Rückgabe unter *streamD* im Regelkontext gespeichert werden, sodass in diesem Fall mit Hilfe von

streamE : Expr = exchangeOperandsInProductSelection(*streamC*);

die Operanden des Kreuzprodukts im ursprünglichen Algebraausdruck vertauscht werden müssen.

Anschließend erfolgt ein weiterer Aufruf der Optimierungsregel `moveSelection`, sodass gegebenenfalls ein weiterer Teil des Selektionsprädikats bereits auf den ursprünglich zweiten Operanden des Operators **product** angewendet werden kann. Im Ergebnis dieser Regelanwendung wird im Erfolgsfall analog zum ersten Aufruf von `moveSelection` erneut eine Vertauschung der Operanden des Produktoperators vorgenommen. Andernfalls erfolgt die Vertauschung der Operanden im an die Optimierungsregel `moveSelection` übergebenen Algebraausdruck *streamE*. Anschließend enthält *streamG* die Operanden des Operators **product** in ihrer urspünglichen Reihenfolge, sodass der Typ des Resultatstroms des umgeformten Algebraausdrucks dem Typ des zu Beginn der Regelanwendung gegebenen Algebraausdruck *streamC* entspricht. Dadurch besitzen die durchgeführten Umformungen nur lokale Auswirkungen und müssen bei Verwendung des aktuellen Algebraausdrucks als Operand eines weiteren Operators nicht berücksichtigt werden. Zudem wurde bisher versucht, einen Teil des aktuellen Selektionsprädikats sowohl ausschließlich auf den ersten als auch nur auf den zweiten Operanden des Operators **product** anzuwenden. Falls dieses zu keiner Veränderung des ursprünglichen Algebraausdrucks geführt hat, wird die nun folgende Bedingung

streamG.isEqual(*streamC*)

wahr. Es erfolgt dann ein erneuter Aufruf einer Regel mit dem Namen `streamToExe`; allerdings wird durch die vorherige Umformung des derzeitigen Algebraausdrucks zu

streamA streamB **join** [**fun**(*variableA*: *objectTupleA*) **applyobjmethod**("and", ‹*refC*›, *refD*)]

erreicht, dass eine andere Regel mit diesem Namen aufgerufen wird. Die Verwendung einer Kreuzprodukts und einer nachfolgenden Selektion – dargestellt durch die Operatoren **product** und **select** – wird nämlich zu einer Verbundoperation – also eines Einsatzes des Operators **join** – vereinfacht. Der vorliegende Aufruf einer Regel namens `streamToExe` führt also nicht zur Anwendung der aktuellen Optimierungsregel, weil deren Muster eine Verwendung der Operatoren **product** und **select** fordert. Eine passende Regel ist daher in Beispiel 5.8 zu sehen und wird weiter unten beschrieben.

Führt die Auswertung der zuvor erwähnten Bedingung

streamG.isEqual(*streamC*)

zu dem Ergebnis, dass *streamG* nicht dem ursprünglichen Algebraausdruck *streamC* entspricht, wird die aktuelle Regel über

streamH : Expr = streamToExe(*streamG*);

für das aktuelle Zwischenergebnis *streamG* gegebenenfalls erneut aufgerufen, um nach Möglichkeit weitere Teilprädikate der bereits untersuchten Selektion vor dem Kreuzprodukt für einen der beiden Objektströme *streamA* und *streamB* anwenden zu können. Abschließend wird das Ergebnis der aktuellen Regelanwendung – in jedem Fall ein unter dem Bezeichner *streamG* im aktuellen Regelkontext gespeicherter Algebraausdruck – über die Rückgabeanweisung

return *streamH*;

an die aufrufende Optimierungsregel zurückgeliefert. Damit sind insbesondere möglichst viele Teilprädikate der Selektion bereits vor die Erzeugung des Kreuzprodukts geschoben worden, sodass nun keine weitere Umformung des Resultats in einen anderen deskriptiven Algebraausdruck erforderlich ist. Stattdessen muss der zurückgelieferte Algebraausdruck in einen Ausdruck der ausführbaren Algebra übersetzt werden. Dabei helfen die in Beispiel 5.8 gezeigten Optimierungsregeln, deren Aufbau im Folgenden erläutert wird. Sie enthalten im Wesentlichen nur noch Transformationsvorschriften zur Überführung eines Ausdrucks der deskriptiven in die ausführbare Algebra.

Die vier in Beispiel 5.8 dargestellten Optimierungsregeln tragen alle den gleichen Namen `streamToExe` wie die zuvor erläuterte Optimierungsregel. Dieser Name verdeutlicht, dass die zugehörige Regel einen Ausdruck der deskriptiven Algebra, der einen Objektstrom repräsentiert, in einen Ausdruck der ausführbaren

Algebra übersetzt, der ebenfalls einen Strom von Objekten darstellt. Die Verwendung mehrerer gleich benannter Regeln führt beim Aufruf einer Regel dieses Namens dazu, dass ausgehend von dem übergebenen Algebraausdruck ohne Platzhalter die erste der fünf Regeln dieses Namens ausgewählt wird, deren Muster mit diesem Algebraausdruck übereinstimmt (siehe auch die Beschreibung der Anwendung von Optimierungsregeln in Abschnitt 5.2.3).

Die erste dieser in Beispiel 5.8 dargestellten Regeln dient der Zusammenfassung der Berechnung eines Kreuzprodukts mit anschließender Selektion zu einer Verbundoperation. Daher besitzt diese Regel

streamA streamB **product select** [*funA*] : Expr

als Muster, sodass für einen übergebenen Algebraausdruck die Operanden des Operators **product** an die Platzhalter *streamA* und *streamB* gebunden sowie anschließend unter diesen Bezeichnern in den Regelkontext eingefügt werden. Das Selektionsprädikat wird ebenfalls an einen Platzhalter – nämlich *funA* – gebunden und kann anschließend unter diesem Bezeichner aus dem Regelkontext ausgelesen werden. Die nächste Zeile enthält den Aufruf einer weiteren Regel `streamToExe`, der allerdings den Algebraausdruck

streamA streamB **join** [*funA*]

als Argument verwendet. Hier wird also bereits die Verbundoperation eingesetzt, wobei die übrigen Elemente des Algebraausdrucks aus dem Regelkontext ausgelesen werden. Die eigentliche Regelanwendung dient dazu, diesen neuen Ausdruck der deskriptiven Algebra in einen Ausdruck der ausführbaren Algebra zu überführen. Das Ergebnis dieser Regelanwendung wird daraufhin an den Bezeichner *streamC* gebunden und anschließend als Resultat der Ausführung der aktuellen Optimierungsregel zurückgegeben.

Die zweite in Beispiel 5.8 gezeigte Optimierungsregel übersetzt einen deskriptiven Algebraausdruck, der den Operator **join** enthält, unter Verwendung des Operators **nestedLoopJoin** in einen Ausdruck der ausführbaren Algebra. Das Muster

{*streamA streamB* **join** [*funA*]} : Expr

stellt sicher, dass ein erforderlicher deskriptiver Algebraausdruck übergeben wird und fügt die Operanden des Operators **join** unter den Bezeichern *streamA*, *streamB* und *funA* in den aktuellen Regelkontext ein. Anschließend werden durch

streamC : Expr = streamToExe({*streamA*});
streamD : Expr = streamToExe({*streamB*});

die ersten beiden Operanden in Ausdrücke der ausführbaren Algebra transformiert, bevor das Ergebnis den Bezeichnern *streamC* und *streamD* zugewiesen wird. Eine Übersetzung des dritten Operanden *funA* ist nicht erforderlich, weil dieser in jedem Fall auch ein gültiger Ausdruck der ausführbaren Algebra ist. Daher können diese drei Bezeichner in der nachfolgenden Rückgabeanweisung

return {*streamC streamD* **nestedLoopJoin** [*funA*]};

verwendet werden, um den gewünschten Ausdruck der ausführbaren Algebra zu erzeugen und anschließend als Ergebnis dieser Regelanwendung zurückzuliefern.

Die dritte Optimierungsregel aus Beispiel 5.8 ist sehr ähnlich zu der zuvor beschriebenen aufgebaut. Sie überführt einen Ausdruck der deskriptiven Algebra, der eine Selektion enthält, in einen entsprechenden Ausdruck der ausführbaren Algebra. Durch die Angabe des Musters

{*streamA* **select** [*funA*]} : Expr

werden der eingehende Objektstrom der Selektion sowie das Prädikat an die Bezeichner *streamA* und *funA* gebunden. Anschließend erfolgt ein Regelaufruf für den ersten Operanden, dessen Ergebnis unter *streamB* in den Regelkontext eingefügt wird, bevor der ausführbare Algebraausdruck

streamB **select** [*funA*]

als Ergebnis dieser Regelanwendung zurückgegeben wird.

Die letzte in Beispiel 5.8 dargestellte Optimierungsregel zeigt exemplarisch die Verwendung von einigen in der bisherigen Beschreibung der Optimierungsregeln für GOODAC noch nicht aufgetretenen Konzepten. Diese Regel ermöglicht die Übersetzung deskriptiver Algebraausdrücke, die den Zugriff auf in der Datenbank gespeicherte Objektmengen über die Angabe des zugehörigen Klassennamens symbolisieren, in einen Ausdruck der ausführbaren Algebra, der diesen Objektzugriff über das Auslesen einer entsprechenden Indexstruktur mit Hilfe des Operators **scan** ermöglicht. Daher erwartet das eingehende Muster

{*identifierA* (*identA*)} : Expr

die Angabe eines Klassennamens *identifierA* – also eine Instanziierung des Operators **name** – sowie einen Bezeichner *identA* für die eindeutige Identifizierung der zugehörigen Komponente im zu erzeugenden Objekstrom. Anschließend kommt im Rahmen der ersten Anweisung

c : Coll<Expr> = PrimitiveRetrieveIndices({*identifierA*});

das Primitiv `PrimitiveRetrieveIndices` zum Einsatz. Dieses speziell für GOODAC entwickelte Primitiv erwartet einen Klassennamen – hier also den unter *identifierA* im Regelkontext gespeicherte Bezeichner – und liefert als Ergebnis seines Aufrufs mit Hilfe eines Zugriffs auf den Systemkatalog eine Kollektion von Algebraausdrücken zurück. Diese Kollektion enthält eine algebraische Beschreibung aller verfügbaren Indexstrukturen für den Zugriff auf alle in der Datenbank gespeicherten Instanzen dieser Klasse. Diese Kollektion wird unter dem Bezeichner *c* in den Regelkontext eingefügt. In der nachfolgenden Anweisung

[*indexA*] : Coll<Expr> = c.get(1);

wird über `c.get(1)` das erste Element aus dieser Kollektion ausgelesen und dem Bezeichner *indexA* zugewiesen. Die Notation [*indexA*] ist hierbei erforderlich, weil die Methode `get` für Kollektionen in jedem Fall eine neue Kollektion – hier mit nur einem Element – erzeugt. Der durch den an den Bezeichner *indexA* gebundenen Algebraausdruck dargestellte Indexzugriff wird im Resultat

indexA identA **scan** [**fun**(*x*: *objectTuple*(⟨(*identA*, *oid*("*identifierA*"))⟩)) **refBoolOp**(true)]

dieser Regelanwendung als Operand des Operators **scan** verwendet. Die beiden übrigen zum Einsatz kommenden Operanden dieses Operators gewährleisten, dass alle in der Datenbank verfügbaren Objekte des durch den Bezeichner *identifierA* repräsentierten Typs aus der Datenbank ausgelesen und in einen Objektstrom eingefügt werden. Die zugehörige Komponente im Objekttupel dieses Objektstroms erhält den eindeutigen Bezeichner, der im aktuellen Regelkontext unter *identA* erreichbar ist.

Anwendung der Optimierungsregeln

Bisher wurden die Definitionen aller in Beispiel 5.7 und Beispiel 5.8 gezeigten Optimierungsregeln erläutert. Im Folgenden soll davon ausgehend die Anwendung der zuvor definierten Optimierungsregeln auf den in Beispiel 4.9 dargestellten deskriptiven Algebraausdruck beschrieben werden. Dazu wird als erstes die Optimierungsregel `refToExe` mit dem gesamten Algebraausdruck als Argument aufgerufen. Das dort eingesetzte Muster stimmt mit dem übergebenen Algebraausdruck überein, wobei der in Beispiel 5.9 dargestellte Algebraausdruck an den Platzhalter *streamA* gebunden und unter diesem Bezeichner im aktuellen Regelkontext gespeichert wird.

Anschließend erfolgt ein Aufruf einer Optimierungsregel mit dem Namen `streamToExe` für diesen im Regelkontext verfügbaren Algebraausdruck. Es stehen insgesamt fünf Optimierungsregeln mit diesem Namen zur Auswahl; schon die erste dieser Regeln – die letzte der in Beispiel 5.7 dargestellten – besitzt ein Muster, das dem übergebenen Algebraausdruck entspricht, weil dort im Wesentlichen das Auftreten des Operators **product** gefolgt vom Operator **select** mit einem Aufruf der Methode `and` für zwei Objektreferenzen im zugehörigen Selektionsprädikat gefordert wird. Also wird diese Regel auf den übergebenen Algebraausdruck angewendet.

Zum besseren Verständnis der nachfolgenden Beschreibung der Anwendung dieser Optimierungsregel `streamToExe` listet Tabelle 5.4 die einzelnen Platzhalter des zuvor erwähnten Musters sowie die daran gebundenen Teile des in Beispiel 5.9 gezeigten deskriptiven Algebraausdrucks auf. Auch im Rahmen der nachfolgenden Beschreibung werden zur Vereinfachung der Erläuterungen alle Algebraausdrücke grundsätzlich in ihrer textuellen Repräsentation dargestellt, obwohl bei der Anwendung von Optimierungsregeln vor allem die für GOODAC spezifische objektbasierte Darstellung sowie das interne generische Format von EGO zum Einsatz kommen.

Der ursprünglich übergebene Algebraausdruck – also der in Beispiel 5.9 dargestellte – wird anschließend von der ersten Anweisung der Regel `streamToExe` an den Bezeichner *streamC* gebunden. Dieser wird danach als Argument für den Aufruf der Regel `moveSelection` innerhalb der Bedingung der bedingten Verzweigung verwendet. Sollte diese Regelanwendung erfolgreich verlaufen, wird ihr Ergebnis unter dem Bezeichner *streamD* in den Regelkontext eingefügt.

Beim Aufruf der Regel `moveSelection` kommen nahzu die gleichen Platzhalter wie zuvor beim Aufruf der Regel `streamToExe` zum Einsatz. Im Muster von `moveSelection` wird nur das Objekttupel innerhalb

Beispiel 5.9 Der in `refToExe` an den Bezeichner *streamA* gebundene Algebraausdruck

City(“c1”)
City(“c2”)
product
select[
fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))>))
applyobjmethod(
“and”,
<**applyobjmethod**(
“and”,
<**applyobjmethod**(
“==”,
<**applyobjmethod**(“getInhabitants”, <>, **c2**(*x*))>,
applyobjmethod(“getInhabitants”, <>, **c1**(*x*)))>,
applyobjmethod(
“==”,
<**refStringOp**(“Bayern”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c2**(*x*)))))>,
applyobjmethod(
“==”,
<**refStringOp**(“NRW”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c1**(*x*)))))]

Tabelle 5.4 Ausgewählte Platzhalter beim Aufruf von `streamToExe` aus Beispiel 5.7

Platzhalter	Teil des Algebraausdrucks
streamA	**City**(“c1”)
streamB	**City**(“c2”)
variableA	*x*
objectTupleA	*objectTuple*(<(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))>)
refC	**applyobjmethod**(“and”, <**applyobjmethod**(“==”, <**applyobjmethod**(“getInhabitants”, <>, **c2**(*x*))>, **applyobjmethod**(“getInhabitants”, <>, **c1**(*x*)))>, **applyobjmethod**(“==”, <**refStringOp**(“Bayern”)>, **applyobjmethod**(“getName”, <>, **applyobjmethod**(“getFederalState”, <>, **c2**(*x*)))))
refD	**applyobjmethod**(“==”, <**refStringOp**(“NRW”)>, **applyobjmethod**(“getName”, <>, **applyobjmethod**(“getFederalState”, <>, **c1**(*x*))))

Tabelle 5.5 Ausgewählte Platzhalter beim ersten Aufruf von `moveSelection`

Platzhalter	Teil des Algebraausdrucks
identA	"c1"
refA	*oid*("City"))
identB	"c2"
refB	*oid*("City"))
streamC	**City**("c1") **select**[**fun**(*x*: *objectTuple*(‹("c1", *oid*("City"))›)) **applyobjmethod**("and", ‹**applyobjmethod**("==", ‹**applyobjmethod**("getInhabitants", ‹›, **c2**(*x*))›, **applyobjmethod**("getInhabitants", ‹›, **c1**(*x*)))›, **applyobjmethod**("==", ‹**refStringOp**("Bayern")›, **applyobjmethod**("getName", ‹›, **applyobjmethod**("getFederalState", ‹›, **c2**(*x*)))))]
streamD	**City**("c1") **select**[**fun**(*x*: *objectTuple*(‹("c1", *oid*("City"))›)) **applyobjmethod**("==", ‹**refStringOp**("NRW")›, **applyobjmethod**("getName", ‹›, **applyobjmethod**("getFederalState", ‹›, **c1**(*x*))))]

des Selektionsprädikats näher aufgeschlüsselt. Eine Übersicht der hierzu eingesetzten Bezeichner und der zugehörigen Teile des übergebenen Algebraausdrucks findet sich in Tabelle 5.5. Weiterhin sind dort die Ergebnisse der ersten beiden Anweisungen dieser Regel – jeweils eine Zuweisung neu gebildeter Algebraausdrücke an einen Bezeichner – zu sehen. In der nachfolgenden bedingten Verzweigung wird mit Hilfe der Eigenschaft `isValid` überprüft, ob *streamC* einen gültigen Algebraausdruck repräsentiert. Dieser Algebraausdruck ist zwar syntaktisch korrekt, allerdings verwendet er zweimal den Operator **c2** innerhalb des Selektionsprädikats und greift somit auf Elemente des ursprünglich zweiten Operanden *streamB* zu. Am Algebraausdruck lässt sich dieses leicht erkennen, weil `''c2''` keinen gültgen Bezeichner für eine Komponente des auftretenden Objekttupels darstellt. Aus diesem Grund entspricht der zurückgegebene Eigenschaftswert zur Eigenschaft `isValid` dem booleschen Wert `false`.

Es erfolgt also noch keine Rückgabe eines neuen Algebraausdrucks, stattdessen wird jetzt für *streamD* der Eigenschaftswert zur Eigenschaft `isValid` ermittelt. Dieses Mal ist der von *streamD* repräsentierte Algebraausdruck nicht nur syntaktisch korrekt, sondern verwendet mit **c1** auch ausschließlich Bezeichner für Komponenten eines Objektstroms, die explizit im auftretenden Objekttupel genannt werden. Somit entspricht der Eigenschaftswert dem booleschen Wert `true` und der durch

streamD streamB **product**
select [**fun**(*variableA*: *objectTuple*(‹(*identA*, *refA*), (*identB*, *refB*)›)) *refC*]

erzeugte Algebraausdruck ohne Platzhalter (siehe auch Beispiel 5.10) wird als Ergebnis dieser Regelanwendung an die aufrufende Regel `streamToExe` zurückgegeben.

Diese Regel setzt ihre Ausführung mit der Zuweisung dieses Resultats der Regelanwendung an den Bezeichner *streamD* innerhalb der Bedingung der bedingten Verzweigung fort. Weil die vorausgegangene Anwendung der Optimierungsregel `moveSelection` erfolgreich war, wird nun die Anweisung

Beispiel 5.10 Nach dem ersten Aufruf von `moveSelection` zurückgegebener Algebraausdruck

City(“c1”)
  select[
    fun(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”))⟩))
    applyobjmethod(
      “==”,
      ⟨**refStringOp**(“NRW”)⟩,
      applyobjmethod(
        “getName”,
        ⟨⟩,
        applyobjmethod(“getFederalState”, ⟨⟩, **c1**(*x*))))]
City(“c2”)
    product
      select[
        fun(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))⟩))
        applyobjmethod(
          “and”,
          ⟨**applyobjmethod**(
            “==”,
            ⟨**applyobjmethod**(“getInhabitants”, ⟨⟩, **c2**(*x*))⟩,
            applyobjmethod(“getInhabitants”, ⟨⟩, **c1**(*x*)))⟩,
          applyobjmethod(
            “==”,
            ⟨**refStringOp**(“Bayern”)⟩,
            applyobjmethod(
              “getName”,
              ⟨⟩,
              applyobjmethod(“getFederalState”, ⟨⟩, **c2**(*x*)))))]

streamE : Expr = exchangeOperandsInProductSelection(*streamD*);

ausgeführt. Diese Anwendung der Optimierungsregel `exchangeOperandsInProductSelection` auf den in Beispiel 5.10 gezeigten Algebraausdruck ist recht einfach strukturiert. Dort werden nur die Operanden des auftretenden Operators **product** sowie die entsprechenden Einträge im Objekttupel des Prädikats der nachfolgenden Selektion vertauscht, sodass diese Regelanwendung den in Beispiel 5.11 zu sehenden Algebraausdruck zurückliefert. Dieser wird anschließend unter dem Bezeichner *streamE* in den Regelkontext eingefügt.

Danach erfolgt für diesen unter *streamE* verfügbaren Algebraausdruck ein ähnlicher Aufruf der Regel `moveSelection` wie zuvor für den durch *streamC* repräsentierten Algebraausdruck. Falls diese Regelanwendung erfolgreich verläuft, wird das zurückgegebene Ergebnis – wiederum ein deskriptivener Algebraausdruck – dem Bezeichner *streamF* zugewiesen.

Die Überprüfung auf Übereinstimmung zwischen dem in Beispiel 5.11 dargestellten Algebraausdruck und dem Muster der Regel `moveSelection` verläuft erneut erfolgreich, sodass einige Teile dieses Algebraausdrucks an die im Muster verwendeten Platzhalter gebunden werden. Eine Übersicht über diese Beziehung zwischen Teilen des für die Anwendung der Optimierungsregel übergebenen Algebraausdrucks und den Platzhaltern wird in Tabelle 5.6 gezeigt.

Die ersten beiden Anweisungen der Optimierungsregel `moveSelection` erzeugen wie schon bei der oben beschriebenen ersten Anwendung dieser Optimierungsregel zwei neue Algebraausdrücke, die unter den Bezeichnern *streamC* und *streamD* in den aktuellen Regelkontext eingefügt werden. Sie repräsentieren zwei unterschiedliche Versuche, um jeweils einen Teil des im vorliegenden Algebraausdruck auftretenden Selektionsprädikats an der Verwendung des Operators **product** vorbeizuziehen, sodass eine Selektion mit diesem Teilprädikat bereits auf den ersten Operanden des Produktoperators angewendet wird. Das Ergebnis in Form der an die Bezeichner *streamC* und *streamD* gebundenen Algebraausdrücke wird in Tabelle 5.6 dargestellt.

Anschließend wird durch Auswertung der Bedingung

streamC.getProperty(isValid)

Beispiel 5.11 Erste Rückgabe von `exchangeOperandsInProductSelection`

City(“c2”)
City(“c1”)
select[
fun(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”))⟩))
applyobjmethod(
“==”,
⟨**refStringOp**(“NRW”)⟩,
applyobjmethod(
“getName”,
⟨⟩,
applyobjmethod(“getFederalState”, ⟨⟩, **c1**(*x*))))]
product
select[
fun(*x*: *objectTuple*(⟨(“c2”, *oid*(“City”)), (“c1”, *oid*(“City”))⟩))
applyobjmethod(
“and”,
⟨**applyobjmethod**(
“==”,
⟨**applyobjmethod**(“getInhabitants”, ⟨⟩, **c2**(*x*))⟩,
applyobjmethod(“getInhabitants”, ⟨⟩, **c1**(*x*)))⟩,
applyobjmethod(
“==”,
⟨**refStringOp**(“Bayern”)⟩,
applyobjmethod(
“getName”,
⟨⟩,
applyobjmethod(“getFederalState”, ⟨⟩, **c2**(*x*)))))]

überprüft, ob der unter *streamC* im Regelkontext abgelegte Algebraausdruck gültig ist. Hierbei liefert die Eigenschaft `isValid` einen Eigenschaftswert, der dem booleschen Wert `false` entspricht. Das liegt darin begründet, dass der untersuchte Algebraausdruck zwar syntaktisch korrekt ist, dass er allerdings durch den Operator **c1** einen eindeutigen Bezeichner zur Kennzeichnung von Teilen eines Objektstroms refenziert, der nicht im an *x* gebundenen Objekttupel auftritt. Aus diesem Grund ist also die Verwendung von **c1**(*x*) in diesem Algebraausdruck nicht erlaubt.

Also fährt die Anwendung der Optimierungsregel `moveSelection` mit der Überprüfung der Bedingung

streamD.getProperty(isValid)

fort. In diesem Fall wird der Eigenschaftswert `true` für die Eigenschaft `isValid` ermittelt, weil der durch *streamD* referenzierte deskriptive Algebraausdruck nicht nur syntaktisch korrekt ist, sondern auch nur zulässige Instanziierungen des Operators **name** einsetzt. Bei der einzigen auftretenden Instanziierung handelt es sich nämlich um **c2** im Teilausdruck **c2**(*x*), der sich somit auf ein gültiges Element des vom Bezeichner *x* repräsentierten Objekttupels bezieht. Somit kommt die nachfolgende Rückgabeanweisung zur Ausführung, die den Algebraausdruck

streamD streamB **product**
select [**fun**(*variableA*: *objectTuple*(⟨(*identA*, *refA*), (*identB*, *refB*)⟩)) *refC*]

als Ausgangspunkt zur Erzeugung des Endresultats verwendet. Dazu werden die Platzhalter gemäß der in Tabelle 5.6 gezeigten Zuordnung aufgelöst, sodass schließlich der in Beispiel 5.12 gezeigte Algebraausdruck das Resultat dieser Regelanwendung darstellt.

Im Anschluss fährt somit die ursprüngliche Regel `streamToExe` mit ihrer Ausführung fort. Sie weist dem Bezeichner *streamF* das von `moveSelection` erzeugte Ergebnis zu, bevor Sie die Anweisung

streamG : Expr = exchangeOperandsInProductSelection(*streamF*);

ausführt. Dadurch erfolgt ein Aufruf der Optimierungsregel `exchangeOperandsInProductSelection`. Durch diesen werden wie bereits bei der ersten Anwendung dieser Regel für den an *streamD* gebundenen

Tabelle 5.6 Ausgewählte Platzhalter beim zweiten Aufruf von `moveSelection`

Platzhalter	Teil des Algebraausdrucks
streamA	**City**(“c2”)
streamB	**City**(“c1”) **select**[**fun**(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”))⟩)) **applyobjmethod**(“==”, ⟨**refStringOp**(“NRW”)⟩, **applyobjmethod**(“getName”, ⟨⟩, **applyobjmethod**(“getFederalState”, ⟨⟩, **c1**(*x*))))]
variableA	*x*
identA	“c2”
refA	*oid*(“City”))
identB	“c1”
refB	*oid*(“City”))
refC	**applyobjmethod**(“==”, ⟨**applyobjmethod**(“getInhabitants”, ⟨⟩, **c2**(*x*))⟩, **applyobjmethod**(“getInhabitants”, ⟨⟩, **c1**(*x*)))
refD	**applyobjmethod**(“==”, ⟨**refStringOp**(“Bayern”)⟩, **applyobjmethod**(“getName”, ⟨⟩, **applyobjmethod**(“getFederalState”, ⟨⟩, **c2**(*x*)))))
streamC	**City**(“c2”) **select**[**fun**(*x*: *objectTuple*(⟨(“c2”, *oid*(“City”))⟩)) **applyobjmethod**(“==”, ⟨**applyobjmethod**(“getInhabitants”, ⟨⟩, **c2**(*x*))⟩, **applyobjmethod**(“getInhabitants”, ⟨⟩, **c1**(*x*)))]
streamD	**City**(“c2”) **select**[**fun**(*x*: *objectTuple*(⟨(“c2”, *oid*(“City”))⟩)) **applyobjmethod**(“==”, ⟨**refStringOp**(“Bayern”)⟩, **applyobjmethod**(“getName”, ⟨⟩, **applyobjmethod**(“getFederalState”, ⟨⟩, **c2**(*x*))))]

Algebraausdruck im aktuellen deskriptiven Algebraausdruck die Operanden des Operators **product** sowie die Paare im Objekttupel innerhalb des Prädikats der auf diesen Produktoperator folgenden Selektion vertauscht. Das Ergebnis dieser Regelanwendung wird in Beispiel 5.13 gezeigt. Es wird im Anschluss unter dem Bezeichner *streamG* in den aktuellen Regelkontext eingefügt.

Bisher wurden durch die vorliegende Regel `streamToExe` die beiden Operanden des Produktoperators im ursprünglichen Algebraausdruck dahingehend überprüft, ob sie als Ausgangspunkt für eine Selektion unter Berücksichtigung eines Teilprädikats der auf diesen Produktoperator folgenden Selektion dienen können. Durch ein Vorziehen von Selektionsprädikaten und das damit verbundene frühe Aussortieren von nicht am Anfrageergebnis beteiligten Datensätzen wird nämlich im Allgemeinen die erwartete Zeit zur Abarbeitung

Beispiel 5.12 Nach dem zweiten Aufruf von `moveSelection` zurückgegebener Algebraausdruck

City(“c2”)
  select[
    fun(*x*: *objectTuple*(⟨(“c2”, *oid*(“City”))⟩))
    applyobjmethod(
      “==”,
      ⟨**refStringOp**(“Bayern”)⟩,
      applyobjmethod(
        “getName”,
        ⟨⟩,
        applyobjmethod(“getFederalState”, ⟨⟩, **c2**(*x*))))]
City(“c1”)
  select[
    fun(*x*: *objectTuple*(⟨(“c1”, *oid*(“City”))⟩))
    applyobjmethod(
      “==”,
      ⟨**refStringOp**(“NRW”)⟩,
      applyobjmethod(
        “getName”,
        ⟨⟩,
        applyobjmethod(“getFederalState”, ⟨⟩, **c1**(*x*))))]
    product
      select[
        fun(*x*: *objectTuple*(⟨(“c2”, *oid*(“City”)), (“c1”, *oid*(“City”))⟩))
        applyobjmethod(
          “==”,
          ⟨**applyobjmethod**(“getInhabitants”, ⟨⟩, **c2**(*x*))⟩,
          applyobjmethod(“getInhabitants”, ⟨⟩, **c1**(*x*)))]

eines Ausführungsplans reduziert. Allerdings können Selektionsprädikate beliebig tief geschachtelt sein, sodass es manchmal nicht ausreicht, die vorliegende Regel nur einmalig aufzurufen. Eventuell führt eine weitere Anwendung dieser Optimierungsregel nämlich zu weiteren vorgezogenen Teilen des ursprünglichen Selektionsprädikats. Aus diesem Grund wird mittels

streamG.isEqual(*streamC*)

überprüft, ob der ursprüngliche Algebraausdruck *streamC* durch die aktuelle Regelanwendung verändert wurde. Das ist im vorliegenden Beispiel der Fall, denn offensichtlich entspricht der an den Bezeichner *streamG* gebundene Algebraausdruck (siehe auch Beispiel 5.13) nicht dem unter *streamC* im Regelkontext verfügbaren Algebraausdruck (siehe auch Abbildung 5.9).

Also erfolgt durch

streamH : Expr = streamToExe(*streamG*);

ein weiterer Aufruf einer Optimierungsregel namens `streamToExe` für den in Beispiel 5.13 gezeigten Algebraausdruck. Dessen Ergebnis wird anschließend unter dem Namen *streamH* in den aktuellen Regelkontext eingefügt und durch die letzte Anweisung der vorliegenden Regel als ihr Resultat an die aufrufende Optimierungsregel zurückgegeben.

Bei dem zuvor erwähnten Aufruf einer weiteren Optimierungsregel namens `streamToExe` kommt erneut die erste der fünf Regeln dieses Namens zum Einsatz, deren Muster mit dem übergebenen Algebraausdruck übereinstimmt. Dabei handelt es sich dieses Mal nicht um die zuvor aufgerufene und als letzte Regel in Beispiel 5.7 dargestellte Optimierungsregel, weil deren Muster die Verwendung der booleschen Operation `and` im Prädikat der auf den Produktoperator folgenden Selektion im übergebenen Algebraausdruck erwartet. Dort tritt jedoch nur noch ein durch == repräsentierter Vergleich auf (siehe auch Beispiel 5.13). Daher wird das Muster der nächsten Optimierungsregel namens `streamToExe` – also der ersten in Beispiel 5.8 dargestellten Optimierungsregel – näher betrachtet. Dieses Muster

{*streamA streamB* **product select** [*funA*]} : Expr

Beispiel 5.13 Zweite Rückgabe von `exchangeOperandsInProductSelection`

City(“c1”)
  select[
    fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”))>))
    applyobjmethod(
      “==”,
      <**refStringOp**(“NRW”)>,
      applyobjmethod(
        “getName”,
        <>,
        applyobjmethod(“getFederalState”, <>, **c1**(*x*))))]
City(“c2”)
  select[
    fun(*x*: *objectTuple*(<(“c2”, *oid*(“City”))>))
    applyobjmethod(
      “==”,
      <**refStringOp**(“Bayern”)>,
      applyobjmethod(
        “getName”,
        <>,
        applyobjmethod(“getFederalState”, <>, **c2**(*x*))))]
    product
      select[
        fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))>))
        applyobjmethod(
          “==”,
          <**applyobjmethod**(“getInhabitants”, <>, **c2**(*x*))>,
          applyobjmethod(“getInhabitants”, <>, **c1**(*x*)))]

erwartet einen beliebigen Algebraausdruck, der eine auf einen Produktoperator folgende Selektion mit zugehörigem Prädikat *funA* einsetzt. Daher wird nun diese Optimierungsregel auf den übergebenen Algebraausdruck angewendet.

Die bei diesem Aufruf an die Platzhalter des Musters gebundenen Teile des als Argument dienenden Algebraausdrucks werden in Tabelle 5.7 gezeigt. Die Anwendung dieser Optimierungsregel führt vor allem zur Ersetzung der Produktoperation mit nachfolgender Selektion durch eine Verbundoperation und initiiert danach die Übersetzung des dadurch erzeugten deskriptiven Algebraausdrucks in einen Ausdruck der ausführbaren Algebra. Dazu wird in der ersten Anweisung

streamC : Expr = streamToExe({*streamA streamB* **join** [*funA*]});

dieser Optimierungsregel zuerst ein neuer Algebraausdruck erzeugt (siehe auch Beispiel 5.14), bevor für diesen Algebraausdruck erneut eine Optimierungsregel mit dem Namen `streamToExe` aufgerufen wird. Das Ergebnis dieser Regelanwendung wird unter dem Bezeichner *streamC* in den aktuellen Regelkontext eingefügt und in der folgenden Anweisung zurückgegeben.

Der zuvor erwähnte Aufruf einer weiteren Optimierungsregel namens `streamToExe` erfordert ein Muster, das mit einem den Operator **join** verwendenden Algebraausdruck übereinstimmt. Aus diesem Grund kann nur die zweite der in Beispiel 5.8 gezeigten Optimierungsregeln zum Einsatz kommen. Diese sorgt dafür, dass zuerst die Operanden des Verbundoperators in die ausführbare Algebra übersetzt werden, bevor der Operator **join** selbst durch den Operator **nestedLoopJoin** der ausführbaren Algebra ersetzt wird. Das eingehende Muster dieser Regel

{*streamA streamB* **join** [*funA*]} : Expr

bindet Teile des übergebenen Algebraausdrucks wie in Tabelle 5.7 gezeigt an die im Muster verwendeten Platzhalter. Anschließend sorgen die beiden Anweisungen

streamC : Expr = streamToExe({*streamA*});
streamD : Expr = streamToExe({*streamB*});

Tabelle 5.7 Ausgewählte Platzhalter beim zweiten und dritten Aufruf von `streamToExe`

Platzhalter	Teil des Algebraausdrucks
streamA	**City**(“c1”) **select**[**fun**(*x*: *objectTuple*(<(“c1”, *oid*(“City”))>)) **applyobjmethod**(“==”, <**refStringOp**(“NRW”)>, **applyobjmethod**(“getName”, <>, **applyobjmethod**(“getFederalState”, <>, **c1**(*x*))))]
streamB	**City**(“c2”) **select**[**fun**(*x*: *objectTuple*(<(“c2”, *oid*(“City”))>)) **applyobjmethod**(“==”, <**refStringOp**(“Bayern”)>, **applyobjmethod**(“getName”, <>, **applyobjmethod**(“getFederalState”, <>, **c2**(*x*))))]
funA	**fun**(*x*: *objectTuple*(<(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))>)) **applyobjmethod**(“==”, <**applyobjmethod**(“getInhabitants”, <>, **c2**(*x*))>, **applyobjmethod**(“getInhabitants”, <>, **c1**(*x*)))

Beispiel 5.14 Verwendung einer Verbundoperation anstelle von **product** und **select**

City(“c1”)
select[
fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”))>))
applyobjmethod(
“==”,
<**refStringOp**(“NRW”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c1**(*x*))))]
City(“c2”)
select[
fun(*x*: *objectTuple*(<(“c2”, *oid*(“City”))>))
applyobjmethod(
“==”,
<**refStringOp**(“Bayern”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c2**(*x*))))]
join[
fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))>))
applyobjmethod(
“==”,
<**applyobjmethod**(“getInhabitants”, <>, **c2**(*x*))>,
applyobjmethod(“getInhabitants”, <>, **c1**(*x*)))]

Tabelle 5.8 Ausgewählte Platzhalter beim vierten Aufruf von `streamToExe`

Platzhalter	Teil des Algebraausdrucks
streamA	**City**(“c1”)
funA	**fun**(*x*: *objectTuple*(<(“c1”, *oid*(“City”))>))
	applyobjmethod(
	“==”,
	<**refStringOp**(“NRW”)>,
	applyobjmethod(
	“getName”,
	<>,
	applyobjmethod(“getFederalState”, <>, **c1**(*x*))))

dafür, dass die Operanden *streamA* und *streamB* der Verbundoperation in die ausführbare Algebra übersetzt werden. Diese Transformation erfolgt für beide Operanden aufgrund ihrer Ähnlichkeit nahezu gleich, sodass an dieser Stelle nur die Übersetzung des an *streamA* gebundenen Teilausdrucks näher thematisiert wird.

Für diese Übersetzung erfolgt erneut ein Aufruf einer der fünf Optimierungsregeln mit dem Namen `streamToExe`. Dieses Mal wird der an *streamA* gebundene Algebraausdruck (siehe auch Tabelle 5.7) als Argument verwendet, sodass nur die vorletzte Optimierungsregel aus Beispiel 5.8 eingesetzt werden kann, weil sie als einzige einen Operator **select** mit beliebigem Eingabestrom und frei wählbarem Selektionsprädikat im eingehenden Muster

{*streamA* **select** [*funA*]} : Expr

besitzt. Die beiden auftretenden Platzhalter dienen beim Aufruf dieser Optimierungsregel als Bezeichner für die in Tabelle 5.8 gezeigten Teile des übergebenen Algebraausdrucks. Der Operator **select** ist Bestandteil beider in GOODAC zum Einsatz kommender Algebren, sodass er selbst nicht verändert werden muss. Das Gleiche gilt für das durch *funA* referenzierte Selektionsprädikat, sodass nur der eingehende Objektstrom *streamA* einer Transformation in die ausführbare Algebra bedarf, um den gesamten an die aktuelle Optimierungsregel übergebenen Algebraausdruck in die ausführbare Algebra zu übersetzen. Die erste Anweisung

streamB : Expr = streamToExe({*streamA*});

sorgt durch eine weitere Regelanwendung für diese Übersetzung, bevor das Ergebnis dieser Regelanwendung verwendet wird, um das Resultat

streamB **select** [*funA*]

der Ausführung der aktuellen Optimierungsregel zu erzeugen.

Dieser Regelaufruf für den zuvor an *streamA* gebundenen Algebraausdruck (siehe auch Tabelle 5.8) führt wiederum zur Auswahl einer Regel namens `streamToExe` mit passendem eingehenden Muster. Dieses muss einer Verwendung des Operators **className** in beliebiger Instanziierung entsprechen, sodass nur die letzte in Beispiel 5.8 gezeigte Optimierungsregel mit dem Muster

{*identifierA* (*identA*)} : Expr

in Frage kommt. Beim Aufruf dieser Optimierungsregel werden die Teile des übergebenen Algebraausdrucks wie in Tabelle 5.9 gezeigt an die im Muster der Regel verwendeten Platzhalter gebunden. Diese Regel hat zum Ziel, einen Ausdruck der ausführbaren Algebra zu erzeugen, der das Auslesen aller in der Datenbank gespeicherten Instanzen zu der durch den gegebenen Ausdruck der deskriptiven Algebra dargestellten Klasse über eine Indexstruktur repräsentiert. Die an den Bezeichner *identA* gebundene Zeichenkette muss auch in der ausführbaren Algebra als Bezeichner des zu erzeugenden Objektstroms dienen, damit die Objekte dieses Stroms auch nach einer Kombination mehrerer Ströme durch das oben erwähnte Kreuzprodukt bestimmt werden können. Dieses ist beispielsweise bei der oben gezeigten Verwendung des Operators **name** in seiner Instanziierung als **c1** oder **c2** in den auftretenden Selektionsprädikaten erforderlich.

Diese aktuell ausgeführte Regel `streamToExe` erreicht Ihr Ziel mit Hilfe dreier Anweisungen. In der ersten Anweisung

c : Coll<Expr> = PrimitiveRetrieveIndices({*identifierA*});

Tabelle 5.9 Ausgewählte Platzhalter beim fünften Aufruf von `streamToExe`

Platzhalter	Teil des Algebraausdrucks
identifierA	**City**(“c1”)
identA	“c1”
indexA	*listIndex*(*oid*(“City”))

Beispiel 5.15 Nach dem fünften Aufruf von `streamToExe` zurückgegebener Algebraausdruck

listIndex(*oid*(“City”))
“c1”
scan[**fun**(*x*: *objectTuple*(‹(“c1”, *oid*(“City”))›)) **refBoolOp**(true)]

Beispiel 5.16 Nach dem vierten Aufruf von `streamToExe` zurückgegebener Algebraausdruck

listIndex(*oid*(“City”))
“c1”
scan[**fun**(*x*: *objectTuple*(‹(“c1”, *oid*(“City”))›)) **refBoolOp**(true)]
select[
fun(*x*: *objectTuple*(‹(“c1”, *oid*(“City”))›))
applyobjmethod(
“==”,
‹**refStringOp**(“NRW”)›,
applyobjmethod(
“getName”,
‹›,
applyobjmethod(“getFederalState”, ‹›, **c1**(*x*))))]

wird das Primitiv `PrimitiveRetrieveIndices` mit dem an *identifierA* gebundenen Klassennamen aufgerufen. Dieses Primitiv muss vom Systemprogrammierer speziell für GOODAC entwickelt werden, weil noch kein generischer Zugriff auf den Systemkatalog möglich ist. Als Ergebnis seines Aufrufs erzeugt dieses Primitiv mit Hilfe des Systemkatalogs von GOODAC eine Kollektion `c` von Algebraausdrücken, die vorhandene Indexstrukturen zum Zugriff auf alle Instanzen der durch ihren Namen angegebenen Klasse darstellen. Bisher werden diese Indexstrukturen im Rahmen des Optimierungsprozesses in GOODAC nicht näher untersucht, stattdessen wird durch

[*indexA*] : Coll<Expr> = c.get(1);

eine Kollektion erzeugt, die den ersten vom Primitiv zurückgelieferten Algebraausdruck unter dem Bezeichner *indexA* enthält. Diese neue Kollektion entsteht durch Anwendung der Methode `c.get(1)`.

Sobald ein Kostenmodell für GOODAC entwickelt worden ist, kann an dieser Stelle zum Beispiel eine die Kostenfunktion repräsentierende Eigenschaft angegeben werden, um – durch die Berechung des zugehörigen Eigenschaftswerts – den Algebraausdruck aus der Kollektion zu extrahieren, der den Indexzugriff mit der geringsten erwarteten Zeit zum Auslesen aller Objekte ermöglicht. Insbesondere im Zusammenhang mit der Berücksichtigung einer Sortierung der auszulesenden Objekte oder der Extraktion nur bestimmter Objekte aus der Datenbank gewinnen ein Kostenmodell und weitere Eigenschaften von Algebraausdrücken von GOODAC an Bedeutung.

In diesem Beispiel wird ohne Berücksichtigung einer Kostenfunktion der in Tabelle 5.9 gezeigte Algebraausdruck an den Bezeichner *indexA* gebunden. Er repräsentiert den Zugriff auf alle in der Datenbank gespeicherten Objekte der Klasse `City`. Dieser Bezeichner wird im Folgenden verwendet, um den zurückzugebenden Algebraausdruck

indexA identA **scan** [**fun**(*x*: *objectTuple*(‹(*identA*, *oid*(“*identifierA*”))›)) **refBoolOp**(true)]

dieser Regel zu erzeugen. Als Ergebnis liefert diese Optimierungsregel damit den in Beispiel 5.15 gezeigten ausführbaren Algebraausdruck. Damit ist die Ausführung dieser Optimierungsregel beendet.

Auch die aufrufende Optimierungsregel – also die dritte der in Beispiel 5.8 dargestellten – zur Übersetzung des eine Selektion enthaltenden Algebraausdrucks kann mit diesem Resultat ihr eigenes Ergebnis erzeugen, das in Beispiel 5.16 gezeigt wird.

Damit ist die Regelanwendung innerhalb der ersten Anweisung

Beispiel 5.17 Nach den ersten drei Aufrufen von `streamToExe` zurückgegebener Algebraausdruck

listIndex(*oid*(“City”))
“c1”
scan[**fun**(*x*: *objectTuple*(‹(“c1”, *oid*(“City”))›)) **refBoolOp**(true)]
select[
fun(*x*: *objectTuple*(‹(“c1”, *oid*(“City”))›))
applyobjmethod(
“==”,
‹**refStringOp**(“NRW”)›,
applyobjmethod(
“getName”,
‹›,
applyobjmethod(“getFederalState”, ‹›, **c1**(*x*))))]
listIndex(*oid*(“City”))
“c2”
scan[**fun**(*x*: *objectTuple*(‹(“c2”, *oid*(“City”))›)) **refBoolOp**(true)]
select[
fun(*x*: *objectTuple*(‹(“c2”, *oid*(“City”))›))
applyobjmethod(
“==”,
‹**refStringOp**(“Bayern”)›,
applyobjmethod(
“getName”,
‹›,
applyobjmethod(“getFederalState”, ‹›, **c2**(*x*))))]
nestedLoopJoin[
fun(*x*: *objectTuple*(‹(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))›))
applyobjmethod(
“==”,
‹**applyobjmethod**(“getInhabitants”, ‹›, **c2**(*x*))›,
applyobjmethod(“getInhabitants”, ‹›, **c1**(*x*)))]

streamC : Expr = streamToExe({*streamA*});

der zweiten in Beispiel 5.8 dargestellten Optimierungsregel beendet. Das in Beispiel 5.16 gezeigte Resultat wird unter hier unter dem Bezeichner *streamC* in den aktuellen Regelkontext eingefügt, bevor die nachfolgende Anweisung

streamD : Expr = streamToExe({*streamB*});

ausgeführt wird. Diese liefert ein nahezu identisches Ergebnis, das an *streamD* gebunden wird. Somit kann durch

return {*streamC streamD* **nestedLoopJoin** [*funA*]};

das in Beispiel 5.17 präsentierte Resultat erzeugt und als Ergebnis dieser Regelanwendung zurückgeliefert werden. Damit ist die Ausführung dieser Optimierungsregel abgeschlossen. Doch auch die letzten beiden noch aktiven Regeln namens `streamToExe` haben ihre Ausführung nahezu beendet. Sie binden das durch die zuvor beendeten Regelanwendungen erhaltene Ergebnis an die Bezeichner *streamC* – im Fall der ersten in Beispiel 5.8 gezeigten Optimierungsregel – oder *streamH* – im Fall der letzten in Beispiel 5.7 gezeigten Optimierungsregel. Im Anschluss geben Sie den an den jeweiligen Bezeichner gebundenen Algebraausdruck zurück und beenden ihre Ausführung, sodass die erste aufgerufene Optimierungsregel `refToExe` den in Beispiel 5.17 dargestellten Algebraausdruck als Ergebnis des in der Anweisung

streamB : Expr = streamToExe({*streamA*});

enthaltenen Regelaufrufs erhält.

Das Ergebnis dieser Regelanwendung – also der in Beispiel 5.17 gezeigte Algebraausdruck – wird unter dem Namen *streamB* in den aktuellen Regelkontext eingefügt. Weil die Regel `refToExe` nur noch die Anweisung

Beispiel 5.18 Nach dem Aufruf von `refToExe` als Endresultat zurückgegebener Algebraausdruck

streamToCollection(
listIndex(*oid*(“City”))
“c1”
scan[**fun**(*x*: *objectTuple*(<(“c1”, *oid*(“City”))>)) **refBoolOp**(true)]
select[
fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”))>))
applyobjmethod(
“==”,
<**refStringOp**(“NRW”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c1**(*x*))))]
listIndex(*oid*(“City”))
“c2”
scan[**fun**(*x*: *objectTuple*(<(“c2”, *oid*(“City”))>)) **refBoolOp**(true)]
select[
fun(*x*: *objectTuple*(<(“c2”, *oid*(“City”))>))
applyobjmethod(
“==”,
<**refStringOp**(“Bayern”)>,
applyobjmethod(
“getName”,
<>,
applyobjmethod(“getFederalState”, <>, **c2**(*x*))))]
nestedLoopJoin[
fun(*x*: *objectTuple*(<(“c1”, *oid*(“City”)), (“c2”, *oid*(“City”))>))
applyobjmethod(
“==”,
<**applyobjmethod**(“getInhabitants”, <>, **c2**(*x*))>,
applyobjmethod(“getInhabitants”, <>, **c1**(*x*)))],
applyclassmethod(“new”, <>, “OOGDMBag”))

return {**streamToCollection**(*streamB*, **applyclassmethod**(“new”, <>, “OOGDMBag”))};

enthält und weil es sich bei dieser Optimierungsregel um den Einstiegspunkt in den Optimierungsprozess handelt, wird nun das Endresultat dieses Optimierungsprozesses, der den in Beispiel 4.9 gezeigten Algebraausdruck als Ausgangspunkt verwendet hat, erzeugt. Der als Endergebnis erzeugte Ausdruck der ausführbaren Algebra ist in Beispiel 5.18 dargestellt.

Zusammenfassung

Mit dieser zuvor erfolgten Beschreibung wurde ein ausführliches Beispiel zur Verwendung von Optimierungsregeln im Rahmen des Optimierungsprozesses in GOODAC vorgestellt. Dabei wurde gezeigt, dass dieser Optimierungsprozess derzeit im Wesentlichen in der Transformation von Ausdrücken der deskriptiven Algebra in entsprechende Ausdrücke der ausführbaren Algebra besteht. Zusätzlich kommen einige Umformungen in der deskriptiven Algebra zum Einsatz, die auf Heuristiken beruhen. So wurden im vorangegangenen Beispiel Optimierungsregeln eingesetzt, die für ein Vorziehen von Teilprädikaten einer Selektion gesorgt haben, damit bei der Abarbeitung des später zu erzeugenden Ausführungsplans im Anfrageergebnis nicht enthaltene Elemente möglichst früh aus den auftretenden Objektströmen ausgesondert werden. Dadurch wird eine geringere erwartete Ausführungszeit zur Abarbeitung eines Ausführungsplans erreicht.

Der als Ergebnis des Optimierungsprozesses ermittelte ausführbare Algebraausdruck (siehe auch Beispiel 5.18) repräsentiert den in Beispiel 3.19 dargestellten textuellen Ausführungsplan. Durch das in Abschnitt 4.4.3 beschriebene Vorgehen lässt sich der erzeugte ausführbare Algebraausdruck auf diesen textuellen Ausführungsplan abbilden. Nur die in Beispiel 3.19 dargestellte Ausgabe des Anfrageergebnisses auf einen C++-Ausgabestrom besitzt keine Entsprechung im hier erzeugten textuellen Ausführungsplan.

Das liegt darin begründet, dass vom Benutzer gestellte OOGQL-Anfragen grundsätzlich auf einen C++-Ausgabestrom umgeleitet werden. Diese Ausgabe wird jedoch erst bei der Erzeugung eines textuellen Ausführungsplans aus dem entsprechenden ausführbaren Algebraausdruck berücksichtigt.

Weiterhin wurde mit diesem Abschnitt ein ausführliches Beispiel für die Verwendung von EGO im Rahmen der Anfragebearbeitung in einem konkreten Datenbankmanagementsystem gegeben. Dabei wurden die wesentlichen Konzepte von EGO aufgegriffen und ihre konkrete Anwendung gezeigt. Im Rahmen der Visualisierung des Optimierungsprozesses, hinsichtlich der einfachen Darstellung der durch Modifikation der Optimierungsstrategie bedingten Veränderungen des Optimierungsresultats sowie bei der Entwicklung einer für GOODAC geeigneten Optimierungsstragie lassen sich weitere Vorteile von EGO erkennen.

Kapitel 6

Zusammenfassung und Ausblick – Erweiterungsmöglichkeiten für GOODAC und EGO

In diesem Kapitel sollen die wesentlichen Ergebnisse der vorliegenden Arbeit zusammengefasst werden (Abschnitt 6.1), bevor in Abschnitt 6.2 unterschiedliche Erweiterungsmöglichkeiten für die Anfragebearbeitung in GOODAC und deren Auswirkungen auf die hier vorgestellten Komponenten thematisiert werden. Der Abschnitt 6.2 beschreibt zudem einige mögliche Erweiterungsmöglichkeiten für EGO, die sich ausgehend von den bisher vorgestellten Eigenschaften für die Zukunft anbieten.

6.1 Zusammenfassung

In dieser Arbeit wurde die Anfragebearbeitung im objektorientierten Datenbank-Kernsystem GOODAC vorgestellt. Dazu wurden nach einer kurzen Einführung in dieses Datenbank-Kernsystem sowie in die Thematik der Anfragebearbeitung alle an der Anfragebearbeitung in GOODAC beteiligten Komponenten präsentiert und ihr Zusammenwirken durch ein Aktivitätsdiagramm (siehe auch Abbildung 1.2) erläutert. Zuerst erfolgte eine Darstellung der Anfragesprache OOGQL und der gegenüber ihrer früheren Einführung veränderten Definition. Im Anschluss wurde mit der OOGDM-ODL die Sprache zur Formulierung von Klassendefinitionen kurz vorgestellt.

Die Ausführungsebene stellt die konkreten Algorithmen zur Ermittlung des Anfrageergebnisses bereit. Die einzelnen Algorithmen und ihr Zusammenwirken wurden ausführlich erläutert; ebenso wurde deutlich, dass es sich bei dem in GOODAC verwendeten Ansatz um eine Fortführung bewährter Technologien aus dem Umfeld relationaler Datenbanksysteme handelt. Der Einsatz der vorhandenen Algorithmen zur Berechnung eines bestimmten Anfrageergebnisses wird durch einen Ausführungsplan beschrieben. Daher wurde der Aufbau derartiger Ausführungspläne ebenfalls vorgestellt.

Zur internen Darstellung von in OOGQL formulierten Anfragen wurde die deskriptive Algebra ausgehend vom zugrunde liegenden Konzept der Second-Order Signature vorgestellt. Sie ermöglicht eine prozedurale Formulierung von Anfragen, sodass die nachfolgende Optimierung einer Anfrage durch ihre Verwendung vereinfacht wird. Ebenso wurde gezeigt, dass sich Ausführungspläne durch Ausdrücke einer ausführbaren Algebra repräsentieren lassen. Somit besteht der Optimierungsprozess von Anfragen derzeit im Wesentlichen aus der Übersetzung eines deskriptiven Algebraausdrucks in einen Ausdruck der ausführbaren Algebra.

Anschließend wurde dieser Optimierungsprozess unter Verwendung von EGO – einer neuen Komponente zur erweiterbaren generischen Anfrageoptimierung – präsentiert. Dazu wurden zuerst ausgehend von einer Übersicht über bestehende Komponenten zur erweiterbaren Anfrageoptimierung die Funktionalität, die Verwendung sowie die Realisierung von EGO thematisiert, bevor die Anbindung von EGO an GOODAC gezeigt wurde.

Im Rahmen dieser Arbeit wurde dabei ausgehend von der in Beispiel 3.18 gezeigten OOGQL-Anfrage die Erzeugung eines zugehörigen deskriptiven Algebraausdrucks (siehe auch Beispiel 4.9) zur internen Darstellung dieser Anfrage durch den OOGQL-Parser (siehe auch Abschnitt 2.3) thematisiert. Weiterhin wurde

in Abschnitt 5.3.3 ausführlich die Übersetzung dieses deskriptiven Algebraausdrucks in einen Ausdruck der ausführbaren Algebra (siehe auch Beispiel 5.18) im Rahmen des Optimierungsprozesses erläutert. Dessen Abbildung auf einen textuellen Ausführungsplan (siehe auch Beispiel 3.19) wurde in Abschnitt 4.4.3 beschrieben, während die eigentliche Verarbeitung eines derartigen Ausführungsplans in Abschnitt 3.2 erläutert wurde. Somit erfolgte im Rahmen dieser Arbeit neben der allgemeinen Beschreibung der dargestellten Konzepte auch eine Präsentation einer konkreten Anfrage in allen Phasen der Anfragebearbeitung.

Insgesamt liefert diese Arbeit also eine Beschreibung des Verlaufs der Anfragebearbeitung in GOODAC sowie der einzelnen beteiligten Komponenten. Dabei wurden einige wesentliche Neuerungen vorgestellt. So existiert etwa mit EGO die erste Komponente zur erweiterbaren Anfrageoptimierung, die eine regelbasierte Formulierung der Optimierungsstrategie ermöglicht. Weiterhin folgt die Definition interner Repräsentationsformen für Anfragen und Ausführungspläne in GOODAC der durch das Konzept der Second-Order Signature verfügbaren Spezifikationstechnik, um auch hier eine möglichst gute Erweiterbarkeit zu gewährleisten. Zudem liegt mit dieser Arbeit eine ausführliche Beschreibung erweiterbarer Konzepte auf der Ausführungsebene vor, die auch das Zusammenspiel der eingesetzten Algorithmen detailliert erläutert. Schließlich sind alle hier beschriebenen Ansätze innerhalb von GOODAC implementiert und voll funktionsfähig.

6.2 Erweiterungsmöglichkeiten für GOODAC und EGO

Bereits bei der Vorstellung der einzelnen Komponenten wurde die Erweiterbarkeit von GOODAC und damit auch seiner einzelnen Bestandteile betont. In diesem Abschnitt werden nun einige Erweiterungsmöglichkeiten aufgegriffen, um ihre Auswirkungen auf die übrigen Komponenten zur Anfagebearbeitung in GOODAC zu beleuchten.

Erweiterung von OOGQL und OOGDM-ODL

Die meisten möglichen Erweiterungen der Anfragesprache OOGQL haben ausschließlich Auswirkungen auf den OOGQL-Parser, der in OOGQL formulierte Anfragen in Ausdrücke der deskriptiven Algebra übersetzt. Dieses hat verschiedene Gründe. So können etwa neue Prädikate und geometrische oder temporale Operationen genauso wie die bereits verfügbaren Prädikate und Operationen auf Methodenaufrufe der beteiligten Objekte abgebildet werden. Daher ist in diesem Fall keine Änderung an den übrigen Komponenten von GOODAC erforderlich. Ähnlich verhält es sich mit der Definition neuer Schlüsselwörter für OOGQL. Solange diese nur syntaktischen Zucker darstellen, können sie vom OOGQL-Parser ebenfalls mit den bestehenden Mitteln der deskriptiven Algebra ausgedrückt werden.

Anders stellt sich dieses jedoch dar, falls OOGQL um neue Schlüsselwörter erweitert wird, die zusätzliche Funktionalität repräsentieren. So kann analog zu SQL ein Schlüsselwort `outerjoin` zur Realisierung einer äußeren Verbundoperation in OOGQL eingeführt werden. Dieses Schlüsselwort besitzt derzeit keine Entsprechung in der deskriptiven Algebra, sodass diese ebenfalls um einen passenden Operator erweitert werden muss. Damit sind auch Optimierungsregeln für den Einsatz von EGO erforderlich, die diesen neuen Operator berücksichtigen. Um diese Funktionalität auf der Ausführungsebene darstellen zu können, muss zudem ein neuer Algorithmus als Knotentyp für ausführbare Anfragegraphen realisiert werden, dessen Repräsentation als neuer Operator in die ausführbare Algebra aufzunehmen ist. Das Hinzufügen neuer Schlüsselwörter zu OOGQL kann also eine Erweiterung fast aller übrigen an der Anfragebearbeitung beteiligten Komponenten erforderlich machen. Nur die Optimierungskomponente EGO selbst bedarf aufgrund des gewählten erweiterbaren generischen Ansatzes keiner Ergänzung.

Die Definitionssprache OOGDM-ODL dient der Bereitstellung textueller Klassendefinitionen für in der Datenbank zu speichernde Objekte; daher beziehen sich die meisten Erweiterungen der OOGDM-ODL auch auf die direkte Beschreibung der Klassen und verursachen keine Änderungen der an der Anfragebearbeitung beteiligten Komponenten von GOODAC. Nur die angegebenen Repräsentationsformen für Klassen – also beispielsweise die zu verwendenden Indexstrukturen zur Speicherung der zugehörigen Objekte – können Erweiterungen der Ausführungsebene erforderlichen machen, wenn dort eben diese Indexstrukturen und entsprechende Zugriffe berücksichtigt werden müssen. Daraus resultieren auch Erweiterungen der ausführbaren Algebra durch das Hinzufügen eines neuen Typkonstruktors für die weitere Indexstruktur und eventuell neuer Operatoren, die ausschließlich auf dieser Indexstruktur arbeiten können. Schließlich muss die neue Indexstuktur im Rahmen des Optimierungsprozesses durch entsprechende Optimierungsregeln Berücksichtigung finden.

Erweiterung der Ausführungsebene

Eine Erweiterung der Ausführungsebene um neue Knotentypen für ausführbare Anfragegraphen liegt häufig darin begründet, dass diese zur Abbildung neuer Funktionalität innerhalb der OOGQL oder der OOGDM-ODL erforderlich werden. Diese Erweiterungen wurden bereits oben beschrieben und werden daher hier nicht aufgeführt.

Anders sieht es aus, falls die bestehende Funktionalität um Alternativen erweitert wird. So ist es beispielsweise möglich, neue Algorithmen zur Berechnung der Verbundoperation als Knotentyp für ausführbare Anfragegraphen in die Ausführungsebene zu integrieren. Damit ein solcher neuer Knotentyp auch im Rahmen der Anfragebearbeitung Verwendung finden kann, muss zum einen ein entsprechender Operator in die ausführbare Algebra aufgenommen werden; zum anderen muss dieser Operater innerhalb der verwendeten Optimierungsregeln zum Einsatz kommen.

Erweiterung der algebraischen Repräsentation

Viele Ursachen, die eine Erweiterung der deskriptiven oder ausführbaren Algebra bedingen, wurden in diesem Abschnitt bereits dargestellt und sollen daher hier nicht mehr aufgegriffen werden. In diesem Zusammenhang wurden ebenso die Auswirkungen des Hinzufügens neuer Typen oder Operatoren zu einer der beiden Algebren dargestellt, sodass diese Erweiterungsmöglichkeiten auch nicht mehr beschrieben werden. Weitere Möglichkeiten zur Erweiterung der algebraischen Repräsentation existieren daneben nicht.

Erweiterung von EGO

EGO lässt sich ebenfalls in einige Richtungen erweitern. An dieser Stelle sollen vor allem die Konsequenzen hinsichtlich der Anfragebearbeitung aufgegriffen werden.

So können in EGO zusätzliche Anweisungsklassen integriert werden, um zum Beispiel weitere Arten von Schleifen – etwa in der Form `while` ... `do` ... `done` – zu unterstützen. Außerdem lassen sich bestehende Anweisungsklassen erweitern, sodass beispielsweise zusätzliche Funktionalität für Kollektionen ermöglicht wird. Derartige Änderungen führen gewöhnlich zu einer Anpassung der Optimierungsregeln, damit diese von der erweiterten Ausdruckskraft profitieren können.

Weiterhin ist es – ähnlich zu SECONDO [Die01, 3 f] – möglich, die bisherige für jede Algebra neu zu entwickelnde Komponente zur Konvertierung von Algebraausdrücken in das interne generische Format durch das direkte Einlesen von SOS-Spezifikationen und einen darauf aufbauenden generischen Algebra-Parser zu ersetzen. Durch diese Änderung kann die Anbindung an ein konkretes Datenbankmanagementsystem durch einfaches Bereitstellen der SOS-Spezifikation sowie der zu verwendenden Optimierungsregeln erfolgen. Insbesondere muss in einem derartigen Ansatz keine konkrete Unterklasse von der Klasse `ExpressionConverter` abgeleitet werden, um die Konvertierung von Algebraausdrücken in das interne generische Format vorzunehmen, sodass ein wesentlicher Schritt zur Einbindung von EGO in die Anfragebearbeitung eines konkreten Datenbankmanagementsystems entfällt. Für die Berechnung von Eigenschaftswerten muss jedoch erneut eine für jedes konkrete Datenbankmanagementsystem zu entwickelnde Komponente eingesetzt werden. Schon Dieker [Die01] beschreibt ein derartiges Verfahren; allerdings bildet er Algebraausdrücke intern durch geschachtelte Listen in textueller Darstellung ab, sodass sich dieses System kaum für den in GOODAC verfolgten objektorientierten Ansatz mit der geforderten Erweiterbarkeit und Flexibilität eignet. Insbesondere die fest vorgegebene Struktur der Listen schränkt mögliche Erweiterungen ein.

Als dritte Erweiterungsmöglichkeit für EGO bietet es sich an, eine Option zur Kompilation von Optimierungsregeln einzuführen. Ein wesentlicher Vorteil dieser Variante liegt darin, dass nach dem ausführlichen Test einer Optimierungsstrategie die verwendeten Regeln als Kompilat vorliegen, sodass sich die Dauer des Optimierungsprozesses verringert. Werden nun nachträglich Änderungen an der Optimierungsstrategie erforderlich oder müssen neue Optimierungsregeln hinzugefügt werden, so können alle Regeln erneut so lange interpretativ verwendet werden, bis ein zufriedenstellendes Ergebnis erzielt wurde. Anschließend lassen sich die Optimierungsregeln wiederum kompilieren.

Erweiterung der Anfrageoptimierung

Neben den bereits angeführten Erweiterungsmöglichkeiten der Anfrageoptimierung, die durch die zuvor beschriebenen Änderungen anderer an der Anfragebearbeitung beteiligter Komponenten bedingt wurden, existieren zwei weitere sinnvolle Ergänzungen für die Anfrageoptimierung in GOODAC.

Einerseits ist es möglich, häufige und komplexe Optimierungsvorgänge durch Primitive abzubilden, um beispielsweise die Anzahl der aufzurufenden Optimierungsregeln zu reduzieren. Weiterhin können Primitive teilweise eine grundlegende Funktionalität – etwa Zugriffe auf den Systemkatalog – bereitstellen, die nicht durch Optimierungsregeln ausgedrückt werden kann.

Andererseits existieren noch keine Überlegungen zur Formulierung einer ausgereiften Optimierungsstrategie für Anfragen in GOODAC. Bisher beschränkt sich der Optimierungsprozess vor allem darauf, einen Ausdruck der deskriptiven Algebra in einen Ausdruck der ausführbaren Algebra zu überführen, der schließlich auf einen Ausführungsplan abgebildet werden kann, welcher ein Vorgehen zur Beantwortung der ursprünglich gestellten Anfrage beschreibt. Selbst einfache Konzepte wie beispielsweise der Vergleich des Einsatzes unterschiedlicher Algorithmen zur Berechnung einer Verbundoperation [SKS02, 13.5] finden noch keinen Eingang in die verwendete Optimierungsstrategie. Es steht daher noch aus, eine adäquate Strategie für den Optimierungsprozess in GOODAC zu ermitteln. Die Vielzahl der in der Literatur verfügbaren Ansätze und Überlegungen – vergleiche für eine erste Übersicht etwa zum einen die Literaturhinweise bei Date *et. al.* [Dat00, 17], Garcia-Molina *et. al.* [GMUW02, 16.9] sowie Silberschatz *et. al.* [SKS02, 14] und zum anderen die Ausführungen von Yu und Meng [YM98] sowie die Aufsätze in der Sammlung von Freytag [FMV94] – verdeutlicht dabei die Komplexität dieser Aufgabe. Als erster Ansatz kann dazu die Definition einer geeigneten Kostenfunktion [SKS02, 14.2] dienen, deren Ergebnis beispielsweise als Eigenschaftswert eines Algebraausdrucks ermittelt wird. Auch hier ist allerdings zu gewährleisten, dass die ermittelte Optimierungsstrategie auch in zukünftigen Erweiterungen von GOODAC im Rahmen des Einsatzes in einem bestimmten Anwendungsbereich verwendet und um anwendungsspezifische Aspekte erweitert werden kann.

Literaturverzeichnis

[ACB97] Aberer, Karl, Dunren Che und Klemens Böhm: *Rule-Based Generation of Logical Query Plans with Controlled Complexity*. In: Bry, François, Raghu Ramakrishnan und Kotagiri Ramamohanarao (Herausgeber): *Deductive and Object-Oriented Databases, 5th International Conference, DOOD'97, Montreux, Switzerland, December 8–12, 1997, Proceedings*, Band 1341 der Reihe *Lecture Notes in Computer Science*, Seiten 399–416, Berlin, Dezember 1997. Springer.

[AF93] Aberer, Karl und Gisela Fischer: *Object-Oriented Query Processing: The Impact of Methods on Language, Architecture and Optimization*. GMD Technical Report 763, German National Research Center for Computer Science, Darmstadt, Juli 1993.

[AS85] Abelson, Harold und Gerald Jay Sussman: *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, 1985.

[BE77] Blasgen, Mike W. und Kapali P. Eswaran: *Storage and Access in Relational Data Bases*. IBM Systems Journal, 16(4):363–377, 1977.

[BG92] Becker, Ludger und Ralf Hartmut Güting: *Rule-Based Optimization and Query Processing in an Extensible Geometric Database System*. ACM Transactions on Database Systems, 17(2):247–303, November 1992.

[BKK96] Berchtold, Stefan, Daniel A. Keim und Hans-Peter Kriegel: *The X-tree : An Index Structure for High-Dimensional Data*. In: Vijayaraman, T. M., Alejandro P. Buchmann, C. Mohan und Nandlal L. Sarda (Herausgeber): *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, Seiten 28–39. Morgan Kaufmann, 1996.

[BLW88] Batory, Don S., T. Y. Leuing und T. E. Wise: *Implementation Concepts for an Extensible Data Model and Data Language*. ACM Transactions on Database Systems, 13(2):231–262, September 1988.

[BMG93] Blakeley, José, William J. McKenna und Goetz Graefe: *Experiences Building the Open OODB Query Optimizer*. In: Buneman, Peter und Sushil Jajoda (Herausgeber): *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Band 22.2 der Reihe *SIGMOD Record*, Seiten 287–296. ACM Press, Juni 1993.

[Boe02] Boege, Katrin: *Qualitative Beurteilung verschiedener Realisierungen eines Modells zur Repräsentation zeitvarianter, räumlicher Daten*. Diplomarbeit im Studiengang Mathematik, Westfälische Wilhelms-Universität Münster, Münster, Dezember 2002.

[Boe03] Boenigk, Cornelia: *PostgreSQL: Grundlagen, Praxis, Anwendungsentwicklung mit PHP*. dpunkt, Heidelberg, 2003.

[Bre00] Breimann, Christian: *Berechnung von Verbundoperationen in Datenbanksystemen*. Examensarbeit für das Lehramt Sekundarstufe II/I, Staatliches Prüfungsamt für Erste Staatsprüfungen für Lehrämter an Schulen Münster, Münster, April 2000.

[Bri01] Brill, Manfred: *Mathematik für Informatiker: Einführung an praktischen Beispielen aus der Welt der Computer*. Hanser, München, 2001.

[BV03] Breimann, Christian und Jan Vahrenhold: *External Memory Computational Geometry Revisited*. In: Meyer, Ulrich, Peter Sanders und Jop Sibeyn (Herausgeber): *Algorithms for Memory Hierarchies*, Band 2625 der Reihe *Lecture Notes in Computer Science*, Kapitel 6, Seiten 110–148. Springer, Berlin, Januar 2003.

[Car87] Carey, Michael J.: *Extensible Database Systems - Letter from the Editor*. Database Engineering, 10(2):1, Juni 1987.

[Car04] Carrie, Ralph: *Erweiterung und Integration einer Algebra zur Anfragebearbeitung in einem erweiterbaren objektorientierten GIS-Kernsystem*. Diplomarbeit im Studiengang Mathematik, Westfälische Wilhelms-Universität Münster, Münster, Januar 2004.

[CB00] Cattell, R. G. G. und Douglas K. Barry (Herausgeber): *The Object Data Standard: ODMG 3.0*. The Morgen Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, 2000.

[CB02] Connolly, Thomas und Carolyn Begg: *Database Systems: A Practical Approach to Design, Implementation, and Management*. International Computer Science Series. Addison-Wesley, Harlow, 3. Auflage, 2002.

[CLR97] Cormen, Thomas H., Charles E. Leiserson und Ronald L. Rivest: *Introduction to Algorithms*. MIT Press, 1997.

[CZ98a] Cherniack, Mitch und Stan Zdonik: *Changing the Rules: Transformations for Rule-Based Optimizers*. In: Haas, Laura M. und Ashutosh Tiwary (Herausgeber): *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Band 27.2 der Reihe *SIGMOD Record*, Seiten 61–72. ACM Press, Juni 1998.

[CZ98b] Cherniack, Mitch und Stan Zdonik: *Inferring Function Semantics to Optimize Queries*. In: Gupta, Ashish, Oded Shmueli und Jennifer Widom (Herausgeber): *VLDB'98: Proceedings of the 24th International Conference on Very Large Data Bases*, Seiten 239–250. Morgan Kaufmann, 1998.

[DAOD95] Dogac, Asuman, Mehmet Altinel, Cetin Ozkan und Ilker Durusoy: *Implementation Aspects of an Object-Oriented DBMS*. ACM SIGMOD Record, 24(1):9–14, März 1995.

[Dat00] Date, C. J.: *An Introduction to Database Systems*. Addison-Wesley, Reading, 7. Auflage, 2000.

[DB95] Das, Dinesh und Don Batory: *Prairie: A Rule Specification Framework for Query Optimizers*. In: Yu, Philip S. und Arbee L. P. Chen (Herausgeber): *Proceedings of the Eleventh International Conference on Data Engineering*, Seiten 201–210. IEEE Computer Society, März 1995.

[DBVH97] Ditt, Hendrik, Ludger Becker, Andreas Voigtmann und Klaus H. Hinrichs: *Constraints and Triggers in an Object-Oriented Geo Database Kernel*. Technischer Bericht 3/97-I, Westfälische Wilhelms-Universität Münster, Münster, 1997.

[Die01] Dieker, Stefan: *Efficient Integration of Query Algebra Modules into an Extensible Database Framework*. Dissertation, FernUniversität Hagen, Hagen, Juni 2001.

[Dit] Ditt, Henrik: *Persönliche Kommunikation*.

[DS02] Donnelly, Charles und Richard Stallman: *Bison. The YACC-compatible Parser Generator, Version 1.35*. Free Software Foundation, Boston, Februar 2002.

[Ege94] Egenhofer, Max J.: *Spatial SQL: A Query and Presentation Language*. IEEE Transactions on Knowledge and Data Engineering, 6(1):86–95, Februar 1994.

[Eil03] Eilers, Björn: *Anforderungsanalyse und Konzeption ausgewählter Komponenten zur Anfragebearbeitung in einem Geo-Datenbanksystem*. Bachelorarbeit im Studiengang Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, Münster, April 2003.

[Elv98] Elverich, Ferdinand: *Entwurf und Implementierung eines Data Dictionaries für einen objektorientierten Geo-Datenbankkern*. Diplomarbeit im Studiengang Mathematik, Westfälische Wilhelms-Universität Münster, Münster, Juli 1998.

[EM99] Eisenberg, Andrew und Jim Melton: *SQL:1999, formerly known as SQL3*. ACM SIGMOD Record, 28(1):131–138, März 1999.

[EN00] Elmasri, Ramez und Shamkant B. Navathe: *Fundamentals of Database Systems*. Addison-Wesley, Reading, 3. Auflage, 2000.

[FG94] Finance, Béatrice und Georges Gardarin: *A rule-based query optimizer with multiple search strategies*. Data & Knowledge Engineering, 13(1):1–29, August 1994.

[FMV94] Freytag, Johann Christoph, David Maier und Gottfried Vossen (Herausgeber): *Query Processing for Advanced Database Systems*. Morgan Kaufmann, San Mateo, 1994.

[Gae96] Gaede, Volker Jürgen: *Extending Query Optimization for Spatial Database Systems*. Dissertation, Humboldt-Universität Berlin, 1996.

[GD87] Graefe, Goetz und David J. DeWitt: *The EXODUS Optimizer Generator*. In: Dayal, Umeshwar und Irving L. Traiger (Herausgeber): *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, Band 16.3 der Reihe *SIGMOD Record*, Seiten 160–172. ACM Press, Dezember 1987.

[GDF+99] Güting, Ralf Hartmut, Stefan Dieker, Claudia Freundorfer, Ludger Becker und Holger Schenk: *SECONDO/QP: Implementation of a Generic Query Processor*. In: Bench-Capon, T., G. Soda und A M. Tjoa (Herausgeber): *DEXA'99, Proceedings*, Band 1677 der Reihe *Lecture Notes in Computer Science*, Seiten 66–87, Heidelberg, Januar 1999. Springer.

[GG98] Gaede, Volker Jürgen und Oliver Günther: *Multidimensional Access Methods*. ACM Computing Surveys, 30(2):170–231, Juni 1998.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design-Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[GM93] Graefe, Goetz und William J. McKenna: *The Volcano Optimizer Generator: Extensibility and Efficient Search*. In: *Proceedings of the Ninth International Conference on Data Engineering*, Seiten 209–218, April 1993.

[GMUW02] Garcia-Molina, Hector, Jeffrey D. Ullman und Jennifer Widom: *Database Systems: The Complete Book*. Prentice Hall, New Jersey, 2002.

[Gra87] Graefe, Goetz: *Rule-Based Query Optimization in Extensible Database Systems*. PhD Thesis, University of Wisconsin – Madison, November 1987. Veröffentlicht als *Computer Science Technical Report # 724, University of Wisconsin – Madison*.

[Gra93] Graefe, Goetz: *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys, 25(2):73–170, Juni 1993.

[Gra94] Graefe, Goetz: *Volcano—An Extensible and Parallel Query Evaluation System*. IEEE Transactions on Knowledge and Data Engineering, 6(1):120–135, Februar 1994.

[Güt89] Güting, Ralf Hartmut: *Gral: An Extensible Relational Database System for Geometric Applications*. Forschungsbericht 293, Universität Dortmund, Dortmund, Januar 1989.

[Güt93] Güting, Ralf Hartmut: *Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization*. In: Buneman, Peter und Sushil Jajoda (Herausgeber): *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Band 22.2 der Reihe *SIGMOD Record*, Seiten 277–286. ACM Press, Juni 1993.

[GV92] Gardarin, Georges und Patrick Valduriez: *ESQL2: An Object-Oriented SQL with F-Logic Semantics*. In: Golshani, Forouzan (Herausgeber): *Proceedings of the Eighth International Conference on Data Engineering*, Seiten 320–327. IEEE Computer Society, 1992.

[HCL+90] Haas, Laura M., Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey und Eugene Shekita: *Starburst Mid-Flight: As the Dust Clears*. IEEE Transactions on Knowledge and Data Engineering, 2(1):143–160, März 1990.

[Heu97] Heuer, Andreas: *Objektorientierte Datenbanken: Konzepte, Modelle, Standards und Systeme*. Addison-Wesley, Bonn, 2. Auflage, 1997.

[HFLP89] Haas, Laura M., J. C. Freytag, G. M. Lohmann und H. Pirahesh: *Extensible Query Processing in Starburst*. In: Clifford, James, Bruce G. Lindsay und David Maier (Herausgeber): *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Band 18.2 der Reihe *SIGMOD Record*, Seiten 377–388. ACM Press, Juni 1989.

[HLUA02] Herrmann, Uwe, Dierk Lenz, Günter Unbescheid und Johannes Ahrends: *Oracle9i für den DBA: Effizient konfigurieren, optimieren und verwalten*. EDITION Oracle. Addison-Wesley, München, 2002.

[HSGK] Heuer, Andreas, Marc H. Scholl, Dieter Gluche und Joachim Kröger: *Das DFG-Projekt CROQUE*. Information und Veröffentlichungen im Rahmen dieses Projekts sind verfügbar unter `http://wwwdb.informatik.uni-rostock.de/Forschung/CROQUE.html`.

[Jäg03a] Jäger, Andreas: *Automatische Analyse und Vorverarbeitung von Klassendefinitionen in Objektdefinitionssprachen*. Bachelorarbeit im Studiengang Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, Münster, April 2003.

[Jäg03b] Jäger, Andreas: *Untersuchung, Migration und Erweiterung einer Komponente zur automatischen Analyse und Vorverarbeitung von Klassendefinitionen in Objektdefinitionssprachen*. Diplomarbeit im Studiengang Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, Münster, November 2003.

[JK84] Jarke, Matthias und Jürgen Koch: *Query Optimization in Database Systems*. ACM Computing Surveys, 16(2):111–152, Juni 1984.

[KD99] Kabra, Navin und David J. DeWitt: *OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization*. VLDB Journal, 8(1):55–78, Mai 1999.

[KE01] Kemper, Alfons und André Eickler: *Datenbanksysteme: Eine Einführung*. Oldenbourg, München, Wien, 4. Auflage, 2001.

[Kla92] Klaustal, Theo: *Das OSCAR-Projekt*. Informatik-Bericht 92/2, Technische Universität Clausthal, Clausthal-Zellerfeld, August 1992.

[KM90] Kemper, Alfons und Guido Moerkotte: *Advanced Query Processing in Object Bases Using Access Support Relations*. In: McLeod, Dennis, Ron Sacks-Davis und Hans-Jörg Schek (Herausgeber): *16th International Conference on Very Large Data Bases*, Seiten 290–301. Morgan Kaufmann, 1990.

[KM94] Kemper, Alfons und Guido Moerkotte: *Query Optimization in Object Bases: Exploiting Relational Techniques*. In: Freytag, Johann Christoph et al. [FMV94], Kapitel 3.

[KMP93] Kemper, Alfons, Guido Moerkotte und Klaus Peithner: *A Blackboard Architecture for Query Optimization in Object Bases*. In: Agrawal, Rakesh, Seán Baker und David A. Bell (Herausgeber): *19th International Conference on Very Large Data Bases, Proceedings*, Seiten 543–554. Morgan Kaufmann, 1993.

[Knu97] Knuth, Donald E.: *Sorting and Searching*, Band 3 der Reihe *The Art of Computer Programming*. Addison-Wesley, Reading, 2. Auflage, 1997.

[KPH98] Kröger, Joachim, Stefan Paul und Andreas Heuer: *Query Optimization: Ordering Rules?* Preprint aus dem Fachbereich Informatik CS-12-98, Universität Rostock, Rostock, Mai 1998.

[Kva02] Kvalbein, Arild: *An SQL parser and query processor for general objects*. Post-Graduate Thesis, Høgskolen i Stavanger, Stavanger, Juni 2002.

[Lan96] LANGE, HOLGER: *Entwurf und Implementierung der Programmierschnittstelle eines objektorientierten GIS-Kerns*. Diplomarbeit im Studiengang Mathematik, Westfälische Wilhelms-Universität Münster, Münster, August 1996.

[Lan98] LANDIN, PETER J.: *A Generalization of Jumps and Labels*. Higher-Order and Symbolic Computation, 11(2):124–143, Dezember 1998. Nachdruck eines gleichnamigen Technischen Berichts des Autors, Univac Systems Programming Research, 29.08.1965.

[LBK02] LEWIS, PHILIP M., ARTHUR BERNSTEIN und MICHAEL KIFER: *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, Boston, 2002.

[LFL88] LEE, MAVIS K., JOHANN CHRISTOPH FREYTAG und GUY M. LOHMANN: *Implementing an Interpreter for Functional Rules in a Query Optimizer*. In: BANCILHON, FRANÇOIS und DAVID J. DEWITT (Herausgeber): *Fourteenth International Conference on Very Large Data Bases, Proceedings*, Seiten 218–229. Morgan Kaufmann, 1988.

[ME92] MISHRA, PRITI und MARGARET H. EICH: *Join Processing in Relational Databases*. ACM Computing Surveys, 24(1):63–113, März 1992.

[Mel02] MELNIKOV, VLADISLAV: *Reengineering der Repräsentationsebene eines Datenbankprojektes für Geo-Anwendungen*. Abschlussarbeit im Studiengang Angewandte Informatik, Westfälische Wilhelms-Universität Münster, Münster, September 2002.

[Mor02] MORTENSEN, BJARKE BUUR: *Profiling the Predator Query Execution Engine*. Technical Report 2002/02, University of Copenhagen, 2002.

[Oes01] OESTEREICH, BERND: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language*. Oldenbourg, München, 5. Auflage, 2001.

[OW96] OTTMANN, THOMAS und PETER WIDMAYER: *Algorithmen und Datenstrukturen*. Spektrum, Heidelberg, 3. Auflage, 1996.

[Pau97] PAUL, STEFAN: *Analyse des Optimierungspotentials der algebraischen Äquivalenzregeln in CROQUE*. Diplomarbeit im Studiengang Informatik, Universität Rostock, Rostock, Dezember 1997.

[Pax95] PAXSON, VERN: *Flex. A fast scanner generator, Version 2.5*. University of California, Berkeley, März 1995.

[Pet03] PETKOVIĆ, DUŠAN: *SQL objektorientiert*. Addison-Wesley, München, 2003.

[PHH92] PIRAHESH, HAMID, JOSEPH M. HELLERSTEIN und WAQAR HASAN: *Extensible/Rule Based Query Rewrite Optimization in Starburst*. In: STONEBREAKER, MICHAEL (Herausgeber): *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, Band 21.2 der Reihe *SIGMOD Record*, Seiten 39–48. ACM Press, Juni 1992.

[Pos03] *PostgreSQL: Das offizielle Handbuch*. Bonn, 2003.

[Pro03] PROGRESS SOFTWARE COOPERATION: *ObjectStore Release 6.1: C++ API User Guide*, Februar 2003.

[Puk03] PUKE, IRIS: *Algorithmen für zeitvariante, räumliche Daten mit diskreten Repräsentationen*. Diplomarbeit im Studiengang Mathematik, Westfälische Wilhelms-Universität Münster, Münster, Oktober 2003.

[RC87] RICHARDSON, JOEL E. und MICHAEL J. CAREY: *Programming Constructs for Database System Implementation in EXODUS*. In: DAYAL, UMESHWAR und IRVING L. TRAIGER (Herausgeber): *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, Band 16.3 der Reihe *SIGMOD Record*, Seiten 208–219. ACM Press, Dezember 1987.

[Ric01] RICCARDI, GREG: *Principles of Database Systems with Internet and Java Applications*. Addison-Wesley, Boston, 2001.

[Rie94] Riedel, Holger: *Effiziente Anfrageauswertung in objektorientierten Datenbanken: Optimierungskonzepte für eine kalkülbasierte Sprache*. Dissertation, Technische Universität Clausthal, Clausthal-Zellerfeld, 1994. Veröffentlicht im Shaker-Verlag, Aachen.

[Rin02] Ringena, Take: *Reengineering der Anfrageausführung eines Datenbank-Projekts für Geo-Anwendungen*. Abschlussarbeit im Studiengang Angewandte Informatik, Westfälische Wilhelms-Universität Münster, Münster, September 2002.

[SAC+79] Selinger, Patricia G., Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie und Thomas G. Price: *Access Path Selection in a Relational Database Management System*. In: Bernstein, Philip A. (Herausgeber): *ACM SIGMOD 1979: International Conference on Management of Data*, Seiten 23–34. ACM Press, 1979.

[Sch93] Schiwietz, Michael: *Speicherung und Anfragebearbeitung komplexer Geo-Objekte*. Dissertation aus dem Fach Informatik, Ludwig-Maximilians-Universität München, München, März 1993.

[Sch97] Schöning, Uwe: *Theoretische Informatik – kurzgefasst*. Spektrum-Hochschultaschenbuch. Spektrum, Heidelberg, 3. Auflage, 1997.

[Sch04] Schmidt, Holger: *Generische Optimierung von Anfragen an einen Datenbankkern für Geo-Anwendungen*. Diplomarbeit im Studiengang Mathematik, Westfälische Wilhelms-Universität Münster, Münster, Mai 2004.

[SGG00] Silberschatz, Abraham, Peter Galvin und Greg Gagne: *Applied Operating System Concepts*. John Wiley & Sons, New York, 2000.

[SKS02] Silberschatz, Abraham, Henry F. Korth und S. Sudershan: *Database System Concepts*. McGraw-Hill, New York, 4. Auflage, 2002.

[SS90] Sciore, Edward und John Sieg Jr.: *A Modular Query Optimizer Generator*. In: *Proceedings of the Sixth International Conference on Data Engineering*, Seiten 146–153. IEEE Computer Society, 1990.

[Ste96] Steinberg, Tilmann: *Entwurf und Implementierung einer Basis-Klassenhierarchie als Implementierungsebene eines objektorientierten GIS-Datenmodells*. Diplomarbeit im Studiengang Mathematik, Westfälische Wilhelms-Universität Münster, Juni 1996.

[Str98] Stroustrup, Bjarne: *Die C++-Programmiersprache*. Addison-Wesley, Bonn, 3. Auflage, 1998.

[Tür03] Türker, Can: *SQL:1999 & SQL:2003: Objektrelationales SQL, SQLJ & SQL/XML*. dpunkt, Heidelberg, 2003.

[vdBH93] Bussche, Jan van den und Andreas Heuer: *Using SQL with Object-Oriented Databases*. Information Systems, 18(7):461–487, Oktober 1993.

[Vit01] Vitter, Jeffrey Scott: *External Memory Algorithms and Data Structures: Dealing with MASSIVE Data*. ACM Computing Surveys, 33(2):209–271, Juni 2001.

[Voi97] Voigtmann, Andreas: *An Object-Oriented Database Kernel for Spatio-Temporal Geo-Applications*. Inaugural-Dissertation, Westfälische Wilhelms-Universität Münster, Münster, 1997.

[Vos99] Vossen, Gottfried: *Datenbankmodelle, Datenbanksprachen und Datenbankmanagement-Systeme*. Oldenbourg, München, Wien, 3. Auflage, 1999.

[Wie01] Wiedmann, Thomas: *DB2: SQL, Programmierung und Tuning*. Computer & Literatur, Böblingen, 2001.

[WJW98] Wu, Yu, Sushil Jajodia und X. Sean Wang: *Temporal Database Bibliography Update*. In: Etzion, Opher, Sushil Jajodia und Suryanarayana M. Sripada (Herausgeber): *Temporal Databases: Research and Practice*, Band 1399 der Reihe *Lecture Notes in Computer Science*, Seiten 338–366, Heidelberg, Januar 1998. Springer.

[WM99] Warshaw, Lane B. und Daniel P. Miranker: *Rule-Based Query Optimization, Revisited*. In: *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management*, Seiten 267–275. ACM Press, November 1999.

[YM98] Yu, Clement T. und Weiyi Meng: *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, San Francisco, 1998.

[Zha94] Zhan, Jibin: *Object-Oriented Query Language Specification and Query Processor Generation*. Master's Thesis, Simon Fraser University, Canada, Juni 1994.

[Zie02] Ziemann, Olaf: *Reengineering eines Datenbankprojektes aus Sicht der Anwendungs-Programmierung*. Abschlussarbeit im Studiengang Angewandte Informatik, Westfälische Wilhelms-Universität Münster, Münster, September 2002.

Lebenslauf

Persönliche Daten

Name	Christian Breimann
Geburt	23.10.1978 in Datteln
Nationalität	Deutsch
Familienstand	ledig, Hochzeit am 30.07.2004
Eltern	Hubert Breimann und Brigitte Breimann, geb. Tiews

Schule & Studium

08/1985 – 07/1989	Kardinal-von-Galen-Schule, Waltrop
08/1989 – 06/1997	Theodor-Heuss-Gymnasium, Waltrop
09.06.1997	Abitur
10/1997 – 11/2000	WWU Münster: Studium Lehramt der Sekundarstufen II/I, für die Fächer Informatik und Mathematik
10.11.2000	Erstes Staatsexamen für das Lehramt der Sekundarstufen II/I für die Fächer Informatik und Mathematik
12/2000 – 05/2002	WWU Münster: Studium Diplom-Informatik mit Anwendungsfach Mathematik
27.05.2002	Diplom in Informatik
seit 06/2002	WWU Münster: Studien zur Promotion in Informatik und Beginn der Dissertation bei Prof. Dr. Klaus H. Hinrichs

Tätigkeiten

02/1998 – 04/1998 08/1998 – 09/1998 05/2000	Miebach, Dortmund: Migration, Teilkonzeption und Erweiterung eines firmeneigenen Informationssystems auf Basis einer Oracle-Datenbank als Werkstudent
07/1999 – 03/2000	IVV 7, WWU Münster: studentische Hilfskraft Rechnerwartung
10/1999 – 02/2000 04/2000 – 08/2000	Lehrstuhl Vossen, WWU Münster: Tutor zu den Vorlesungen „Informatik I“ und „Informatik II“ als studentische Hilfskraft
12/2000 – 08/2004	Institut für Informatik, WWU Münster: Wissenschaftlicher Mitarbeiter