Fachbereich 12 Informatik und Mathematik
Institut für Informatik

Bachelorarbeit

Dynamische Übersetzung von PQL-Abfragen in Java-Bytecode

Hilmar Ackermann
Studiengang: Informatik
Matrikelnummer: 4422257

Frankfurt am Main
7. Oktober 2013

Betreuer: Jun. Prof. Dr. Christoph Reichenbach

Gesetzt am 7. Oktober 2013 um 7:45 Uhr mit LaTeX.

Erklärung gemäß Bachelor-Ordnung Informatik 2007 §24 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 7. Oktober 2013

Hilmar Ackermann

Danksagung

Diese Bachelorarbeit entstand am Institut für Informatik an der Goethe Universität Frankfurt am Main.

An dieser Stelle möchte ich mich bei meinem Betreuer Jun. Prof. Dr. Christoph Reichenbach für die gute und umfangreiche Betreuung bedanken.

Abstract. PQL ist eine in Entwicklung stehende Sprache, eingebettet in Java, die deklarative, logische Terme in ausführbaren Code übersetzt, dabei sowohl sequentiell, als auch parallelisierbar einsetzbar ist und einen Schwerpunkt in punkto Optimierung und Effizienz setzt. Einen Interpreter und einen Compiler gibt es bereits für PQL, einige Optimierungen sind aber erst zur Laufzeit effektiv umsetzbar, beispielsweise kann man hier bereitstehende Informationen über den genauen Typ eines Objects zur Optimierung nutzen. Hier wäre Bedarf für einen Laufzeitübersetzer, der mithilfe diesen Informationen besser optimierten Code erzeugen kann. Dabei geschieht die Übersetzung von PQL in Java-Bytecode zur Laufzeit der Java Virtual Machine, direkt bevor der Code ausgeführt werden soll. Im Rahmen dieser Bachelorarbeit wurde solch ein dyamischer Übersetzer entwickelt, der einen Großteil des PQL-Sprachumfangs unterstützt, auch konnten Verbesserungen der Performanz durch Laufzeitoptimierungen dabei gemessen werden.

Inhaltsverzeichnis

1 Motivation

1.1 Motivation der bereits existierenden Sprache PQL

Man kann sagen, dass seit Entwicklung der ersten Prozessoren deren Taktraten fast kontinuierlich, Jahr für Jahr, angestiegen sind. Doch seit 2004 hat dieser Trend ein Ende (siehe Abbildung 1): Die Taktraten steigen nicht mehr, stattdessen die Anzahl der Prozessorkerne. Die parallele Ausführung von Programmen ist für höhere Perfomanz unabdingbar und bleibt für die Informatiker der näheren Zukunft sicher einer der größten Herausforderungen.

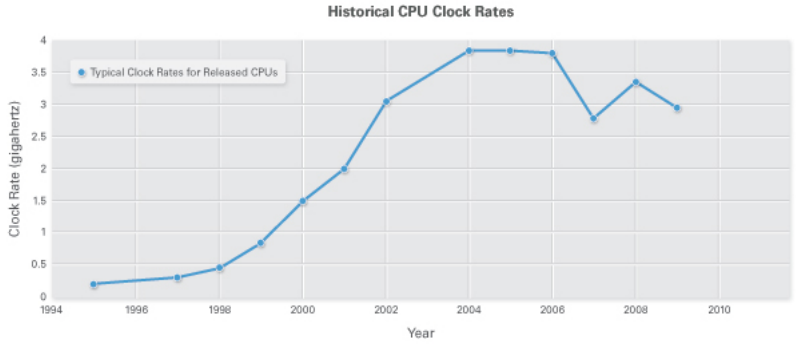


Abbildung 1: Die historische Entwicklung der Taktraten von handelsüblichen Prozessoren[15]

Man kann also sagen, dass das Stichwort *Parallelisierung* im Laufe der knapp letzten 10 Jahre in der Informatik-Branche einen immer wichtigeren Stellenwert eingenommen hat. Parallele Hardware existiert allerdings schon seit Beginn der massenhaften Verbreitung des Heim-Computers, sowohl im Professional-Bereich[1], als auch etwas später in Personalcomputern für Privatnutzer.[2] Lange Zeit nutzte man dies aber nur für streng abgetrennte Teilbereiche, bei-

[1] Als Beispiel wäre hier die *Cray 1* zu nennen, ein Supercomputer, der bereits parallele Vektoroperationen durchführen konnte. Die erste Cray 1 wurde 1977 gekauft.[4]

[2] Vergleiche dazu beispielsweise die 1982 eingeführte Hercules Graphics Card, die so konstruiert war, dass man zwei Grafikkarten parallel schalten konnte und diese sich den zu berechnenden Bildraum teilten.[3]

spielsweise für Grafikberechnungen, die man häufig sowieso Bildpunkt für Bildpunkt oder Vertex für Vertex berechnen musste und somit eine Parallelisierung keinen nennenswerten Mehraufwand bedeutete. Zwar gibt es auch in anderen Bereichen schon lange bekannte Algortithmen, die sich auch gut parallel realisieren ließen, aber im Allgemeinen wird die typische Programmierung vom sequentiellen Denken beherrscht. Da aber der wichtigste Faktor bei der Leistungssteigerung in neueren Prozessoren ein höherer Grad von Parallelisierung ist (mehr Kerne, aber ungefähr gleichbleibende Taktrate pro Kern), müssen auch die klassischen Programme immer mehr Wert darauf legen, dass die aufwendigsten Berechnungen parallel ausgeführt werden können.

Genau hier setzt PQL an. PQL ist eine nicht Turing-vollständige, deklarative, logische Sprache (basierend auf Termen der Prädikatenlogik erster Stufe), die als Erweiterung auf Java aufgesetzt ist. Somit kann die PQL-Schnittstelle verwendet werden, um aufwändige Probleme effizient parallel berechnen zu lassen, während man in Java weiterhin Teile des Programms sequentiell beschreiben kann. Das bringt uns eine ganze Reihe von immanenten Vorteilen: Die rein deklarative Natur der Sprache gibt uns großen Spielraum für starke Optimierungen, vergleichbar im Ausmaß beispielsweise mit denen von relationalen Abfragen in Datenbanken durch ihr Datenbankmanagementsystem. Gleichzeitig kann man spezielle Anforderungen in reinem Java belassen, so dass nicht die Flexibilität der umgebenden Sprache verloren geht.

Aus der Sicht eines PQL-Entwicklers, der an maximaler Optimierung interessiert ist, bildet das Spachdesign eine gute Basis: Die deklarativen Terme bilden nur eine Art *Gerüst*, man könnte sagen, der Benutzer klärt bezüglich der erwünschten Programmausführung das „Was“, überlässt das (hoffentlich maximal effizient umgesetzte) „Wie“ aber dem PQL-Übersetzer. Dies steht zum Kontrast der meisten anderen nicht-logischen, nicht-deklarativen, üblichen Sprachen, wie beispielsweise Java oder C. Hier geht der Programmierer sehr viel näher auf die gewünschte Ausführung des Codes ein, zum Beispiel indem er die Reihenfolge unabhängiger *for-schleifen* oder *if-Abfragen* bestimmt.

Aus der Sicht eines Programmierers, der PQL benutzt, ist eine gute Performanz natürlich positiv, allerdings wird er etwas in seinen Möglichkeiten eingeschränkt: Nicht jedes Programm, dass man in Java schreiben kann, kann

auch in reinem PQL geschrieben werden.[3] Dieser Nachteil ist aber nur bedingt tragisch, da man nicht umsetzbaren Code weiterhin in herkömmlichen Java schreiben kann.

[3]Ein Beispiel hierfür wäre der transitive Abschluss. Siehe dazu auch den letzten Absatz von Kapitel 3.

1.2 Abgrenzung zu anderen parallelen Erweiterungen

Wenn wir von anderen, zu PQL ähnlichen Erweiterungen sprechen, kommt PLINQ [11] dem wohl am nächsten. PLINQ ist auf .Net [16] aufgesetzt und bietet auch eine große Unterstützung für parallele Entwicklungen. Im Gegensatz zu PQL basiert PLINQ nicht auf logischen Termen (*forall/exists*) sondern auf relationalen Anfragen (*select-from-where*). Doch einer der entscheidensten Unterschiede besteht wohl darin, dass PLINQ (im Gegensatz zu PQL) nicht vollständig deklarativ ist. Dies beflügelt die Mannigfaltigkeit der beschreibbaren Ausdrucksmöglichkeiten auf der einen Seite, begrenzt aber auch die Optimierungsmöglichkeiten für den Übersetzer, das gegebene Problem in optimaler Laufzeit auf der vorhandenen Hardware zu lösen.

Schauen wir uns dazu ein Beispiel an[4]. Sei die Aufgabenstellung: Wir haben ein Set von Premiumkunden (*PREMIUMCUSTOMERS*) und eine Map, die zu Kunden eine Bestellung zuordnet (*CUST2ORDERS*). Nun suchen wir alle Premiumkunden, die mehr als 1000 Einheiten bestellt haben und wollen diese — zusammen mit Ihrer jeweiligen Bestellung als Wert — in eine Map abspeichern. In PQL könnte man das folgendermaßen aufschreiben:

```
1 Map m = query (Map.get(cust) == order):
2     premiumcustomers.contains(cust) &&
3     cust2orders.get(cust) == order && order.amount > 1000;
```

In PLINQ würde man folgendes schreiben, um die gleiche Funktionalität zu erhalten:

```
1 var m = from cust in premiumcustomers.AsParallel()
2         from order in cust2orders[cust].AsParallel()
3         where order.amount > 1000
4         select new f cust, order g;
```

Der Unterschied ist nun der, dass wir dem Übersetzer bereits gesagt haben, in welcher Reihenfolge die Abfragen vorzunehmen sind: Das PLINQ-Programm wird zuerst über alle *PREMIUMCUSTOMERS* drüberiterieren und dann durch alle Elemente in *CUST2ORDERS* laufen. Schauen wir uns dazu das Beispiel noch einmal an, diesmal aber übersetzt in iterativen Pseudocode:

[4] übernommen aus der offiziellen Doku von PQL, Kapitel 1 [9]

```
1 Map M = new Map
2 foreach CUSTOMER in PREMIUMCUSTOMERS
3         ORDERSOFCUSTOMER = CUST2ORDERS.get(CUSTOMER)
4         foreach ORDER in ORDERSOFCUSTOMER
5                 if ORDER.AMOUNT > 1000
6                         M.insert(CUSTOMER, ORDER)
```

Der Unterschied zwischen dem PQL-Code und dem PLINQ-Code ist nun der, dass der entsprechende PQL-Übersetzer die beiden *foreach-Schleifen* problemlos tauschen könnte, im Gegenzug dazu in PLINQ die Reihenfolge bereits vom Programmierer festgelegt wurde. Welche Variante aber effizienter ist, hängt nun ausschließlich vom Inhalt der beiden Datenstrukturen *PREMIUMCUSTOMERS* und *CUST2ORDERS* ab. Dazu gibt es auch später nochmal eine ausführliche Erläuterung, siehe dazu Kapitel 1.3.

Zusätzlich zu PLINQ gibt es auch noch eine ganze Reihe von anderen Erweiterungen, die es dem Programmierer ermöglichen, Teile seines Codes parallel auszuführen, doch — genau wie PLINQ — sind sie in der Regel nicht vollständig deklarativ. Als Beispiele wären hier zu nennen *Grand Central Dispatch (GCD)* [13] als Erweiterung für Cocoa (Mac OS X) oder auch die in Java8 eingeführten Lambda-Ausdrücke, welche sich auch zum Parallelisieren von Schleifen, etc.. eignen[2].

1.3 Motivation des dynamischen Übersetzers

Die Entwicklung eines dynamischen Übersetzers, der den Code direkt in Java-Bytecode verwandelt, bringt eine neue Stufe der Optimierungs-Möglichkeiten mit sich: Die effizienteste Ausführungsart gegebener abstrakter Problemstellungen können häufig nur zur Laufzeit bestimmt werden. Denn für die Effizienz eines Programms ist nicht nur der eigentliche Code von Interesse, auch der von außen hereinkommende Input kann die Performanz von Programmen maßgeblich beeinflussen. Können wir also zur Laufzeit Einfluss auf die Art der Ausführung einer gegebenen Aufgabenstellung nehmen, so können wir die Perfomanz entscheidend verbessern.

Betrachten wir dies nun noch einmal ausführlich: Wie schon im vorherigen Kapitel besprochen, ist einer der großen Stärken von PQL die vollständige Deklarativität. Diesen Vorteil kann man aber erst voll ausnutzen, wenn man Informationen für die Programmausführung nutzt, die erst zur Laufzeit bekannt sind. Nehmen wir folgendes Beispiel:

```
1 Set s = query (Set.contains(x)):
2     set1.contains(x) && set2.contains(x);
```

Wir suchen alle Einträge, die sowohl in *set1*, als auch in *set2* vorhanden sind. Wenn ein entsprechender Übersetzer die Größe der Sets nicht kennt, würde das Programm nun zuerst alle Einträge aus *set1* nehmen und dann für jeden dieser Einträge in *set2* suchen, ob auch hier der jeweilige Wert vorhanden ist. Zur Laufzeit wissen wir im Gegensatz dazu aber die Größe der vorliegenden Sets. Hat *set2* nun sehr viel weniger Einträge als *set1*, so ist es wesentlich perfomanter zuerst über alle Einträge in *set2* drüber zu iterieren und diese dann mit *set1* zu vergleichen. Wir werden das ganze nun nochmal minimal mathematisch betrachten. Wir definieren dazu folgendes: (die nun angegebenen Berechnungsaufwands-Größenordnungen sind realistisch und bei vielen Set-Implementierungen in dieser Art messbar, beispielsweise dem auf einem balancierten Binärbaum aufbauenden, java-internen TreeSet[5]):

[5]Siehe dazu auch 'Java ist auch eine Insel', Kapitel 13.5.2 [14]

Iteriere über Set S: $\mathcal{O}(|S|)$

Lookup in S: $\mathcal{O}(log(|S|))$

$|set1| > |set2|$

Dann gilt:

Iteriere über set1, lookup in set2 = $\mathcal{O}(|set1|) * \mathcal{O}(log(|set2|)) >$

$\mathcal{O}(log(|set1|)) * \mathcal{O}(|set2|)$ = Iteriere über set2, lookup in set1

Daneben gibt es auch noch viele andere Fälle, bei denen wir optimale Laufzeiten nur erreichen können, wenn wir die genaue Programmausführung erst zur Laufzeit bestimmen, beispielsweise:

* Wir können das Wissen über die genauen Typen ausnutzen. Haben wir beispielsweise ein Befehl, der durch ein Set durchiterieren muss, so wird das unterschiedlich gemacht, abhängig davon, ob wir ein Java-internes Set oder ein PQL-eigenes-Set benutzen. Bei kompilierten Code müssten wir nun zuerst abfragen, um welche Art von Set es sich handelt. Wissen wir dies schon zur Laufzeit, können wir diesen Check einsparen.

* Auch das Wissen über die vorhandenen Prozessorkerne können wir ausnutzen, um die Last der Berechnung bestmöglich auf die eingebaute Hardware aufzuteilen.

* Teilweise können wir auch häufiger *Konstanten-Inlining* (dies bezeichnet das Schreiben von konstanten Werten direkt in den Bytecode) betreiben, da es Variablen gibt, die in einem PQL-Programm nur gelesen werde, also quasi konstant sind, deren Wert aber erst zur Laufzeit bekannt ist. Dies kann passieren, wenn wir den Wert beispielsweise im Voraus durch Java-Code oder ein anderes PQL-Programm berechnen.

Solcher Art Optimierungen sind übrigens nicht nur bei PQL denkbar, auch in klassischen — rein sequentiellen — Programmiersprachen wären Vorteile bei einer just-in-time-Übersetzung denkbar. Programme, die zur Laufzeit andere Programme beobachten und anschließend versuchen, diese aufgrund der

gewonnen Informationen zu optimieren kennt man in der Fachsprache auch unter dem Stichwort *meta-programming*:

„A program analysis (a meta-program) observes the structure and environment of an object-program and computes some value as a result. Results can be data- or control-flow graphs, or even another object-program. Examples of these kind of meta-systems are: program transformers, optimizers, and partial evaluation systems.“[6]

Natürlich hat eine solche Art der Code-Übersetzung nicht nur einen Perfomanz-Vorteil, sondern braucht auch zur Laufzeit ein bisschen Zeit, um den ausführbaren Code zu erzeugen.[7] Da wir in PQL aber häufig Operationen nutzen, die vergleichsweise zeitaufwändig über große Arrays, Maps oder Sets operieren, ist dieser zusätzliche Erzeugungs-Zeitaufwand vernachlässigbar, wie Messungen dazu belegen. (siehe auch Kapitel 6)

[6] aus 'Accomplishments and Research Challenges in Meta-programming', Kapitel 2 [5]

[7] Dies ist der gleiche Kompromiss, den auch die Java just-in-time-Übersetzung machen muss, denn auch hier wird der vorhandene Bytecode erst zur Laufzeit in den endgültigen Maschinencode übersetzt.

2 Hintergrund: PQL

Um PQL zu programmieren, gibt es hierfür eine eigene Syntax, welche jene von klassischem Java erweitert. Die Benutzung von PQL in Java wurde bereits ausführlich in der offiziellen Dokumentation besprochen [9], ich werde deswegen hier nur kurz die Grundlagen anschneiden.

Vorangestellt (können aber bei größeren Ausdrucken auch innerhalb anderer Ausdrücke geschachtelt sein) befinden sich meistens die folgenden Ausdrücke: *forall, exists, query(..), reduce(..)*. Ein paar Beispiele mit Erklärung:

```
1 forall int x: x == x
```

Wird *true* sein, da für alle x gilt, dass sie gleich sich selbst sind. (Hierbei ist zu beachten, dass dieser Code durch den gesamten Wertebereich von int durchiterieren muss, ein Aufwand von 2^{32} Berechnungen)

```
1 exists int x: x == 7
```

Wird auch *true* sein, da es ein x gibt, dass 7 ist.

```
1 query(Set.contains(x)): x == 42
```

Erstellt ein Set, das genau eine 42 enthält.

```
1 reduce(sumDouble) x: mySet.contains(x)
```

sumDouble ist eine vordefinierte Methode, die die Summe über zwei doubles berechnet. Dies funktioniert wie eine klassische Faltung: Die Methode erhält den neuen Wert x und das bisherige Ergebnis der Berechnung (zu Beginn ist das 0). Da wir nun also summieren, werden alle Werte nacheinander auf das Ergebnis aufsummiert, sodass wir am Ende die Summe des kompletten Set-Inhalts *mySet* haben.

Die Anweisung, welche bei den vorangegangenen Beispielen immer jeweils direkt hinter dem Doppelpunkt kam, kann viele verschiedene Ausdrücke annehmen und selbstverständlich konjunktiv und disjunktiv miteinander ver-

schachtelt werden. Ein Beispiel für eine Konjunktion wäre:

```
1 query(Set.contains(x)): x == y && mySet.contains(y)
```

Dies würde beispielsweise eine Kopie des Sets *mySet* erzeugen. Ein Beispiel für eine Disjunktion wäre:

```
1 query(Set.contains(x)): x == 5 || x == 7
```

Dies würde ein Set mit den Einträgen 5 und 7 erzeugen.

Zusätzlich zu den nun schon verwendeten Ausdrücken, können diese konjunktiv oder disjunktiv verbundenen Ausdrücke noch einige andere Aufgaben erfüllen, die wichtigsten sind arithmetische Operationen, Vergleichsoperationen und Containeroperationen (*Set.contains, Map.get, Array[]*).

Schauen wir uns nun noch kurz die Grammatik zu PQL etwas genauer an (Abbildung 2). Bei den eben bereits vorgestellten Ausdrücken vor dem Doppelpunkt handelt es sich dabei um die sogenannten QUANT-EXPR, die anderen, konjunktiv oder disjunktiv verbundenen Ausdrücke nennen sich die QEXPR. Ganz oben in der Abbildung finden wir den „Einstiegspunkt“ der Grammatik, die QUERY. Zusätzlich zu den QEXPR und QUANT-EXPR gibt es also noch *id* und JAVA-EXPR. Bei der *id* handelt es sich um eine Java-Variable oder eine logische Variable (eine Variable, die durch *forall*, *exists*, etc.. quantifiziert wird) und bei der JAVA-EXPR um einen Ausdruck in Java. Die JAVA-EXPR wird dabei innerhalb der PQL-Sequenz wie eine Konstante behandelt, haben wir beispielsweise folgenden Code:

```
1 forall x: x == myJavaMethod()
```

Dann wird einmal zu Beginn die Java-Methode *myJavaMethod* aufgerufen und anschließend für alle weiteren Abfragen durch einen konstanten Wert ersetzt — nämlich genau jenen, den die Methode *myJavaMethod* zurückgegeben hat.

QUERY	::=	⟨*QUANT-EXPR*⟩ | id | ⟨*JAVA-EXPR*⟩ | ⟨*QEXPR*⟩
QUANT-EXPR	::=	⟨*QUANT*⟩ ⟨*ID*⟩ ':' ⟨*QUERY*⟩
		| **query** '(' ⟨*MATCH*⟩ ')' ':' ⟨*QUERY*⟩
		| **reduce** '(' id ')' ⟨*ID*⟩ [**over** ⟨*ID-SEQ*⟩] : ⟨*QUERY*⟩
QUANT	::=	**forall** | **exists**
QEXPR	::=	'(' ⟨*QUERY*⟩ ')' | ⟨*QUERY*⟩ ⟨*BINOP*⟩ ⟨*QUERY*⟩
		| ⟨*QUERY*⟩ **instanceof** ⟨*JAVA-TY*⟩ | ⟨*UNOP*⟩ ⟨*QUERY*⟩
		| ⟨*QUERY*⟩ '?' ⟨*QUERY*⟩ ':' ⟨*QUERY*⟩
		| ⟨*QUERY*⟩ '.' 'get' '(' ⟨*QUERY*⟩ ')'
		| ⟨*QUERY*⟩ '[' ⟨*QUERY*⟩ ']'
		| ⟨*QUERY*⟩ '.' 'contains' '(' ⟨*QUERY*⟩ ')'
		| ⟨*QUERY*⟩ '.' id | ⟨*QUERY*⟩ '.' **length**
		| ⟨*QUERY*⟩ '.' **size** '(' ')'
BINOP	::=	'||' | '&&' | '|' | '&' | '^' | '%' | '*' | '+'
		| '-' | '/' | '>' | '<' | '<=' | '>=' | '!='
		| '==' | '<<' | '>>' | '>>>' | '=>'
UNOP	::=	'!' | '~' | '-'
MATCH	::=	'Set' '.' 'contains' '(' ⟨*ID*⟩ ')'
		| 'Map' '.' 'get' '(' ⟨*ID*⟩ ')' '==' ⟨*ID*⟩ [**default** ⟨*QUERY*⟩]
		| 'Array' '[' ⟨*ID*⟩ ']' '==' ⟨*CM*⟩
ID	::=	⟨*ID*⟩ | ⟨*JAVA-TY*⟩ id
ID-SEQ	::=	⟨*ID*⟩ | ⟨*ID-SEQ*⟩ ',' ⟨*ID*⟩

Abbildung 2: PQL/Java syntax[9]

3 Hintergrund: Zwischensprache

Nach außen hin präsentiert sich PQL als Sprache mit eigener Syntax, eingebettet in Java (wie im vorherigen Kapitel besprochen). Intern wird es aber anders repräsentiert — es handelt sich um die PQL-Zwischensprache. Dabei wird jeweils ein zusammenhängender PQL-Ausdruck in ein PQIL-Programm übersetzt, wovon der Nutzer aber nichts mitbekommt.

Ein solches PQIL-Programm besteht aus genau einem Join. Dieses kann einfach nur ein atomarer Befehl sein, aber auch beispielsweise eine Kontrollstruktur, die wiederum weitere Joins enthält. Ein Join kann ganz allgemein in primitives Join und Kontrollstruktur klassifiziert werden. (siehe Abbildung 3) Ein Join hat grundsätzlich einen Namen, gefolgt von seinen Parametern und

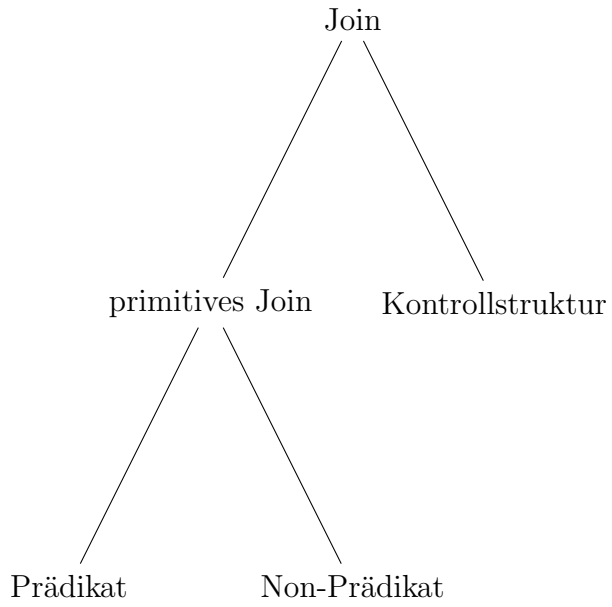


Abbildung 3: Die Klassifizierung eines Join

seinen Branches (im Falle von Kontrollstrukturen). Die Parameter erwarten dabei folgende Typen:

* int
* long
* double
* Object

Es gibt einige Joins, die auch die Funktionialität von *boolean, short, char* oder *float* unterstützen, diese Datentypen werden intern aber als *int*, bzw. *double* abgebildet.

Die Anzahl der Parameter eines Joins hängt von seinem jeweiligen Command ab, allerdings geben wir beim Erstellen eines Joins für jeden Parameter an, ob er gelesen, geschrieben oder ignoriert (*Wildcard*) werden soll.

COMMAND(?x, !y, _)

Hier wird beispielsweise das erste Argument gelesen, das zweite geschrieben und das dritte ist ein *Wildcard*. Die Kombinationsmöglichkeiten davon sind allerdings pro Befehl begrenzt, denn es ist definiert, welche Argumente welche Zugriffsart unterstützen. Bei einem einfach Additionsbefehl ist beispielsweise nur folgendes erlaubt:

ADD_Int(?x, ?y, ?z)

ADD_Int(?x, ?y, !z)

Im ersten Fall überprüfen wir, ob $x + y == z$ gilt, im zweiten Fall setzen wir $z = x + y$. Diese Variationsmöglichkeiten — auch genannt *Zugriffsmodi* — sind eine Sprachentscheidung zugunsten höherer Perfomanz und erweitern nicht die Ausdrucksmöglichkeiten von PQIL. Schauen wir uns dazu folgende Befehle an:

ADD_Int(?x, ?y, !tmp)

EQ_Int(?tmp, ?z)

Wenn wir beide Befehle hintereinander ausgeführt hätten, so wäre dies equivalent zu den ersten der beiden vorher gesehenen Ausdrücke (*ADD_Int(?x, ?y, ?z)*), ohne dass dieser Zugriffsmodus nötig gewesen wäre. Diese Hintereinanderausführung von Befehlen nennen wir übrigens *konjunktiver Block*. Ein solcher *konjunktiver Block* wird beispielsweise für die Übersetzung des PQL-Programms gebraucht, welches wir im letzten Kapitel beim Stichwort *Konjunktion* vorgestellt hatten (*query(Set.contains(x)): x == y && mySet.contains(y)*). Im Gegenzug dazu gibt es auch noch einen *disjunktiven Block*, auch für diesen findet man im vorherigen Kapitel eine Beispiel-Verwendung.

Nun bräuchten wir noch einen Ausdruck, mit dem es uns möglich wäre, im

Erfolgsfall eines Befehls oder eines ganzen Blocks das berechnete Ergebnis weiter zu verarbeiten. Man könnte es beispielsweise in eine Datenstruktur speichern wollen, um am Ende eine Sammlung aller Ergebnisse zu erhalten. Dafür gibt es die Reductors, welche zusätzlich zu den Joins existieren. So ein Reductor ist dabei um einen Block von Joins gewickelt und beschreibt ganz grundsätzlich ein Verhalten im Erfolgsfall des ganzen Blocks. Haben wir beispielsweise eine Abfrage, die für jedes mögliche *int* x abfragt, ob $x < 0$ und wir wollten die Ergebnisse in ein Set abspeichern, dann könnte man ein Reductor nutzen, der alle negativen Zahlen innerhalb des int-Definitions-Bereich $(-1, -2, .., -2147483648)$ in dieses Set speichern würde. Darüber hinaus gibt es auch noch andere Reductors, es gibt beispielsweise einen Reductor, mit dem man eine benutzerdefinierte Java-Methode im Erfolgsfall aufrufen kann.

Eine umfassende Dokumentation zum gesamten Sprachumfang befindet sich in der **PQIL revision 0.5.2**.[1]

Damit kommen wir am Ende noch einmal zu dem, was PQL nicht kann. Tatsächlich kann man in PQL nicht alles ausdrücken, was man in Java ausdrücken kann. Ein Beispiel hierfür wäre das Bestimmen der transitiven Hülle einer gegebenen Relation. Sei ein *Graph G* gegeben, mit *Knotenmenge V*, bei dem jeder Knoten genau ein Nachfolger hat, wobei alle Nachfolger in einem *Array A* der Größe $|V|$ gespeichert seien. Wollen wir nun wissen, ob man *Knoten X* von *Knoten Y* aus erreichen kann, so ginge das in reinem PQL nur, wenn wir wüssten dass die entsprechende Nachfolger-Strecke maximal n Schritte lang ist, denn um einen solchen nach oben limitiert langen Weg zu suchen, bräuchten wir dementsprechend auch $\mathcal{O}(n)$-viele PQIL-Befehle.

4 Hintergrund: Java-Bytecode und ASM

4.1 Java-Bytecode

Die Grundlagen von PQL haben wir nun besprochen, aber nun ist ja das Thema der Bachelorarbeit, dass vorhandener PQL-Code zur Laufzeit in Java-Bytecode übersetzt wird. Dazu erst einmal etwas Hintergrundwissen zu Java und seinem Bytecode:

Java hat einen entscheidenden Unterschied sowohl zu klassisch interpretierten als auch zu klassisch kompilierten Sprachen. In Java wird der Rohquelltext genommen und mithilfe eines Compilers[8] in eine Zwischensprache, den sogenannten Java-Bytecode übersetzt. Im Gegensatz zu herkömmlichen kompilierten Code ist dieser aber nun nicht direkt auf einem Prozessor ausführbar, bleibt aber dafür plattformunabhängig, da er erst im zweiten Schritt zur Laufzeit in den für den Prozessor jeweilig benötigten Maschinencode übersetzt wird.

In manchen Bereichen ähnelt der Java-Bytecode klassischem Maschinencode: Die Befehle sind meistens vergleichsweise modular, so finden wir simple arithmetische Operationen (zum Beispiel *Add*), die auf einem internen Stack operieren und dabei die obersten Werte vom Stapel nehmen, die gewünschte Berechnung ausführen und dann das Ergebnis zurückschreiben. Auch Stackoperationen (*push, pop, swap*, etc..) oder klassische *jump-to-label* Befehle (unbedingter Sprung: *goto*, bedingte Sprünge: *ifeq, ifne*, etc..) repräsentieren typische atomare Befehle, die auch häufig in Assembler-, bzw. Lowlevel-Sprachen anzutreffen sind.[9]

Andere Eigenheiten des Bytecodes unterscheiden sich aber auch stark von klassischem Maschinencode: Da jede java-Datei in eine eigene class-Datei kompiliert wird, müssen die Benennungen externer/globaler Symbole (Methoden, Attribute, etc..) beibehalten werden. Aus einer class-Datei, die den Bytecode enthält, sind weiterhin Methoden-, Klassen- und Attributnamen in Reintext herauslesbar. Im Gegensatz zu Maschinencode kennt der Bytecode damit also auch Objects, insbesondere als Spezialfall eines Objects auch Arrays, die auch mit eigenen Bytecode-Befehlen (beispielsweise *newarray* oder

[8] In der Kommandozeile ruft man dafür 'javac' auf.

[9] Für eine komplette Liste aller Bytecode-Befehle siehe hier: `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings`

arraylength) ausgestattet sind. Auch kennt der Java-Bytecode keine klassischen Register: Temporäre Werte und die Argumente einer Methode werden als lokale Variablen abgespeichert, die Anzahl der verfügbaren lokalen Variablen ist zumindestens in der Theorie unbegrenzt. Der Stack (*operand stack*) und die lokalen Variablen bilden pro Methode einen sogenannten *Frame*. Sowohl die maximale Größe für den *operand stack*, als auch für die lokalen Variablen sind vor Ausführen der Methode bekannt und werden im Java-Bytecode mitgespeichert.

Um nun den Code auf dem jeweiligen Endgerät auszuführen, braucht man dazu die *Java Virtual Machine (JVM)*. Von dieser wird der Bytecode zur Laufzeit in den vom jeweilig eingebauten Prozessor lesbaren Maschinencode übersetzt und ausgeführt. Dies geschieht nicht direkt für den gesamten Code sondern partiell und es kann dementsprechend zur Laufzeit geschehen, dass weitere Teile des Codes angefordert werden und dann zu diesem Zeitpunkt erst noch übersetzt werden müssen. Die vollständige Spezifikation der JVM kann man online abrufen.[8]

Im nachfolgenden nochmal kurz eine Gegenüberstellung der wichtigsten Vor- und Nachteile zwischen statischer und dynamischer Übersetzung:

Vorteile dynamischer Übersetzung (z.B. Java-Bytecode)	**Vorteile statischer Übersetzung (kompilierter Code)**
Programme können als closed Source (beispielsweise für proprietäre Software) weitergegeben werden ohne gleichzeitig die volle Plattformunabhängigkeit zu verlieren.[10]	Wir brauchen kein Laufzeitsystem zum Ausführen des Codes. Ein solches System läuft während der Programmausführung im Hintergrund und hat in der Regel einen hohen zusätzlichen Speicherbedarf.
Da die Programme erst zur Laufzeit erzeugt werden, kann man hier weitere Optimierungen durchführen, beispielsweise können wir das genaue Wissen über den Prozessor-Befehlssatz ausnutzen.	Wir brauchen während der Programmausführung keine zusätzliche Übersetzungszeit, da der Code schon im endgültigen Maschinencode vorliegt.

Wenn wir dynamisch übersetzen, könnten wir dies im Prinzip auch wahlweise statisch tun. Wir müssten das Programm nur einmal noch zusätzlich in Maschinencode übersetzen und können es dann behandeln, als hätten wir es direkt statisch übersetzt. Insbesondere im PQL-Laufzeitübersetzer — hier handelt es sich schließlich auch um einen dynamischen Übersetzer — ist dies problemlos möglich, da wir den generierten Bytecode einfach nur abspeichern müssten und ihn anschließend immer wieder ausführen könnten.	

Eine andere Sprache, die dem Java-Bytecode-System sehr ähnelt, ist beispielsweise die *Common Intermediate Language (CIL)* [17], die unter anderem in .Net verwendet wird. Historisch gesehen sollte man noch *p-Code* erwähnen, welcher als Vorläufer von Java bezeichnet werden kann und auch bereits erst zur Laufzeit in den endgültigen Maschinencode übersetzt wurde.[11]

[10] Selbstverständlich geht das auch mit klassischen kompilierten Programme, aber möchte man den Code in Reintext nicht mitliefern, so bleibt einem meist keine andere Möglichkeit, als für alle erwarteten Plattformen ein eigenständig kompiliertes Programm mitzuliefern, was eine ganze Reihe von Nachteilen mit sich bringt.

[11] Den p-Code gab es seit 1970 und wurde in den späten 70ern populär. Dabei konnte er auf zahlreichen Maschinen emuliert werden, zum Beispiel auf den Prozessoren 6502 (MOS Technology), dem 8080 (Intel), dem Z-80 (Zilog) oder dem PDP-11 (Digital Equipment Corporation).[6]

4.2 ASM

Klassischerweise wird nun also Java-Bytecode durch den Java-Compiler erzeugt. Dies hilft uns aber in unserem Fall nicht weiter, da wir zur Laufzeit weiteren Bytecode erzeugen wollen. Da die Umwandlung in Maschinencode erst zur Laufzeit erfolgt, ist das auch tatsächlich möglich. Bytecode kann man — wie der Name schon sagt — in einem Bytearray abbilden und dann zur Laufzeit mit Java-Standard-Methoden laden und ausführen. Da wir den Bytecode nicht direkt erstellen wollen, benötigen wir dazu eine Art Assembler, der den Bytecode erzeugt. Hier greifen wir auf eine schon existente Lösung zurück — ASM ('A Java bytecode engineering library'). Die komplette Funktionalität ist sehr ausführlich in der offiziellen Dokumentation beschrieben[10], allerdings werde ich die grundlegende Funktionsweise hier trotzdem kurz beschreiben.

Grundsätzlich gibt es zwei Varianten in ASM, Bytecode zu erzeugen:

* Die Core API (klassische Variante)
* Die Tree API (alternative Variante)

Bei der Core API rufen wir die einzelnen Befehle einen nach dem anderen in einer Art Assemblersprache auf. Die Art des Aufrufes läuft über eine Visitor-Klasse. Wollen wir beispielsweise ein Integer in den lokalen Variablen von Position 5 auf den Stack laden, so schreiben wir:

```
1 visitVarInsn(ILOAD, 5)
```

ILOAD steht dabei für (I)nteger-(LOAD). Eine Ausnahme der reinen Abarbeitung Befehl für Befehl bilden hier nur Methoden und Sprungmarken (Labels). Methoden geben bei der Erzeugung ein Methodenobject zurück, auf welches wir zugreifen müssen, wenn wir Bytecode-Befehle in der Methode haben wollen. Sprungmarken werden als Object erzeugt und werden sowohl beim Springen, als auch beim Festlegen ihrer Platzierung benötigt.

Bei der Tree-API ist jeder Befehl, jede Methode, etc.. ein eigenes Object. Dies hat den Vorteil, dass auch nach einmaliger Erstellung des Code-Gerüstes leicht Änderungen vorgenommen werden können, ist aber sowohl für den Programmierer, als auch für die Bytecode-erzeugenden Routinen ineffizienter. Die offizielle Dokumentation fasst es folgendermaßen zusammen:

> „Using the tree API to generate a class takes about 30% more time (see Appendix A.1) and consumes more memory than using the core API. But it makes it possible to generate the class elements in any order, which can be convenient in some cases.“[12]

Ich habe mich — in erster Linie aus Perfomanz-Gründen — für die Core API entschieden.

[12] ASM 4.0 A Java bytecode engineering library, Kapitel 6.1.2 (Seite 93)[10]

5 Implementierung

5.1 Design-Entscheidungen

Zuerst stellte sich die Frage, welches Framework für die Bytecode-Funktionalität genutzt werden sollte. Dabei standen im Großen und Ganzen folgende zur Auswahl:

Name	**Beschreibung**
ASM[13]	ein gutes und performantes Framework zum Erzeugen und Manipulieren von Java-Bytecode
BCEL[14]	ist vergleichsweise langsam und nicht „State of the Art“[15]
SERP[16]	ähnlich wie BCEL von seinen Nachteilen[15]
Javassist[17]	Funktioniert im Lowlevel-Bereich ähnlich wie ASM, wäre evtl. auch eine Alternative gewesen

Wie bereits früher erwähnt, wurde sich dabei für ASM entschieden. Wie in der Aufzählung ersichtlich, wäre auch Javassist in Frage gekommen, allerdings überzeugte mich die gute und ausführliche Dokumentation von ASM.[10]

ASM bietet nun zwei Arten der Bytecode-Erzeugung an: Die *Core API* und die *Tree API*. Wie bereits im vorherigen Kapitel erläutert, wurde sich dabei für die erste Art entschieden.

Da das Projekt eine ganze Menge assembler-ähnlichen Code benötigte, war es von Anfang an klar, dass wir große, unübersichtliche Code-Abschnitte haben werden. Dies habe ich zuerst einmal so hingenommen, für eine zukünftige Erweiterung des Projekts wären hier noch Ausbaumöglichkeiten denkbar, beispielsweise das Abkapseln größerer Codeabschnitte in eigene Klassen.

[13] siehe auch: `http://asm.ow2.org/`
[14] siehe auch: `http://commons.apache.org/proper/commons-bcel/`
[15] Dieser Meinung sind beispielsweise die Coder von cglib, einer Library, die auch ASM benutzt. (siehe auch [12])
[16] siehe auch: `http://serp.sourceforge.net/`
[17] siehe auch: `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/`

5.1.1 Separatisierung des dynamischen Übersetzers

Eine wichtige Design-Entscheidung war es, den Code des neuen dynamischen Übersetzers maximal getrennt zum schon vorhandenen Code zu behandeln. Dies wurde durch drei Maßnahmen erreicht: Zuerst einmal ist der problemlos abtrennbare Teil des dynamischen Übersetzers komplett und ausschließlich im eigenen Paket edu.umass.bc enthalten.

Ich habe mich bei der internen Kommunikation häufig für eine Informationskodierung mittels Flags entschieden.[18] Zum einen, da es eine einfache aber effektive und ausreichende Art der Informationsübermittlung darstellte, zum anderen, da es sich gut in das bereits vorhandene System eingliederte, in dem auch einige Informationen per Flags gespeichert wurden.[19] Außerdem stellen Flags in punkto Speicherbedarf eine kompakte und effiziente Art dar, Informationen zu kodieren.

Zur Identifizierung vorhandener Joins brauchen wir nun also eine Methode, die uns das gegebene Join beschreiben. Man hätte dazu einfach in jedes Join eine Methode einbauen können, die uns die gewünschten Informationen direkt zurückgibt. Dies hätte aber dem Grundsatz der maximalen Trennung von vorhandenem und neuem Code wiedersprochen, deshalb habe ich mich für die Erstellung von Visitor-Klassen[20] entschieden. Zwar gab es im PQL-Code schon vorher Visitorklassen für Joins, welche ich als generische Klasse hätte verwenden können und diese hätte dann abhängig vom verwendeten Join die Flags zurückgegeben. Dies hätte aber wiederum dem Grundsatz der höchsten Performanz wiedersprochen, da das Auflösen dieses Konstrukts mehr Zeit gebraucht hätte. Zur Veranschaulichung der nun eingerichteten Visitorklasse, schauen wir uns nun als Beispiel die Klasse Conjunctive an. Es handelt sich hierbei um die konjunktive Kontrollstruktur (ihre Erbreihenfolge ist folgende: Conjunctive ⇒ PreConjunctive ⇒ AbstractBlock ⇒ ControlStructure ⇒ Join). Um die Identität zu erfahren, fügen wir einfach folgende Zeile ein:

```
1 @Override public int accept (JoinFlagsVisitor visitor) {
```

[18] Alle verwendeten Flags befinden sich im übrigen in der Klasse BcFlags im package edu.umass.bc

[19] siehe zum Beispiel die Klasse edu.umass.pql.VarConstants, welche die Flags bereitstellt für die Kodierung der benötigten Env-Variablen

[20] siehe dazu auch: 'Design Patterns. Elements of Reusable Object-Oriented Software.' (Seite 9) [18]

```
2     return visitor.visit(this);
3 }
```

Das Override kommt daher, dass im Join eine Ursprungsmethode existiert, die für alle Subklassen ohne *JoinFlagsVisitor-accept-Methode* ein Standardverhalten definiert. Ansonsten kann diese Zeile einfach kopiert werden, ohne in der Klasse selbst Anpassungen vornehmen zu müssen. Als kleine Bemerkung wäre noch zu sagen, dass dieses *accept* nicht in Conjunctive, sondern in ihrer direkten Überklasse PreConjunctive steht, da es mehrere Conjunctive-Klassen gibt, aber PreConjunctive uns bereits alle benötigten Informationen gibt (nämlich, dass es sich um einen konjunktiven Block handelt) und dort dann dementsprechend nur eine *accept-Methode* benötigt wird.

Nun muss das ganze natürlich noch im Visitor berücksichtigt werden. Er muss uns nun die gewünschten Flags zurückgeben, in unserem Fall beispielsweise:

```
1 public int visit (AbstractBlock.PreConjunctive e) {
2     return TYPE_CONJUCTIVE;
3 }
```

In dem Fall haben wir jetzt nur eine Definition des Join-Typen und keine klassischen Flags. In den meisten anderen Fällen wird aber noch der Variablentyp mit angegeben, beispielsweise bei ADD_Int:

```
1 public int visit (Arithmetic.ADD_Int e) {
2     return TYPE_ADD | TYPE_INT;
3 }
```

Als dritter Punkt wäre der eigentliche Aufruf des dynamischen Übersetzers zu nennen. Es existiert zuerst einmal eine Hauptklasse, welche die Schnittstelle des dynamischen Übersetzers darstellt, sie heißt RuntimeCreator. In ihr befindet sich ein boolean-Attribut *useRuntimeCreator*. Ist es gesetzt, soll der dynamische Übersetzer benutzt werden. Die auszuführende Routine ruft dann einfach abhängig von *useRuntimeCreator* den RuntimeCreator auf oder nicht. Im Beispiel der Testumgebung (diese wurde in PQL eingerichtet und bietet eine einfache Möglichkeit, PQIL-Code direkt auf dem PQL-Interpreter auszuführen) sieht das dann folgendermaßen aus:

```
1 if (RuntimeCreator.useRuntimeCreator)   //Das oben
     vorgestellte Attribut 'useRuntimeCreator'
2 {
3     if (!RuntimeCreator.alreadyInitialized) //Wir müssen
         nur einmal initialisieren, manchmal wird dies schon
```

```
            außerhalb gemacht um dann hier nur den Bytecode
           mehrfach auszuführen
 4          RuntimeCreator.init( it, env ); //Haben wir den
              Laufzeitübersetzer einmal initialisiert, können
              wir den gleichen Code immer wieder (auch mit
              anderen Parametern) ausführen
 5      assertFalse(                        //assertFalse ist
           ein Feature der Testumgebung und beschreibt den
           erwarteten Ausgang unserer Ausführung. Gibt der
           Code doch als Rückgabewert 'true' zurück, wird eine
            Exception geworfen
 6          RuntimeCreator.test(env)        //Hier geschieht
              die eigentliche Ausführung des erzeugten Java-
              Bytecodes
 7      );
 8  }
 9  else
10  {
11      it.reset(this.env);                 //der Interpreter
           muss vor jeder Ausführung erst zurückgesetzt werden
12      assertFalse(                        //assertFalse ist
           ein Feature der Testumgebung und beschreibt den
           erwarteten Ausgang unserer Ausführung. Gibt der
           Code doch als Rückgabewert 'true' zurück, wird eine
            Exception geworfen
13          it.next(this.env)               //Hier geschieht
              die eigentliche Ausführung des Interpreters
14      );
15  }
```

Zum genauen Aufruf und der Benutzung des RuntimeCreators siehe auch Kapitel 5.3.

5.1.2 Entscheidungen zur Perfomanz

Das Stichwort *Performanz* war auch ein primäres Ziel bei der Entwicklung des Laufzeitübersetzers. Dabei gibt es zum einen die Geschwindigkeit, in der das erzugte Programm später ausgeführt wird, siehe dazu auch das Kapitel 5.5. Zum anderen muss man auch darauf achten, dass die Code-Erzeugung selbst nicht zu lange braucht, denn in einem späteren Einsatz des dynamischen Übersetzers, würde er in einigen Programmen immer wieder zur Laufzeit aufgerufen würden und somit auch auf die Gesamtlaufzeit des Pro-

gramms Einfluss nehmen. Deshalb wurde entschieden, das komplette Interface — sowohl nach außen, als auch intern — größtenteils mit statischen Methoden und Attributen zu implementieren. Diese sind im Vergleich zu nicht-statischen Methoden am effektivsten:

„As static invocations obtain their effective method addresses directly from the instruction stream, they do not require any additional memory access in the dispatch. This clearly makes them the fastest invocation kind.“[21]

5.1.3 Regeln der Namensgebung

Bei der Namensgebung habe ich mich größtenteils an die Java-üblichen Regeln gehalten: Klassennamen beginnen mit einem Großbuchstaben, Attribute und lokale Variablen (größtenteils) in der *CamelCase-Schreibweise* und Konstanten, bzw. Flagbezeichnungen komplett in Großbuchstaben, getrennt durch Unterstriche (_).

[21] Increasing the Performance and Predictability of the Code Execution on an Embedded Java Platform, Kapitel 4.3, Seite 120[7]

5.2 Strukturierung

Grundsätzlich macht es die große Menge an reinem Assemblercode schwierig, einen stetig gut strukturierten Code zu erhalten, allerdings habe ich mich bemüht, die Unübersichtlichkeit zu minimieren. Dabei werden häufig verwendete Subroutinen in eigene Methoden ausgelagert (siehe auch Kapitel 5.2.2).

5.2.1 Übersetzung der Joins

Eine große Herausforderung stellte die korrekte Abarbeitung der einzelnen Befehle dar. Die grundsätzliche Struktur behandelt dabei jeden Befehl ähnlich einer Funktion mit boolschem Rückgabewert:

* true Das Join wurde erfolgreich ausgeführt (Vergleichsoperation trifft zu, arithmetische Operation ausgeführt, angegebenes Schlüssel-Wert-Paar in einer Map gefunden, etc..).

* false Das Join wurde nicht erfolgreich ausgeführt (Vergleichsoperation trifft nicht zu, angegebenes Schlüssel-Wert-Paar nicht in einer Map gefunden, etc..).

Wenn *one* für eine Variable mit dem Inhalt 1 stehen würde, gäbe folgender Aufruf beispielsweise *false* zurück (da 1+1 nicht 1 ist):

```
1 ADD_Int(?one, ?one, ?one)
```

ADD_Int ist dabei beispielsweise ein lineares Join. Diese Joins werden bei jeder Ausführung unabhängig voneinander *true* oder *false* zurückgeben, der Rückgabewert hängt nur von der Berechnung oder dem Vergleich ab, die sie gerade ausgeführt haben. Andere Joins verhalten sich so ähnlich wie eine Schleife (suchen wir beispielsweise alle Schlüssel, die einen angegebenen Wert in einer Map haben, so müssen wir durch diese Schlüssel durchiterieren). Diese nicht-linearen Joins geben bei jedem Iterationsschritt *true* zurück und wenn sie am Ende angelangt sind, geben sie *false* zurück.

Um die Abarbeitungsreihenfolge der Befehle noch besser zu erklären, schauen wir uns nun nochmal ein komplexeres Beispiel mit folgendem PQL-Code an:

```
1 reduce (myFunction) y:
2         range(0,99).contains(x) &&
3         y == x + 5 &&
4         y == 42
```

myFunction ist dabei eine benutzerdefinierte Funktion, die alle Ergebnisse der Berechnung erhält, in dem Fall erhält sie genau einmal den Wert 42, da im Bereich zwischen 0 und 99 genau einmal der Wert $37 = 42 - 5$ liegt. Bei der Umwandlung in ein PQIL-Programm wird in diesem Fall jede Zeile in einen PQIL-Befehl umgewandelt, dies würde in vereinfachter Form (auf die Parameter wurde zur besseren Lesbarkeit verzichtet) so aussehen:

```
1 Reductor           //rufe myFunction mit y auf
2 INT_RANGE_CONTAINS //zähle von 0 bis 99 und speichere in x
3 ADD_Int            //berechne y = x+5
4 EQ_Int             //überprüfe, ob y == 42 gilt
```

Das entsprechende Kontrollfluss-Diagramm sieht man in Abbildung 4. Erfolg und Nicht-Erfolg steht bei den einzelnen Befehlen hierbei für folgende Bedeutung:

* INT_RANGE_CONTAINS: Hier zählen wir von einem angegebenem Startwert bis zu einem angegebenem Endwert. So lange wir noch zählen, bedeutet es Erfolg. Ist das Ende erreicht, so gibt der Befehl Nicht-Erfolg zurück.
* ADD_Int: Im Falle des oben angegebenen Programms würde hier tatsächlich addiert werden (es gibt auch einen Zugriffsmodus, bei dem der ADD_Int Befehl addiert und dann das Ergebnis direkt mit einem anderen Wert vergleicht) und der Befehl gibt dementsprechend immer Erfolg zurück.
* EQ_Int: Ist der Inhalt der angegebenen Variable gleich dem der anderen angegebenen Variable (die 42 im oberen Beispiel müsste in dem Falle in eine temporäre Variable gespeichert werden, dies würde man aber vor der eigentlichen Ausführung machen), so gibt es Erfolg zurück, ansonsten Nicht-Erfolg.

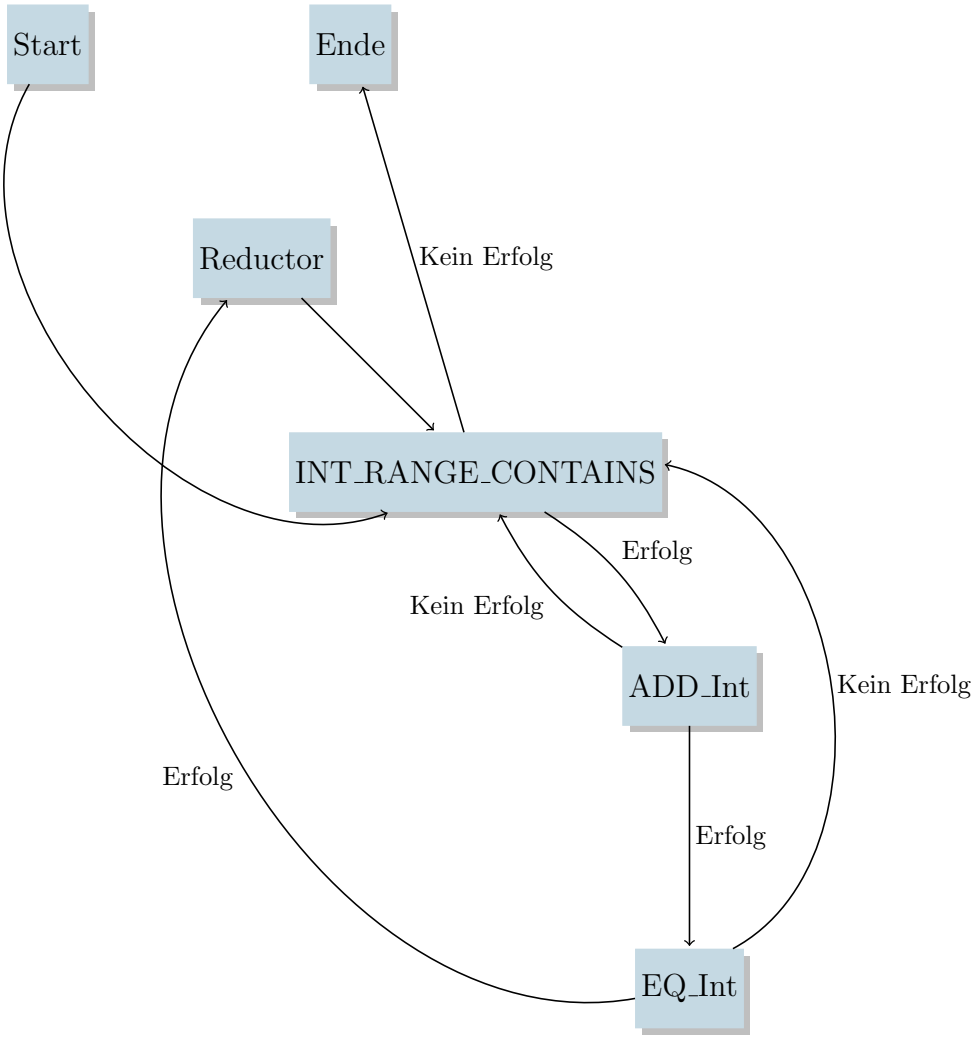


Abbildung 4: Beispielprogramm als Kontrollfluss-Diagramm

Eine Besonderheit dieses Programms bildet der Reductor, zu dem nur im Erfolgsfall des letzten Befehls innerhalb des *INT_RANGE_CONTAINS* gesprungen wird.

Der Rückgabewert wird nun aber unterschiedlich behandelt: Haben wir direkt zu Beginn nur einen einzigen, atomaren Befehl (zum Beispiel eine arithmetische Operation), wird das boolsche Ergebnis direkt zurückgegeben. Befinden wir uns in irgendeiner Art Schleife (befinden wir uns beispielsweise in einem INT_RANGE_CONTAINS), dann wollen wir im Falle eines *false* zum Schleifenkörper zurückspringen und ansonsten zum nächsten Befehl voranschreiten. Das komplizierteste ReturnBehavior stellt allerdings die Reduction dar: Hier führen wir im Erfolgsfall eine benutzerdefinierte Aktion durch: Werte werden zu Sets, Maps oder Arrays hinzugefügt, oder benutzerdefinierte Java-Methoden werden aufgerufen. Dies alles wird durch die Klasse ReturnBehavior gesteuert. Sie wird immer dann aufgerufen, wenn es in einem PQIL-Befehl bei der Bytecode-Erzeugung zu einem Erfolgs-, bzw. Nicht-Erfolgs-Fall kommt. Abhängig von der Position des Joins im restlichen Code entscheidet die ReturnBehavior-Klasse, was hier geschieht und fügt den entsprechenden Bytecode ein. Dabei kann es sich um einen Sprung zu einem vorherigen PQIL-Befehl, dem Einfügen von Reductor-Code oder dem kompletten Ende des PQIL-Programms (welches durch ein *Return* aus der erzeugten Bytecode-Methode realisiert wird) handeln. Diese Funktionalität wurde komplett in der Methode *insertReturnCode* innerhalb der ReturnBehavior-Klasse mithilfe einer *switch-case-Anweisung* implementiert. Die beiden anderen wichtigsten Methoden dieser Klasse sind *getJumpBehavior* und *getReductionBehavior* und werden jeweils in einem Schleifen-PQIL-Befehl, bzw. einem Reductor aufgerufen, um das neue Returnverhalten zu definieren.

5.2.2 Häufig verwendete Subroutinen

Einige häufig benötigte Routinen wurden in Subroutinen ausgelagert, sprich in eigene Methoden innerhalb der Klasse RuntimeCreator. Dazu zählen *pushValue* (lege Variablenwert auf den Stack), *storeValue* (nehme Variablenwert vom Stack und speichere ihn), *castValue* (konvertiere Typ vom obersten Element auf dem Stack), *checkCast* (konvertiere Object-Typ vom obersten Object auf dem Stack), *instanceOf* (finde Object-Typ vom obersten Object auf dem Stack heraus) und *loadConstant* (lade konstanten Wert auf den Stack). Es folgt eine ausführliche Beschreibung dieser Befehle:

pushValue Diese Methode lädt die gewünschte Variable aus dem Environment (Env) und legt sie auf den Stack. Der gewünschte Typ wird per Argument übergeben — entspricht der Typ nicht dem Typen der Environment-Variable, so wird er konvertiert.

storeValue Diese Methode schreibt die gewünschte Variable ins Environment (Env). Der zu schreibende Wert befindet sich dabei im per Argument übergebenem Typen auf dem Stack — passt der Typ nicht mit dem Variablentyp zusammen, wird er vorher noch konvertiert.

castValue Der Methode wird übergeben, welcher Typ gerade zuoberst auf dem Stack liegt und welcher Typ gewünscht wird und die Methode konvertiert den Typen dann dementsprechend. Bei einigen Casts können dabei zur Laufzeit Exceptions geworfen werden.[22]

checkCast Der Java-Bytecode-Befehl CHECKCAST ist eine Variante, um ein unbekanntes Object auf dem Stack in ein gefordertes Object zu konvertieren. Bei Nicht-Übereinstimmung wird eine Exception geworfen.[23] Solch ein Checkcast ist aber vergleichsweise aufwändig und kann in einigen Fällen optimiert werden (siehe dazu auch Kapitel 5.5). Deshalb wird anstatt eines sofortigen CHECKCAST Aufrufes diese Subroutine aufgerufen.

instanceOf Ähnlich wie die Methode checkCast, optimiert diese Subroutine eventuell nicht benötigte INSTANCEOF-Aufrufe weg. INSTANCEOF wird benötigt, um herauszufinden, ob ein unbekanntes Object auf dem Stack einem gewissen Typen entspricht.[24]

---

[22]Beispielsweise bei einem Object-to-Int-Cast, bei dem das Object keine Instance einer numerischen Klasse (beispielsweise Integer) ist.

[23]Ist in einer Operation der Typ erst zur Laufzeit bekannt, so wollen wir eine Exception natürlich vermeiden, stattdessen kann man dann im Vorraus ein instanceof aufrufen. Dieser Befehl macht einen Typcheck, auf den im Anschluss individuell reagiert werden kann.

[24]Dies wird beispielsweise beim PQL-Befehl POLY_LOOKUP benötigt: Wir haben ein Object gegeben, und sollen nun einen Wert mit gegebenem Schlüssel herausfinden. Dabei kann das Object eine Java-interne Map, eine PQL-Map (Subklasse der abstrakten Klasse AMap) oder ein Array sein. Es gibt allerdings keinen Java-Bytecode Befehl, der für all diese Fallunterscheidungen das erwünschte Ergebnis erzielt. Zwar könnte man bei Java-internen Maps und PQL-Maps die gleichen Methoden nutzen, aber auch diese zwei Maptypen werden aus Performanzgründen unterschiedlich behandelt.

loadConstant[*] Das [*] steht für *Integer, Long* oder *Double*. Dabei legt diese Methode einen konstanten Wert auf den Stack. Ihre Daseinsberechtigung erhält sie dadurch, dass einige speziellen, häufig verwendeten Werte einen speziellen, optimierten Ladebefehl haben.[25]

Zusätzlich zu diesen Basis-Subroutinen gibt es noch eine Spezial-Subroutine zum Konvertieren besonderer Typen (*insertSpecialTypeCast*):

insertSpecialTypeCast Diese Methode konvertiert den Typen — genau wie die castValue-Methode — eines Wertes auf dem Stack. Der Unterschied dabei ist, dass sie nur „spezielle“ Casts durchführt, zu diesen zählen alle Casts, bei denen Quelle oder Ziel nicht einem der vier Standardtypen *Object*, *int*, *long* oder *double* entspricht. Dabei wird diese Methode nur von der *castValue-Methode* als Hilfsroutine gebraucht und hat ansonsten keine Verwendung. Die Argumente *from* und *to*, welche den Quell- und Zieltyp angeben, könenn dabei -1 sein: In diesem Fall liegt der Wert auf dem Stack gerade im entsprechenden Basistyp vor oder soll nach der Typ-Konvertierung im diesem vorliegen. Mit Basistyp ist in dem Fall auch wieder ein *Object*, *int*, *long* oder *double* gemeint, mit dem Unterschied, dass dabei jedem „speziellem“ Typ **genau** ein Basistyp zugeordnet ist (z.B. ist *boolean* oder *short* dem Basistyp *int* zugeordnet, *float* dem Basistyp *double*, etc..). Der Cast -1 zu *boolean* erwartet beispielsweise ein *int* auf dem Stack und konvertiert ihn in ein *boolean*.

Auch in Subroutinen ausgelagert wurden dann noch die Abarbeitung einiger PQIL-Befehle, da sie im Großen und Ganzen sehr ähnlichen Code erzeugen. Dabei wurden die arithmetischen Befehle (*createArithmeticCode*) und die Vergleichsbefehle (*createComparisonCode*) ausgelagert:

createArithmeticCode Hier werden alle arithmetischen Befehle in Bytecode übersetzt, dazu zählen Addition, Subtraktion, etc.., aber auch Bitbefehle, wie zum Beispiel *And* oder *Or*.

createComparisonCode Hier werden alle Vergleichs-Befehle in Bytecode übersetzt, dazu zählen Gleichheitsüberprüfung, größergleich, etc.. Eine Besonderheit dabei ist, dass diese Routine auch innerhalb anderer Befehle

[25]Beispielsweise enthält der Java-Bytecode-Befehlssatz einen Ladebefehl für Integer-Werte zwischen -1 und 5, die Befehle heißen dabei iconst_*, wobei * folgende Werte annimmt: m1, 0, 1, 2, 3, 4, 5

benutzt wird, zum Beispiel bei manchen Zugriffsmodi im PQIL-Befehl *LOOKUP*, um Objectgleichheit zu überprüfen.

5.2.3 Kommentare im Code

Grundsätzlich war der Bytecode sehr schwer zu Debuggen. Um die Arbeit etwas zu vereinfachen, wurden noch zwei Hilfs-Methoden implementiert, *insertComment* (zum Einfügen von Kommentaren) und *insertDebugInformation* (zum Einfügen von Debug-Meldungen zur Laufzeit):

insertComment Diese Methode fügt Reintext in den erzeugten Code ein, um diesen lesbarer zu machen. Dies geschieht allerdings nur, wenn diese Option mittels boolschem Attribut *writeCommentsInCode* aktiviert ist. Das ganze funktioniert so, dass der Kommentar als String auf den Stack gelegt wird und direkt anschließend wieder vom Stack entfernt wird. Somit ändert es nichts an den Ergebnissen, die eine Ausführung vom erzeugten Bytecode produziert.

insertDebugInformation Diese Methode gibt einen Kommentar mittels *System.out.println* aus, wenn die entsprechende Stelle im Code erreicht wird. Diese Methode ist nur zu Debugging-Zwecken vorhanden und sollte in der endgültigen Fassung des RuntimeCreators nicht mehr aufgerufen werden.

5.3 Schnittstelle

Der Aufruf des *RuntimeCreators* besteht aus zwei statischen Methoden. Zum einen einer Initialisierungs-Methode:

```
1 public static void init (Join new_query, Env _initEnv)
```

Das Join *new_query* bildet den Einstieg, in welchem dann per Verschachtelung das gesamte PQIL-Programm enthalten ist. (siehe dazu auch Kapitel 3) Der Name *new_query* rührt daher, dass wir aufgrund der kompletten Benutzung statischer Methoden und Attribute (siehe dazu auch Kapitel 5.1.2), eventuelle vorherige in Bytecode übersetzte Programme überschreiben. Das *_initEnv* bestimmt das *Environment*, in welchem das Programm später ausgeführt wird und sollte nach dem Initialisieren nur noch bedingt geändert werden.[26] Die Environment-Klasse *Env* (edu.umass.pql.Env) wurde aus dem bereits existierenden PQL-Code übernommen und wird im folgenden kurz beschrieben:

Wie bereits im Kapitel 3 besprochen, arbeitet PQL intern mit den vier Datentypen *int*, *long*, *double* und *Object*. Für die Verwaltung aller benutzten Variablen ist dabei die Env-Klasse zuständig. Um auf eine Variable zugreifen zu können benötigen wir sowohl ihren Typ als auch eine Indexnummer, die sie identifiziert. Dies wird in einem Variablen-Indexierungs-Code kodiert, der sich aus folgenden Teilen zusammensetzt: Dem Typen (*int, long, double, Object*), dem Index (die vier Typen werden jeweils in einem eigenen Array gespeichert und jede Variable kann jeweils durch einen eindeutigen Index-Typ-Schlüssel identifiziert werden) und zusätzlichen Flags, beispielsweise ob es sich im jeweiligen Kontext um einen Lese- oder Schreibzugriff handelt.

Ist die Initialisierung abgeschlossen, befindet sich in der RuntimeCreator-Klasse der erzeugte Bytecode und wir müssen ihn nur noch ausführen. Dies geschieht mit einem Aufruf der folgenden Methode:

```
1 public static boolean test (Env env)
```

Das neu übergebene Environment sollte sich dabei aber nur in den Variablen

[26] Dies bezieht sich vor allem auf mindestens einmal im Code als konstant benutzte Variablen, da diese teilweise schon direkt in den Code geschrieben werden.

ändern, auf die es nie einen konstanten Zugriff innerhalb des Codes gibt. Die weiter oben vorgestellte *init-Methode* erzeugt also einmal in einer statischen Variable den entsprechenden Bytecode, der dann per *test-Methode* beliebig häufig ausgeführt werden kann — natürlich auch mit einer anderen Startbelegung der Variablen. Dies ist somit etwas anders im Vergleich zum Interface des Interpreters: Hier haben wir keinen Aufwand zum einmaligen Initialisieren, allerdings müssen wir für jeden Durchlauf zuerst das Join zurücksetzen, welches das ganze PQIL-Programm enthält (mittels *reset-Methode*) und können dann mittels *next-Methode* das Programm im Interpreter ausführen. Intern funktioniert das dann über rekursive Aufrufe von *reset*, bzw. *next* der direkt untergeordneten Joins.

Der boolsche Rückgabewert der eben vorgestellten *test-Methode* des Laufzeitübersetzers gibt dabei Auskunft über den Erfolg der Operation. Wenn *one*, bzw. *two* für eine Variable mit konstantem Inhalt 1, bzw. 2 stehen würde, gäbe folgender PQL-Code dabei beispielsweise *true* zurückgeben:

```
1 ADD_Int(?one, ?one, ?two)
```

Im Gegenzug dazu würde der folgende Code zum Beispiel *false* zurückgeben:

```
1 ADD_Int(?one, ?one, ?one)
```

Man beachte dabei, dass folgender Code wiederum *true* zurückgeben würde:

```
1 ADD_Int(?one, ?one, !one)
```

Dies liegt daran, dass es sich hier um einen anderen Zugriffsmodus handelt. In dem Fall würde in die Variable *one* das Ergebnis von 1+1 (also 2) geschrieben werden.

5.3.1 Zusätzliche Optimierung durch Erweiterungen

Die bisher beschriebene Schnittstelle ist nun also nötig, um den dynamischen Übersetzer überhaupt einsetzen zu können. Nun möchte ich aber im Weiteren noch auf zwei Erweiterungen eingehen, die es nur im dynamischen Übersetzer gibt und die man — wenn man sie nutzen möchte — **vor** Erstellung des Bytecodes mit angeben muss. Dabei handelt es sich um rein

optionale Zusatzangaben, die dem Übersetzer helfen, den erzeugten Code performanter zu gestalten.

Zum einen erhält die bereits einige Absätze vorher besprochene Variablen-Kodierung ein weiteres Flag, das sogenannte Const-Flag.[27] Ist dieses Flag gesetzt, so kann davon ausgegangen werden, dass der jeweilige Lesezugriff immer den gleichen Wert erhält, nämlich genau den, welchen die gewünschte Variable im Environment hat, welches der Initialisierungsmethode des dynamischen Übersetzers übergeben wird.

Die zweite Optimierungs-Erweiterung bildet eine Möglichkeit, Objects, deren Typen im Voraus bekannt sind und sich während der ganzen Programmausführung nicht ändern werden, fest zu legen. Dies kann der Übersetzer für Optimierungen nutzen, da eventuelle *Typ-Checks* eingespart werden können. Folgendes Beispiel demonstriert die Verwendung:

```
1 RuntimeCreator.clearTypeInformations();
2 RuntimeCreator.setTypeInformation(0, env);
```

Das *clearTypeInformations* muss dabei einmal ganz am Anfang aufgerufen werden, um eventuelle frühere, gesetzte Typ-Informationen zu verwerfen. Anschließend setzen wir mit Hilfe von *setTypeInformation* den gewünschten Typ fest. Dabei wir der Typ genommen, der im angegeben Environment das Object mit dem angegebenen Index hat. Alternativ kann man auch den Typ per String übergeben, dabei werden Paket- und Klassennamen allerdings per Slash / getrennt, beispielsweise:

```
1 RuntimeCreator.setTypeInformation(0, "java/util/Set");
```

Zu erwähnen ist hierbei dann noch, dass diese Typinformationen nicht immer angegeben werden dürfen, da sie noch nicht optimal implementiert wurden. Siehe dazu Kapitel 5.6.2.

[27] Die Flagdefinition ist — genau wie alle anderen Variablen-Kodierungs-Konstanten — in der Klasse *edu.umass.pql.VarConstants* und heißt *VAR_CONST_FLAG*.

5.4 Herausforderungen / Schwierigkeiten

Eine der größten Schwierigkeiten bestand in der korrekten Umsetzung der Programm-Ausführungs-Logik. Da PQL eine logische Sprache ist, der Java-Bytecode aber eine klassische *imperative Sprache* repräsentiert, muss der gegebene Code entsprechend transformiert werden.

Viele Probleme traten dabei bei der korrekten Implementierung der schon weiter oben besprochenen ReturnBehaviors auf (siehe auch Kapitel 5.2.1). Haben wir beispielsweise einen *contains-Befehl*, der durch ein Set iteriert, so springen nachfolgende Befehle im Falle eines Erfolgs zum nächsten Befehl und im Falle eines Nicht-Erfolgs zurück zum Schleifenkörper. Ist das ganze wiederum in einen Reductor geschachtelt, so muss der letzte Befehl innerhalb der Schleife im Erfolgsfall zum Reductor springen, um beispielsweise einen Wert in eine Datenstruktur zu speichern. Dieser Reductor muss wiederum zum Schleifenkörper zurückspringen, damit der *contains-Befehl* weitergeht.

5.5 Optimierungen

Anschließend gab es dann noch eine Phase der Optimierung. Der erste Schritt dazu war eine komplette Ausgabe des erzeugten Codes. Dies geht sehr einfach mit dem Framework ASM. Im Normalfall hat man bereits eine *ClassWriter-Instance* zum Erzeugen des Bytecodes und eine *ClassVisitor-Instance*, die man als Schnittstelle benutzt um die gewünschten Befehle einzufügen, zu „besuchen“. Ersetzen wir den *ClassVisitor* durch den sogenannten *TraceClassVisitor* (eine Subklasse von *ClassVisitor*), so wird der Code gleichzeitig auf einem angegebenen PrintWriter ausgegeben. Dies sieht dann beispielsweise folgendermaßen aus:

```
1 ClassWriter classWriter = new ClassWriter(ClassWriter.
      COMPUTE_FRAMES|ClassWriter.COMPUTE_MAXS);
2 PrintWriter printWriter = new PrintWriter(System.out);
3 ClassVisitor classVisitor = new TraceClassVisitor(
      classWriter, printWriter);
```

Die Optimierungen reichten von ganz simplen Ersetzungen von Befehlsketten durch effizientere, aber mit equivalenter Bedeutung bis hin zu komplizierteren Strukturveränderungen. Im nachfolgenden werde ich die wichtigsten Optimierungen aufzählen und kurz beschreiben (in der gleichen Reihenfolge, wie sie auch erkannt und implementiert wurden):

* **Environment-Zugriffs-Minimierung.** Zu Beginn lief jeder Lese- und Schreibzugriff auf eine Variable über das Environment (*Env*). Dies war erst einmal die schnellste Möglichkeit, die erwünschten Befehle zu implementieren, ist aber Laufzeit-technisch gesehen teuer. Man muss beachten, dass jeder Zugriff auf ein externes Object zugreifen muss, hieraus ein Attribut auslesen, dann die Operation ausführen und schließlich noch im Regelfall das Ergebnis zurückschreiben. Wird dieser Befehl dann auch noch innerhalb einer *contains-* oder *get-Schleife* ausgeführt, so summiert sich dieser Performanz-Verlust schnell tausendfach auf. Um dies zu verhindern, wurde der Laufzeitübersetzer so abgeändert, dass die internen Berechnungen nun auf die lokalen Variablen der erzeugten Bytecode-Methode ausgelagert wurden. Dabei werden alle Env-Variablen, die mindestens einmal benutzt werden, zu Beginn aus dem Environment ausgelesen und in eine separate lokale Variable gespeichert.

Alle Lese- und Schreibzugriffe innerhalb des Codes wurden anschließend auf diesen lokalen Variablen durchgeführt und erst ganz am Ende wurden all diese Variablen wieder zurück in das Environment geschrieben.

* **Vermeidung hoher Indices lokaler Variablen.** Jede lokale Variable im *Methoden-Frame* hat ihren eigenen Index. Direkt zu Beginn gab es einen globalen Zähler, der bei jeder neu benötigten lokalen Variable um eins erhöht wurde und somit geschah die Index-Verteilung iterativ. Bei der zuvor beschriebenen Optimierung wurden plötzlich neue lokale Variablen gebraucht, die im ganzen Code verstreut benötigt wurden. Aus implementierungs-technischen Details war die erste, am einfachsten zu realisierende Lösung, auf die neu benötigten Variablen-Indices einen festen Offset zu addieren (in meinem Fall 1000), in der Annahme, dass alle anderen lokalen Variablen nie diesen Offset überschreiten würden. Dies ist natürlich zum einen nicht unbedingt ein guter Programmierstil, zum anderen wirkten sich die hohen Indices tatsächlich sehr negativ auf die Performanz aus.[28] Im Folge darauf wurden die Indices der temporären Environment-Variablen, genau wie die bereits vorhandenen, inkrementell alloziert.

* **Konstanten-Inlining.** Bisher konnten Konstanten nur benutzt werden, indem man sie wie normale Variablen lud, welche immer den gleichen Wert enthielten und somit für jeden Zugriff auf die Konstante ein Lesezugriff vonnöten war. Im endgültigen Maschinencode wäre also ein Speicherzugriff oder zumindestens ein Zugriff auf ein Register vonnöten[29]. Durch das neu hinzugefügte Zugriffs-Flag *VAR_CONST_FLAG* kann man nun konstante Zugriffe markieren. Im Code wird nun statt dem Lesezugriff die Konstante mit einem anderen Java-Bytecode-Befehl direkt geladen.

* **Inkrement statt Addition.** Eine simple Iteration wurde zuerst durch eine Addition mit 1 implementiert. Dabei gibt es auch einen

[28] Der genaue Grund dafür ist mir nicht bekannt, es ist aber davon auszugehen, dass der höhere Speicherverbrauch sich negativ auf das *Caching* auswirkt. Auch denkbar wäre, dass die *JVM* durch die höheren Indices eventuell mehr Schwierigkeiten hat die begrenzt verfügbaren Register optimal auf die benutzten lokalen Variablen zu verteilen.

[29] Die Annahme ist hierbei natürlich, dass die Java-Laufzeitumgebung dies nicht heraus optimiert. Im Zweifelsfall sollte man aber immer davon ausgehen, dass die Java VM eine Optimierung in einigen Fällen nicht automatisch hinbekommt.

eigenen Increment-Befehl, den man stattdessen nutzen kann.

* **Keine Stack-Tausch-Operationen.** Es gibt einen Bytecode-Befehl namens *SWAP*. Dieser tauscht die beiden obersten Werte auf dem Stack aus. Dieser Befehl ist aber nicht unbedingt perfomant und sollte versucht werden zu vermeiden, was mit einer Ausnahme auch gelang. In einer initialen Abschätzung des Speedups dieser Optimierung beobachten wir im Schnitt ungefähr eine Steigerung von 5%, in einer Messung sogar eine Steigerung von über 15%.

* **Möglichst selten Typecasts nutzen.** Zu Beginn befanden sich viele *CHECKCAST* und *INSTANCEOF* Befehle im generierten Bytecode. Diese überprüfen, ob das oberste Object auf dem Stack dem angegebenen Typ entspricht und geben dies entweder als boolschen Wert zurück (im Falle von *INSTANCEOF*) oder probieren, das Object entsprechend zu konvertieren (im Falle von *CHECKCAST*). Diese Operationen sind relativ langsam und sollten — wenn möglich — vermieden werden. Dies gelang teilweise durch eine manuelle Angabe statischer Typ-Informationen, bevor der Code generiert wird. Siehe dazu auch Kapitel 5.3.1. Zu erwähnen sei hier noch, dass diese „manuellen“ Angaben zukünftig auch automatisiert ermittelt werden können, da das Compiler-Frontend dies in der Praxis bereits herausfindet.

* **Spezielle strukturelle Optimierungen.** Zusätzlich zu den bereits genannten Optimierungen fanden sich manchmal im erzeugten Code unnötige Befehlszeilen, zum Beispiel aufgrund spezieller Zugriffsmodi von Joins, redundanter Nutzung von lokalen Variablen, etc.. Dieser Performanz-Ballast wurde, wenn erkannt, entfernt.

5.6 Fortschritts-Standpunkt

5.6.1 Implementierungsumfang

Bei der Implementierung habe ich mich zuerst auf die zentrale Funktionalität von PQL konzentriert. Dabei sollte man beachten, dass noch nicht der komplette Sprachumfang im dynamischen Übersetzer verfügbar ist. Allerdings konnten alle vier Beispielprogramme, die auch in der offiziellen Dokumentation [9] — vorrangig für Perfomanzmessungen — vorgestellt wurden, problemlos mit dem Laufzeitübersetzer ausgeführt werden.

Um die Frage nach dem Implementierungsumfang nun noch einmal exakt zu beantworten, hier noch einmal die komplette Liste aller verfügbaren Joins:

Befehlsliste	**Zugriffsmodi**	**Bedeutung**
ADD_Int, ADD_Long, ADD_Double, SUB_Int, SUB_Long, SUB_Double, MUL_Int, MUL_Long, MUL_Double, DIV_Int, DIV_Long, DIV_Double, MOD_Int, MOD_Long, BITOR_Int, BITOR_Long, BITAND_Int, BITAND_Long, BITXOR_Int, BITXOR_Long, BITSHL_Int, BITSHL_Long, BITSHR_Int, BITSHR_Long, BITSSHR_Int, BITSSHR_Long	?x ?y !z	z = x [*] y, das Sternchen wird dabei durch die jeweilige Berechnung ersetzt (bei ADD beispielsweise durch +)
	?x ?y ?z	z == x [*] y, das Sternchen wird dabei durch die jeweilige Berechnung ersetzt (bei ADD beispielsweise durch +)
NEG_Int, NEG_Long, NEG_Double, BITINV_Int, BITINV_Long	?x !y	y = [*] x, das Sternchen wird dabei durch die jeweilige Berechnung ersetzt
	?x ?y	y == [*] x, das Sternchen wird dabei durch die jeweilige Berechnung ersetzt
EQ_Int, EQ_Long, EQ_Double, EQ_Object, EQ_String	?x ?y	x == y
	?x !y	y = x

NEQ_Int, NEQ_Long, NEQ_Double, NEQ_Object, NEQ_String, LT_Int, LT_Long, LT_Double, LTE_Int, LTE_Long, LTE_Double	?x ?y	x [*] y, das Sternchen wird dabei durch den jeweiligen Vergleichsoperator ersetzt
INT_RANGE_CONTAINS, LONG_RANGE_CONTAINS	?from ?to !x	Zählt von *from* bis *to* in der Variable x.
	?from ?to ?x	Überprüft, ob sich x innerhalb von *from* bis *to* befindet.
COERCE_Boolean, COERCE_Byte, COERCE_Short, COERCE_Char, COERCE_Float	?x ?y	Konvertiert x in den angegebenen Typen und schreibt das Ergebnis in y.
FIELD	?obj !x	Liest ein angegebenes Attribut aus dem Java-Object *obj* und schreibt das Ergebnis in x.
ConjunctiveBlock	—	Fasst mehrere Joins zu einem konjunktiven Block zusammen.
CONTAINS	?obj ?x	Überprüft, ob im Set *obj* Wert x enthalten ist.
	?obj !x	Iteriert durch das ganze Set *obj* und schreibt die Werte in x.

LOOKUP, POLY_LOOKUP, ARRAY_LOOKUP_Int, ARRAY_LOOKUP_Long, ARRAY_LOOKUP_Short, ARRAY_LOOKUP_Char, ARRAY_LOOKUP_Boolean, ARRAY_LOOKUP_Byte, ARRAY_LOOKUP_Float, ARRAY_LOOKUP_Double, ARRAY_LOOKUP_Object	?obj ?x ?y	Überprüft, ob in der Map / im Array *obj* der Schlüssel x mit dem Wert y enthalten ist.
	?obj ?x !y	Schreibt alle zum Schlüssel x der Map / des Arrays *obj* zugehörigen Werte in y.
	?obj !x ?y	Schreibt alle Schlüssel mit Wert y der Map / des Arrays *obj* in x.
	?obj !x !y	Schreibt alle Schlüssel-Wert-Kombinationen der Map / des Arrays *obj* in x und y.
	?obj _ !x	Schreibt alle vorhandenen Werte der Map / des Arrays *obj* in x.
	?obj !x _	Schreibt alle vorhandenen Schlüssel der Map / des Arrays *obj* in x.

Zusätzlich ist dann hier nochmal eine Liste aller Reductors, die derzeit vom Laufzeitübersetzer unterstützt werden:

ARRAY, INT_ARRAY, LONG_ARRAY, DOUBLE_ARRAY, OBJECT_ARRAY, BYTE_ARRAY, CHAR_ARRAY, SHORT_ARRAY, BOOLEAN_ARRAY, FLOAT_ARRAY, MAP	?x ?y !obj	Dieser Reductor schreibt den Schlüssel x mit dem Wert y in die Map / das Array *obj*.

DEFAULT_MAP	?dflt ?x ?y !obj	Dieser Reductor schreibt den Schlüssel x mit dem Wert y in die Default-Map[30] *obj* mit dem Default-Wert *dflt*.
N_MAP	?x !obj	Dieser Reductor nimmt den Schlüssel x, ruft mit diesen einen angegebenen Sub-Reductor aus[31], erhält durch ihn den dazugehörigen Wert und schreibt Schlüssel und Wert in die Map *obj*.
N_DEFAULT_MAP	?dflt ?x !obj	Dieser Reductor nimmt den Schlüssel x, ruft mit diesen einen angegebenen Sub-Reductor aus[31], erhält durch ihn den dazugehörigen Wert und schreibt Schlüssel und Wert in die Default-Map[30] *obj* mit dem Default-Wert *dflt*.
SET	?x !obj	Dieser Reductor schreibt den Wert aus x in das Set *obj*.
METHOD_ADAPTER	?x !y	Dieser Reductor nimmt den Wert x und y, leitet sie an eine angegebene Methode weiter und schreibt das Ergebnis zurück in y. Dies kann man zum Beispiel dazu nutzen, alle Ergebnisse, die dieser Reductor bekommt, aufzusummieren.
EXISTS	?x !y	Dieser Reductor schreibt x in y, wenn im umschlossenen Join mindestens einmal der Erfolgsfall eintritt.

Noch nicht implementiert, aber eigentlich zum PQL-Sprachumfang gehörend sind folgende PQIL-Befehle: POLY_SIZE, SET_SIZE, INT_ARRAY_SIZE, LONG_ARRAY_SIZE, SHORT_ARRAY_SIZE, BOOLEAN_ARRAY_SIZE, CHAR_ARRAY_SIZE, BYTE_ARRAY_SIZE, FLOAT_ARRAY_SIZE, DOUBLE_ARRAY_SIZE, OBJECT_ARRAY_SIZE, MAP_SIZE (um die Größe eines Sets, Maps oder Arrays herauszubekommen), JAVA_TYPE, INT, LONG, BOOLEAN, BYTE, SHORT, CHAR (zum Überprüfen, ob ein Wert innerhalb des Wertebereichs des angegebenen Typs liegt, aber auch zum Iterieren durch alle möglichen Werte im entsprechenden Wertebereich), INSTANTIATE (zum Instanzieren neuer Objekte), DisjunctiveBlock (fasst mehrere Joins zu einem disjunktiven Block zusammen) und FORALL (ein Reductor, um herauszufinden ob alle Durchläufe des umschlossenen Blocks erfolgreich durchlaufen).

Um ein kurzes persönliches Resümee aus dem Implementierungsumfang zu ziehen: Grundsätzlich ist die Programmierung eines solchen dynamischen Übersetzers keine Unmöglichkeit, verschlingt aber viel Zeit — vor allem mehr, als von mir ursprünglich erwartet. Dies ist dann auch der Grund, wieso nicht der komplette Befehlsumfang im Rahmen dieser Arbeit umgesetzt wurde. Ich sehe den Hauptgrund des hohen Zeitaufwandes darin, dass wir uns doch — gezwungenermaßen — auf einer sehr niedrigen Programmierebene befinden, welche lange unübersichtliche Code-Abschnitte nach sich zieht und außerdem vergleichsweise schwer zu debuggen ist. Gerade der zweite Punkt ist natürlich auch im Großen und Ganzen eine Frage der Erfahrung, da Java auf *Low-Level-Ebene* einige Eigenschaften besitzt, die auf der *High-Level-Ebene*

[30] die Default-Map funktioniert im Prinzip wie eine normale Map, mit dem Unterschied, dass sie einen Default-Wert besitzt. Fragt man nun einen Schlüssel ab, der in der Default-Map nicht vorhanden ist, so wird statt *null* der Default-Wert zurückgegeben.

[31] Diese Map-Reductors, welche einen weiteren Sub-Reductor aufrufen, wurden ursprünglich aus Performanz-Gründen eingeführt. Siehe dazu auch die offizielle PQL-Doku, Seite 14. [9]

nicht direkt ersichtlich sind.[32]

5.6.2 Schnittstellen zur Erweiterung

Ein Hauptschwerpunkt, an dem man wohl am ehesten den dynamischen Übersetzer noch ausbauen wird, wäre wohl die Erweiterung des unterstützten Sprachumfangs durch weitere Befehle aus PQIL und eventuell Unterstützung weiterer Zugriffsmodi schon implementierter PQIL-Befehle. Die meisten neuen Befehle würden wohl die *switch-case-Anweisung* in folgender Methode im RuntimeCreator erweitern:

```
1 private static void createSpecialInstructionCode (
      MethodVisitor mv, Join query, int flags) throws
      Exception
```

Der *MethodVisitor mv* wird dabei benötigt, um weiteren Code in die erzeugte Bytecode-Methode zu schreiben. Das *Join query* beinhaltet das derzeit zu übersetzende Join. Die Flags schließlich beinhalten Informationen zu dem gerade zu übersetzenden Join. Dabei wird ihr Inhalt direkt am Anfang schon in Typ (*type*) und Instruktion (*instruction*) getrennt. Die Variable *instruction* beinhaltet den Namen des Joins und der Typ beschreibt bei manchen Instruktionen den mit angegebenen Typen. *ADD_Int* würde beispielsweise in *TYPE_ADD* als Instruktion und *TYPE_INT* als Typ gesplittet werden. Damit die Flags bei eventuellen neu zu implementierenden PQIL-Befehlen auch funktionieren, müssten diese allerdings noch in der *Visitor-Klasse* nachgetragen werden, siehe dazu auch Kapitel 5.1.1.

Auch ein großer Punkt, der im dynamischen Übersetzer noch nicht realisiert wurde, ist die fehlende Möglichkeit zur Parallelisierung. Zum derzeitigen Standpunkt sind Programme des Laufzeitübersetzers nur sequentiell ausführbar. Beim PQL-Compiler hingegen ist die Parallelisierung implemen-

[32]Ein ganz häufiges Problem stellte beispielsweise die korrekte Verwendung von *CHECKCAST* dar. Dabei erwartet jeder Befehl, der auf einem Object vom *operand stack* arbeitet, dass diese Instanz im korrekten Typ vorliegt. In manchen Fällen ist der Typ bereits bekannt und muss nicht extra konvertiert werden (wurde beispielsweise ein Element aus einem zweidimensionalen Object-Array geladen, so liegt dieses bereits als eindimensionales Object-Array vor), in den meisten Fällen ist ein Cast-Befehl aber notwendig. Wann genau ein *CHECKCAST* nun aber notwendig war oder nicht erschloss sich mir nicht immer auf den ersten Blick, was dementsprechend auch manchmal erst einmal zu Fehlern geführt hatte.

tiert, sie funktioniert so, dass im Vorherein entschieden wird, welche Teile des Programms in welchem Thread bearbeitet werden. Betrachten wir die Klasse *CustomParallelDecoratorJoin*, eine Subklasse von *Join* und insbesondere auch *ControlStructure*, so finden wir unter anderem folgende Attribute:

```
1 protected int range_pos;
2 protected int range_stop;
```

Diese geben bei einem durchzuiterierendem PQIL-Befehl (beispielsweise einem *contains-Befehl*) an, welcher Bereich vom derzeitig aktiven Thread abgedeckt wird. Dies müsste vom dynamischen Übersetzer noch implementiert werden. Anschließend müssen die einzeln berechneten Bereiche noch zusammengefasst werden, die Funktionalität dafür wurde aber auch für den herkömmlichen Interpreter schon benötigt.

Weitere Verbesserungen wären im Bereich des Bug-Fixing (selbstverständlich), aber auch im Bereich einer besseren Schnittstelle denkbar. Beim Schreiben des zweiten Punkt dachte ich in erster Linie an die derzeit relativ unsaubere Implementierung der Angabe bekannter Object-Typen (vergleiche dazu auch den zweiten Punkt im Kapitel 5.3.1). Die zusätzlichen Typinformationen sind dabei nämlich noch nicht optimal umgesetzt und dürfen nicht in allen Fällen benutzt werden. Haben wir beispielsweise eine Variable, die in einer *contains* oder *get* Methode immer wieder neu gesetzt wird, so muss diese bei jeder Iteration zwingend neu konvertiert werden und darf nicht per Typ-Information festgesetzt werden. Wird dies dennoch gemacht, kann es dazu führen, dass das Programm nicht ausführbar ist. Da eine Umstellung auf ein System, dass vollautomatisch erkennt, ob eine solche Konvertierung — trotz bekannter Typinformation — zwingend erforderlich ist, zu viel Zeit im Vergleich zum Nutzen gebraucht hatte, wurde entschieden dies erstmal nicht zu priorisieren.

Auch noch denkbar wäre es den Laufzeitübersetzer in der Richtung zu erweitern, dass er in der Lage ist mehrere Bytecode-Übersetzungen gleichzeitig zu speichern und diese dann auch in beliebiger Reihenfolge mehrfach ausführen zu können. Derzeit überschreibt jeder Übersetzungsvorgang des Laufzeitübersetzers den vorherig erzeugten Bytecode.

6 Auswertungen

6.1 Überblick über die Messungen

Die Messungen lassen sich in drei Gruppen einteilen. In der ersten Gruppe wurde eine gegebene Problemstellung jeweils einmal in klassischem Java-Code, im Laufzeitübersetzer und im Compiler übersetzt und anschließend ausgeführt. Dabei wurde dies mit 4 verschiedenen Benchmarks gemacht, welche bereits in der offiziellen Dokumentation von PQL [9] eingeführt worden. Namentlich handelt es sich dabei um die Benchmarks: *threegrep*, *bonus*, *webgraph* und *idf*.[33]

Bei der zweiten Gruppe handelt es sich um Messungen, die den perfomanztechnischen Nutzen vom Laufzeitübersetzer demonstrieren sollen. Hierfür wurden drei Benchmarks entwickelt: *arraynested*, *setnested* und *mapnested*. Dabei laufen Sie immer sehr ähnlich ab: Wir haben zwei Ganzzahlenbeinhaltende Datenstrukturen desselben Typs[34] und es werden alle Zahlen gesucht, die kleiner als 10 sind und sich gleichzeitig in beiden Datenstrukturen befinden. Betrachten wir die Ausführung iterativ, so kann man von einer äußeren und einer inneren Schleife sprechen, dies wird nochmal kurz in folgendem Pseudo-Code skizziert:

```
1 Set RESULTS = new Set
2 foreach ELEMENT in DATENSTRUKTUR1   //die 'äußere'
     Schleife
3     if (ELEMENT < 10)
4         foreach OTHER_ELEMENT in DATENSTRUKTUR2 //die '
              innere' Schleife
5             if (OTHER_ELEMENT == ELEMENT)
6                 RESULTS.add(ELEMENT)
```

[33] *idf* hieß im Original noch *wordcount*, diese beiden Benchmarks machen aber im Prinzip fast dasselbe.

[34] Bei diesen Datenstrukturen handelt es sich logischerweise entsprechend der Benchmark-Namen jeweils um *Arrays*, *Sets* und *Maps*.

In unseren drei Benchmarks ist dabei die Datenstruktur, welche die äußere Schleife bildet, sehr viel größer[35] als jene der inneren Schleife, sodass der Code bei der Ausführung zuerst einmal sehr viele Ganzzahlen durchlaufen muss, obwohl diese sich ohnehin nicht in der Datenstruktur der inneren Schleife befinden. Die Messungen wurden nicht nur auf dem Laufzeitübersetzer, sondern auch mit dem Interpreter und dem Compiler durchgeführt. Dabei wurde zuerst die Messung nur mit statischen Optimierungen[36] durchgeführt und anschließend sowohl auf dem Interpreter, als auch auf dem Laufzeitübersetzer mit dynamischer Optimierung wiederholt. Die Messungen des Compilers konnten selbstverständlich nicht mit dem dynamischen Optimierer durchgeführt werden, da der Compiler den Code nicht zur Laufzeit erzeugt, bzw. interpretiert. Dieser Messung gegenübergestellt wird dann der Durchlauf mit dynamischer Optimierung, bei der im Ergebnis die beiden Datenstrukturen getauscht werden und dementsprechend eine starke Performanzsteigerung zu erwarten ist.

In der dritten Gruppe wurde schließlich noch die Laufzeitentwicklung gemessen, wenn wir die Größen der beteiligten Datenstrukturen vergrößerten. Dabei wurden wieder die drei Benchmarks *arraynested*, *setnested* und *mapnested* verwendet. Die Messungen unterscheiden sich dabei nur in der Größe der bereits größeren Datenstruktur (von der äußeren Schleife), dabei wurden die Messungen je einmal mit Faktor 1, 10 und 100 gemacht. Außerdem wurde der Zufallsbereich der beinhaltenden Zufallszahlen so abgeändert, dass im Erwartungswert in allen Messungen die absolut gleiche Menge der Zahlen in der Datenstruktur der äußeren Schleife kleiner als 10 ist. Möchte man diese Messungen mit denen der zweiten Gruppe vergleichen, so sollte man dabei beachten, dass die Messungen mit Faktor 100 (und nicht mit Faktor 1) denen der zweiten Gruppe entsprechen.

Zusätzlich dazu wurde bei jeder Messung noch die Erstellungszeit des Laufzeitübersetzers gemessen, also die Zeit, welche noch zuätzlich zur Ausführungszeit zum Erstellen des Java-Bytecodes zur Laufzeit benötigt wurde.

[35]Dabei handelt es sich beim *Set-* und *Array-Benchmark* um einen Faktor 100 000, beim *Map-Benchmark* um einen Faktor 100. Die genauen Größen der Datenstrukturen sind dabei 100 und 10 000 000 Einträge (*Set, Array*), bzw. 10 und 1000 Einträge (*Map*). Die Datenstrukturen des *Map-Benchmark* musste stark verkleinert werden, da sonst eine der Messungen zu lange gedauert hätte. Siehe dazu auch Kapitel 6.3, bzw. Kapitel 6.4 für die genaue Begründung der hohen Laufzeit.

[36]Dies sind all die Optimierungen, die wir machen können ohne konkrete Laufzeitinformationen zu haben.

Alle Messungen wurden dabei auf 6 verschiedenen Platformen durchgeführt:

* JDK 1.6, virtuelle Maschine: *Server*
* JDK 1.6, virtuelle Maschine: *JamVM*
* JDK 1.6, virtuelle Maschine: *Cacao*
* JDK 1.7, virtuelle Maschine: *Server*
* JDK 1.7, virtuelle Maschine: *JamVM*
* JDK 1.8, virtuelle Maschine: *Server*

Dabei wurden alle Messungen auf dem gleichen, eigens dafür eingerichteten Server vorgenommen. Dieser hatte eingebaut einen 4-Core 2x Hyperthread Xeon E3-1230 3.3 GHz (x86_64), 16 GB RAM, beim Betriebssystem handelte es sich um ein Debian GNU/Linux 7.1. Die Auslastung anderer Prozesse im Hintergund wurde auf ein Minimum reduziert und alle Messungen wurden nacheinander durchgeführt.

Jede Messung wurde dabei 10 mal hintereinander durchgeführt, zusätzlich dazu wurden vorher 3 Messungen durchgeführt und direkt verworfen um eventuelle Aufwärmeffekte zu verhindern. Bei jeder Messung wurde dabei unterschieden zwischen Initialisierungszeit und Ausführungszeit. Zur Initialisierungszeit gehörten das notwendige Zurücksetzen des Interpreters und die Bytecode-Erstellung des Laufzeitübersetzers. In den Ergebnisgraphen werden dabei primär nur die eigentlichen Ausführungszeiten betrachtet. Die Graphen, welche jeweils die Erstellungszeit des Laufzeitübersetzers betrachten, wurden hingegen aus den Initialisierungszeiten erstellt.

6.2 Die Benchmarks in PQL-Code

Hier werden nun nochmal alle sieben Benchmarks in ihrem jeweiligen PQL-Code dargestellt:

```
1 //threegrep
2 query(Set.contains(byte[] ba)): exists i: array[i] == ba
3     && exists j:
4        ba[j] == ((byte)'0')
5     && ba[j + 1] == ((byte)'1')
6     && ba[j + 2] == ((byte)'2')
```

```
7     && range(0, RECORD_SIZE - 3).contains(j)
8
9  //bonus
10 query(Map.get(employee) == double bonus):
11    employees.contains(employee)
12    && bonus ==
13        employee.dept.bonus_factor
14        * (reduce(sumDouble) v:
15       exists Bonus b: employee.bonusSet.contains(b)
16       && v == b.bonus_base)
17
18 //webgraph
19 query(Set.contains(Webdoc doc)):
20           documents.contains(doc)
21     && exists link: doc.outlinks.contains(link)
22     && exists link2 : link.destination.outlinks.contains(
          link2)
23     && link2.destination == doc
24
25 //idf
26 query(Map.get(int word_id) == int idf default 0):
27    idf == reduce(sumInt) one over doc: one == 1
28       && documents.contains(doc)
29       && exists i: doc.words[i] == word_id
30
31 //arraynested
32 query(Set.contains(int x)):
33     generatedArray1[y] == x
34     && x < 10
35     && generatedArray2[z] == x
36
37 //setnested
38 query(Set.contains(int x)):
39     generatedSet1.contains(x)
40     && x < 10
41     && generatedSet2.contains(x)
42
43 //mapnested
44 query(Set.contains(int x)):
45     range(0,999).contains(y)
46     && generatedMap1.get(y) == x
```

```
47      && x < 10
48      && range(0,9).contains(z)
49      && generatedMap2.get(z) == x
```

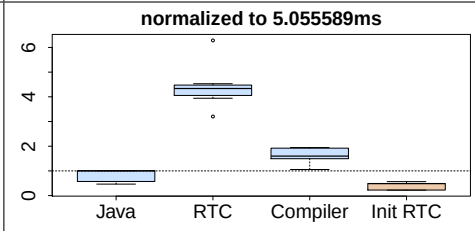
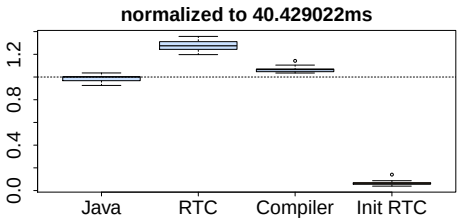
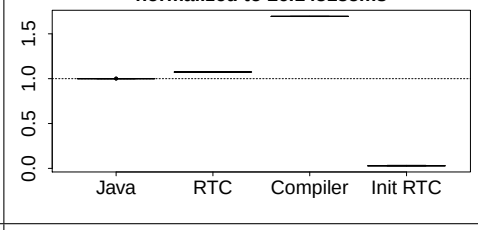
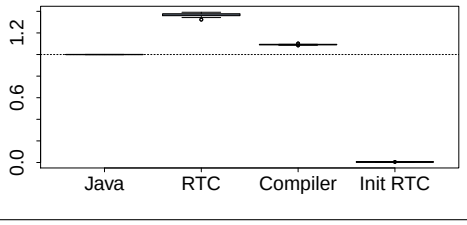
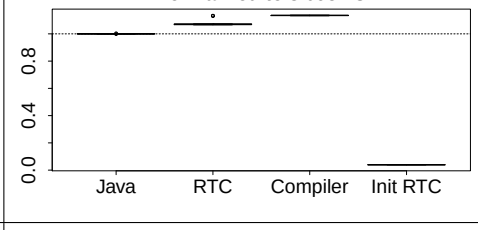
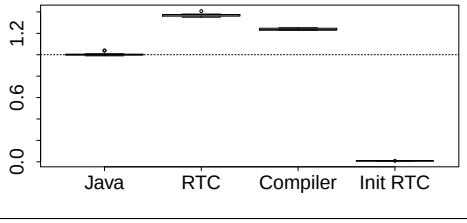
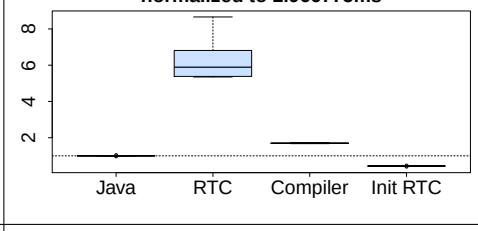
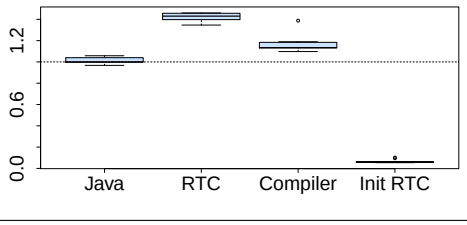
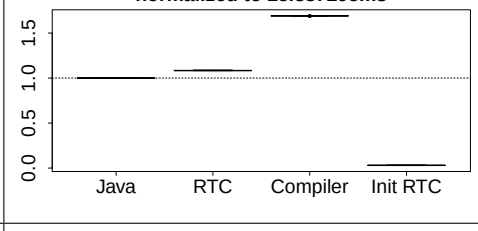
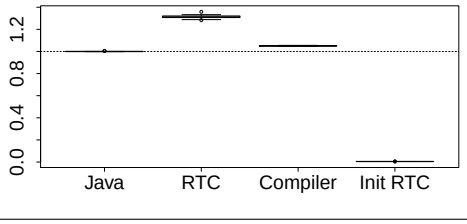
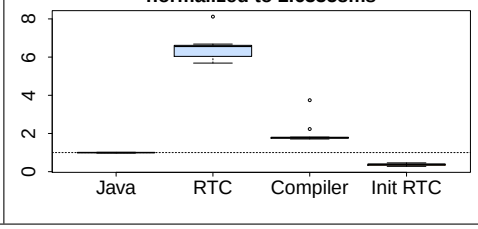
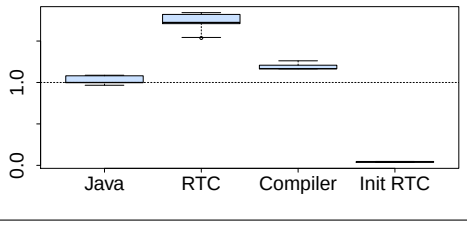
6.3 Ergebnis-Graphen

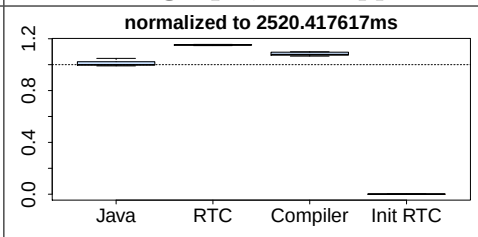
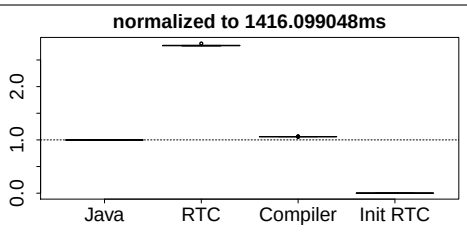
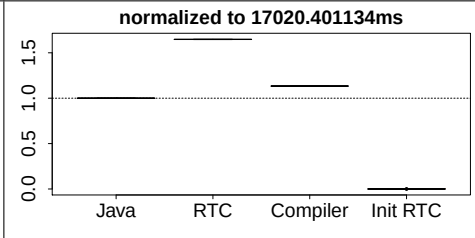
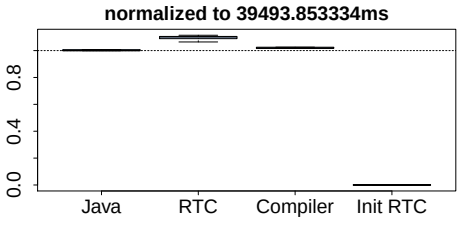
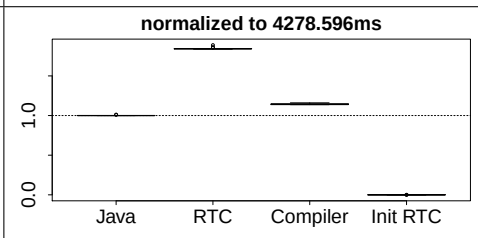
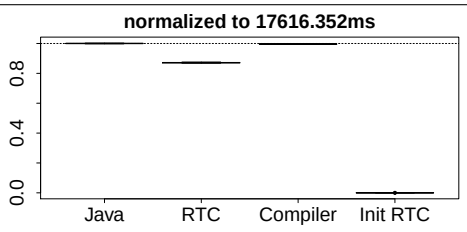
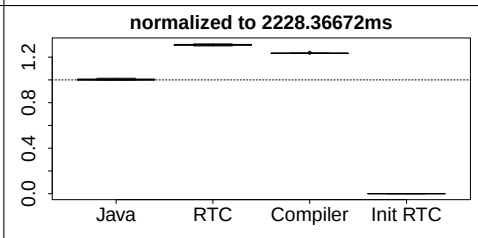
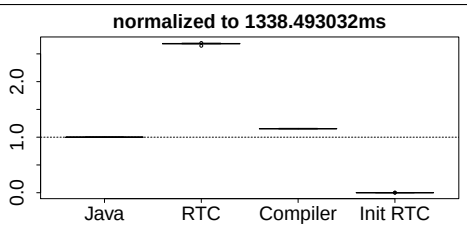
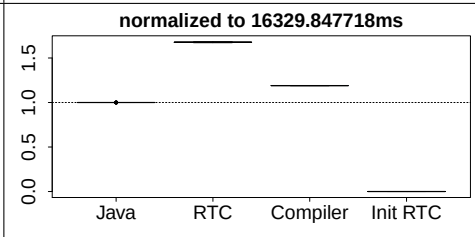
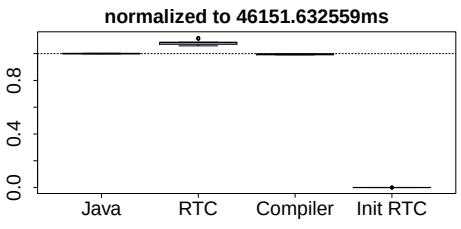
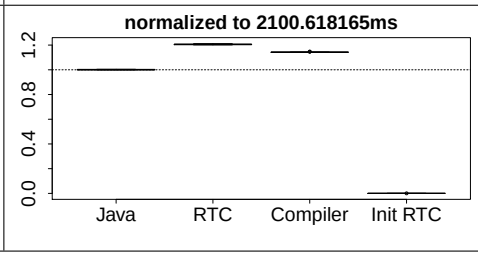
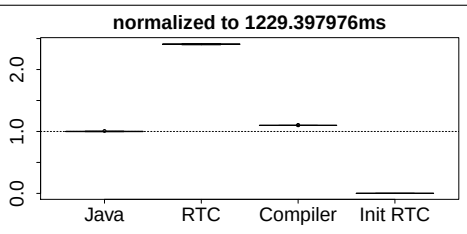
Es folgen die Graphen, in welchen die Messungen dargestellt werden. Bei den Graphen der ersten und zweiten Gruppe handelt es sich dabei um Boxplots.[37] Die Graphen der dritten Gruppe geben im Gegenzug nur Auskunft über den Median der jeweiligen Messungen. Sämtliche Werte — sowohl in den ersten beiden, als auch in der dritten Gruppe — an der linken Achse sind dabei auf das *Median* der ersten Messung (jeweils ganz links) normalisiert. Im Klartext heißt das: Wir finden das Median der ersten Messung immer bei 1.0 und alle anderen Werte sind als relative Werte im Vergleich zu diesem Median zu betrachten. Die Position von 1.0 ist dabei immer als horizontale, gestrichelte Linie noch einmal hervorgehoben. Welcher Zeit nun der normalisierte Median entspricht, wird bei den jeweiligen Graphen immer ganz oben als Millisekunden angegeben (ein Wert X an der linken Spalte enstpricht also einer Zeitdauer von $X*$[angegebener Normalisierungs-Zeit in ms]).

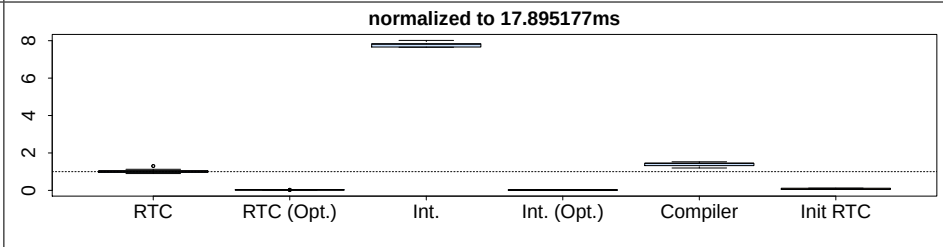
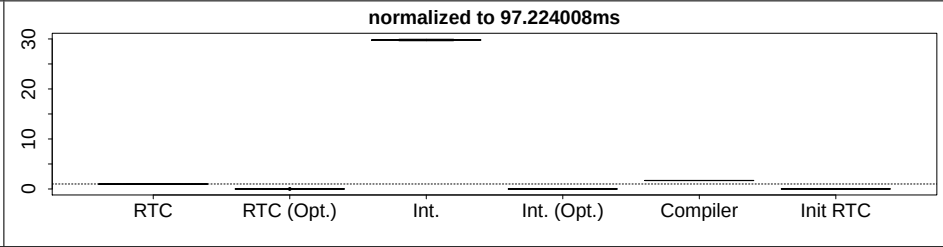
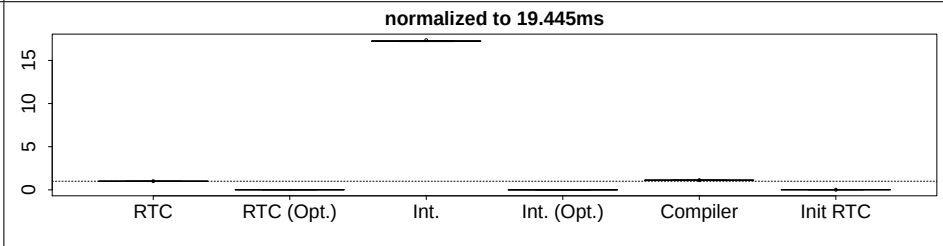
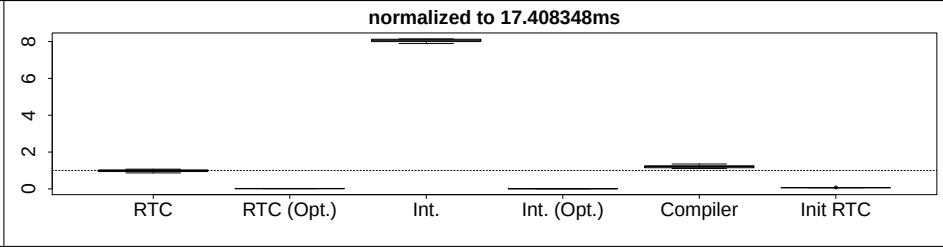
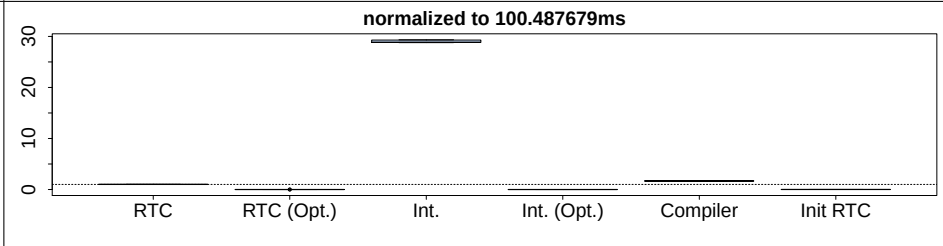
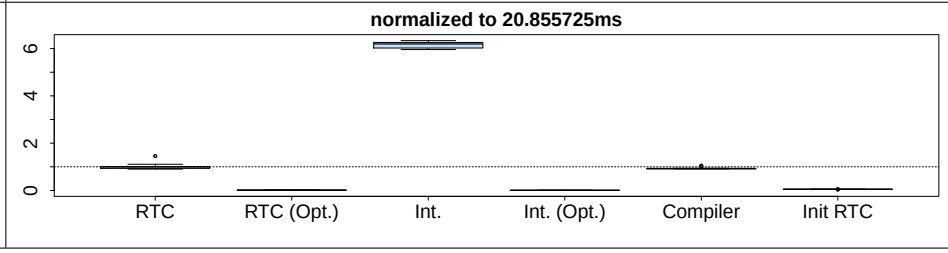
Die Zeilen unterscheiden sich dabei jeweils in der Platform, welche benutzt wurde. Die Spalten geben an, um welchen Benchmark konkret es sich handelt.

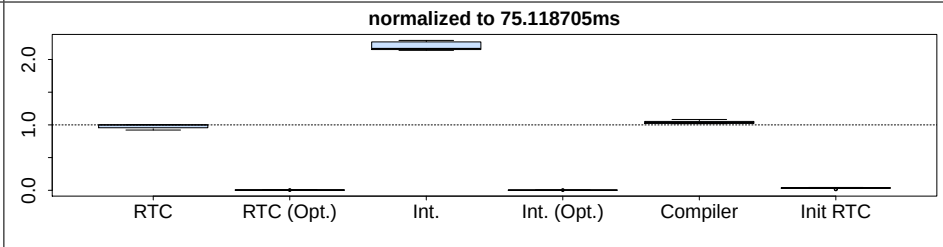
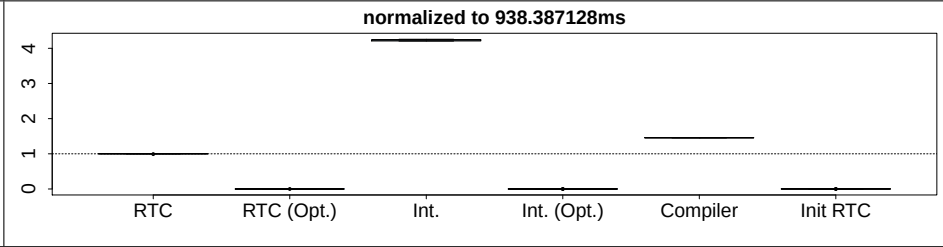
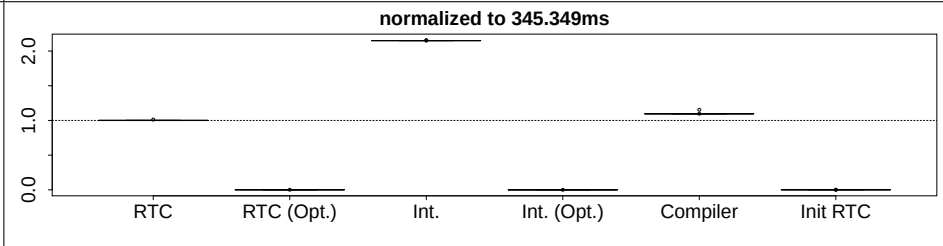
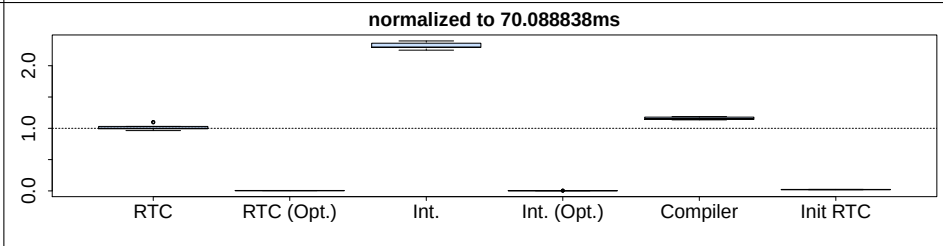
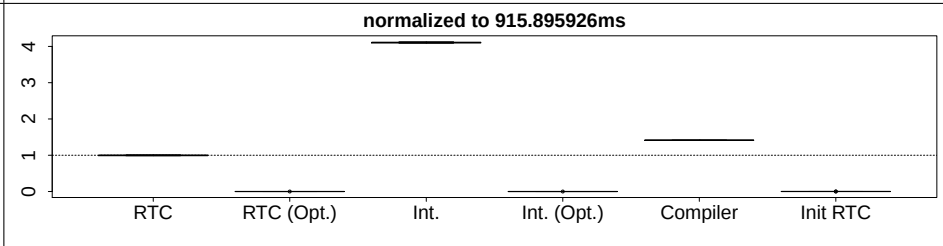
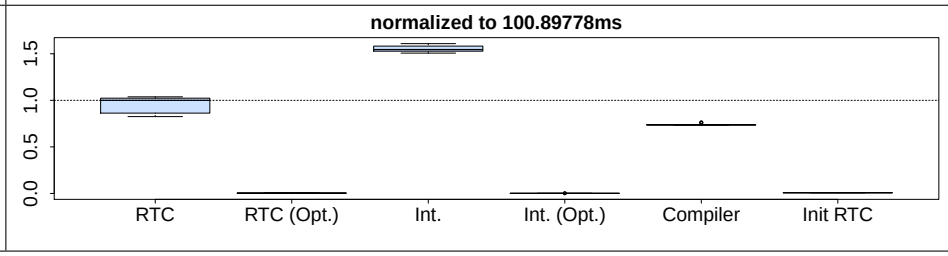
In der ersten Gruppe wurde der manuelle Java-Code (**Java**), der Laufzeitübersetzer (**RTC** (RunTimeCreator)), der Compiler (**Compiler**) und die Erstellungszeit des Java-Bytecodes vom Laufzeitübersetzer (**Init RTC**) gegenübergestellt. In der zweiten Gruppe wurde der Laufzeitübersetzer nur mit statischer Optimierung (**RTC**), der Laufzeitübersetzer mit dynamischer Optimierung (**RTC (Opt.)**), der Interpreter nur mit statischer Optimierung (**Int.**), der Interpreter mit dynamischer Optimierung (**Int. (Opt.)**), der Compiler (**Compiler**) und die Erstellungszeit des Java-Bytecodes vom Laufzeitübersetzer (**Init RTC**) gegenübergestellt. In der dritten Gruppe wurde nur der Laufzeitübersetzer mit dynamischer Optimierung (**RTC (Opt.)**) und der Compiler (**Compiler**) verglichen.

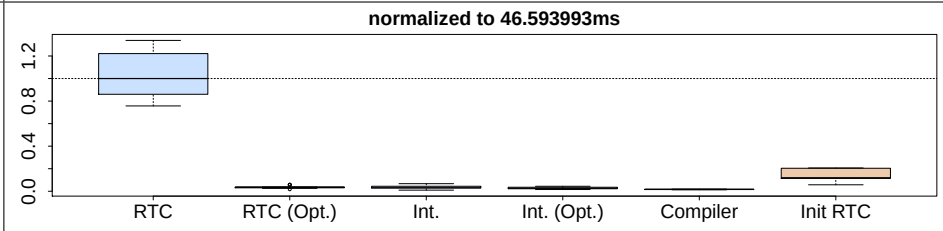
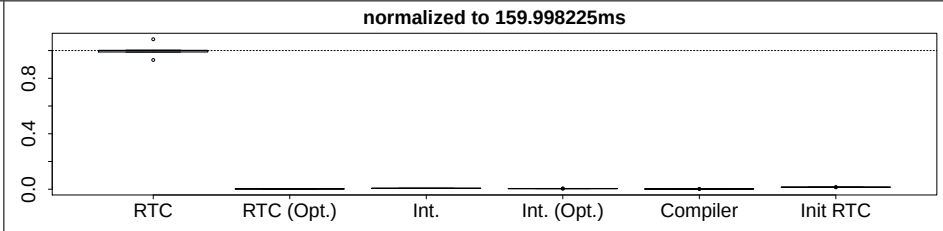
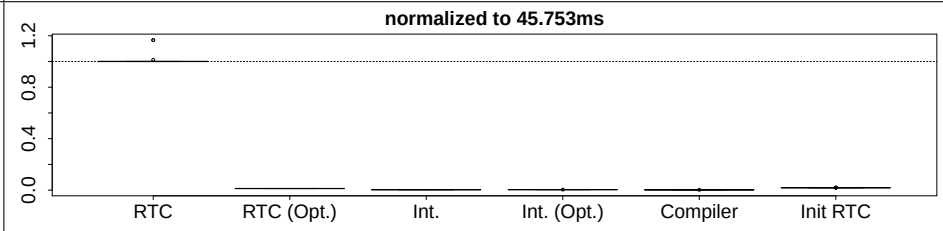
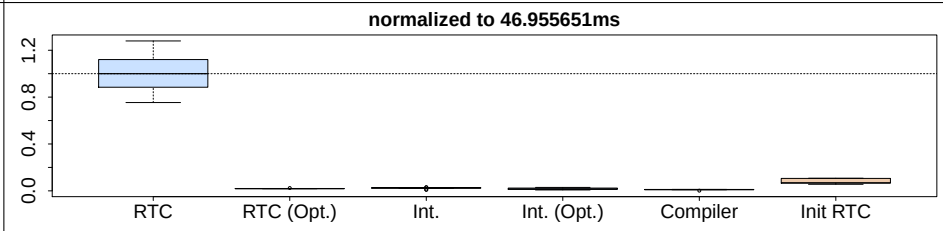
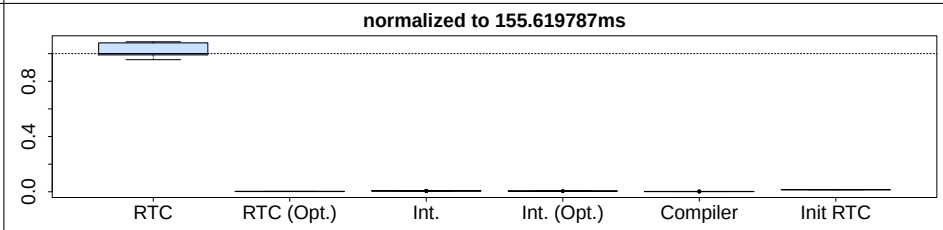
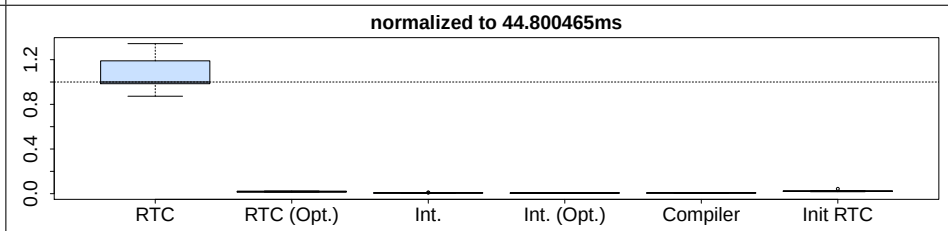
[37] Die Kästen stellen dabei die 50% der durchschnittlichsten Messungen dar und die Linie innerhalb der Box den Median. Für weitere Informationen über den Boxplot siehe auch: `http://de.wikipedia.org/wiki/Boxplot`

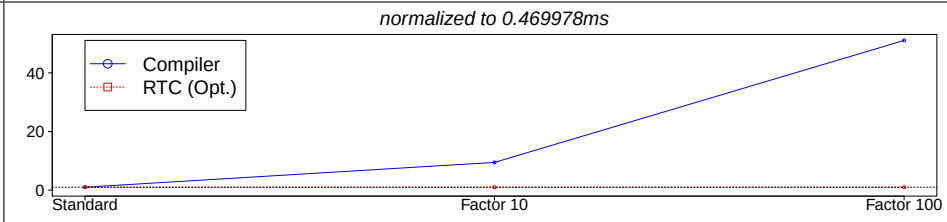
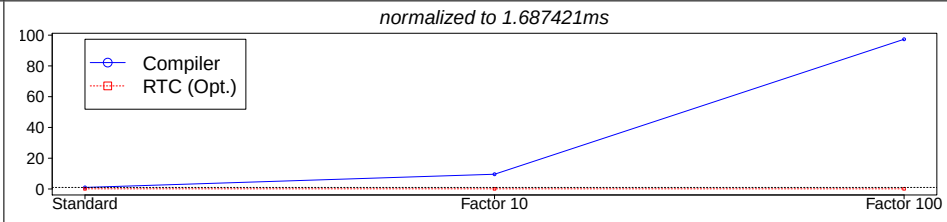
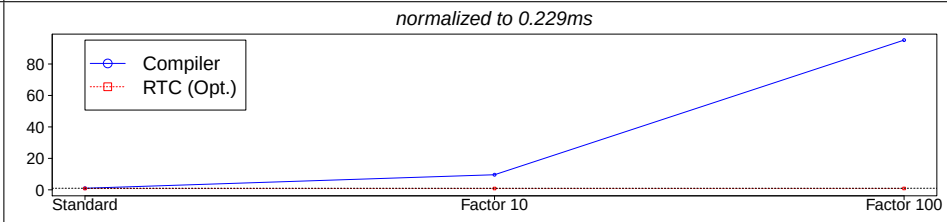
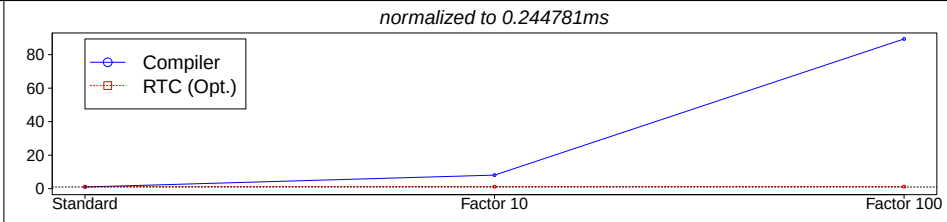
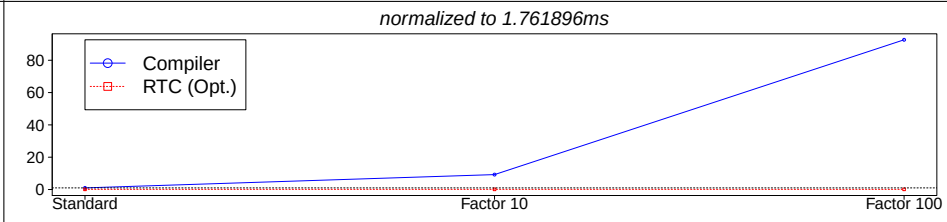
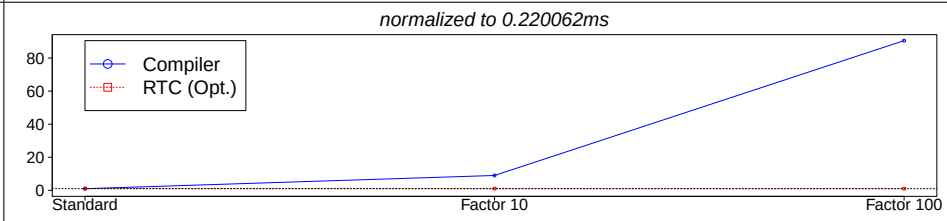
	threegrep, 1. Gruppe	**bonus**, 1. Gruppe
1.6 Server		
1.6 Jamvm		
1.6 Cacao		
1.7 Server		
1.7 Jamvm		
1.8 Server		

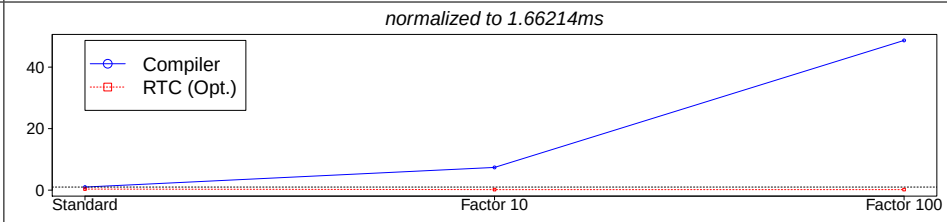
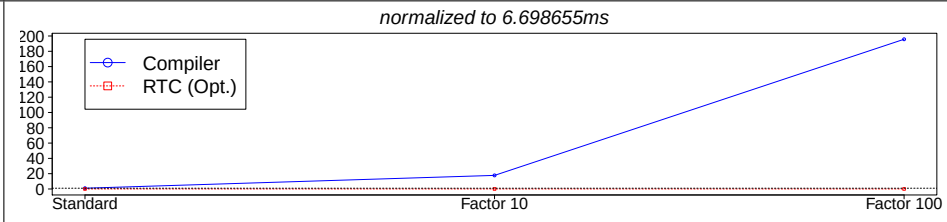
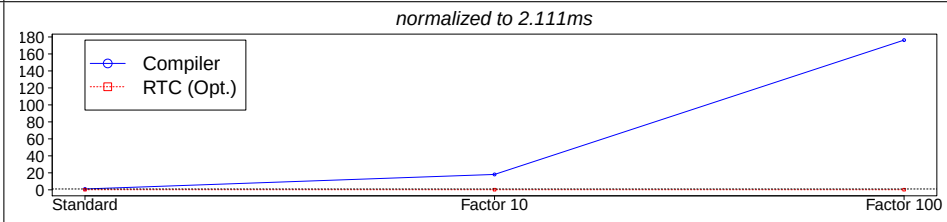
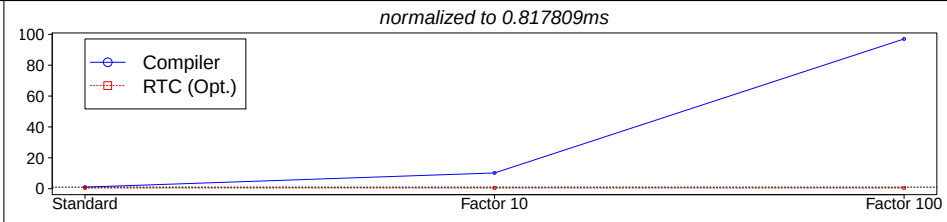
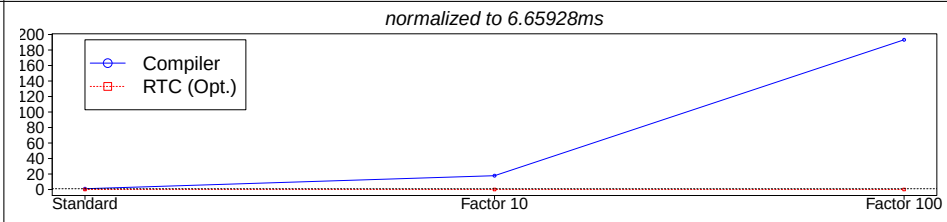
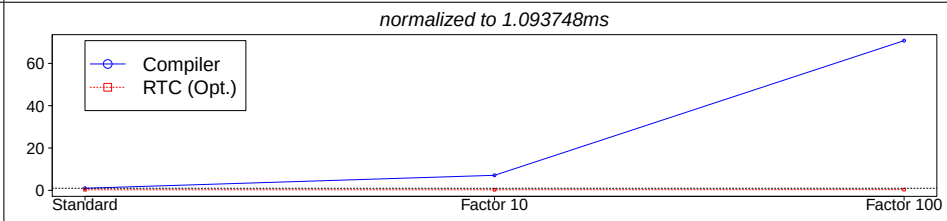
	webgraph, 1. Gruppe	**idf**, 1. Gruppe
1.6 Server		
1.6 Jamvm		
1.6 Cacao		
1.7 Server		
1.7 Jamvm		
1.8 Server		

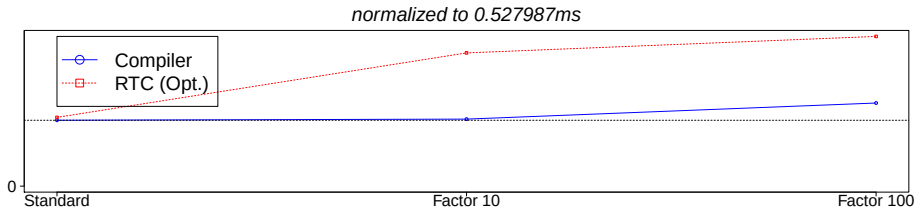
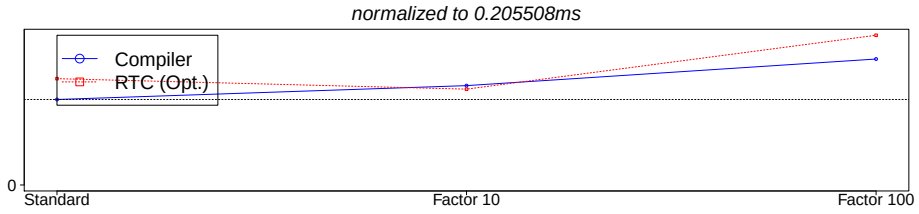
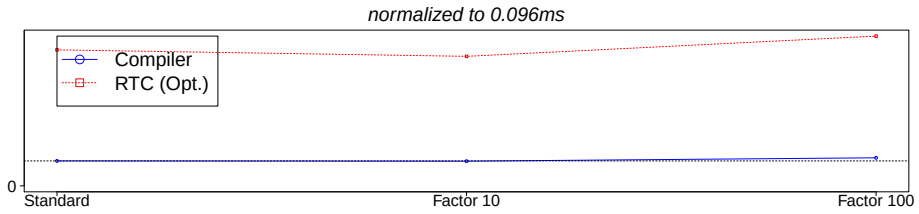
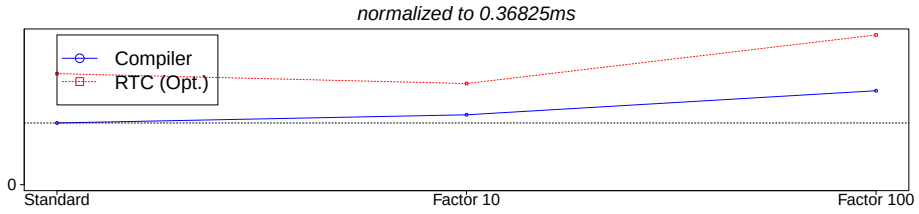
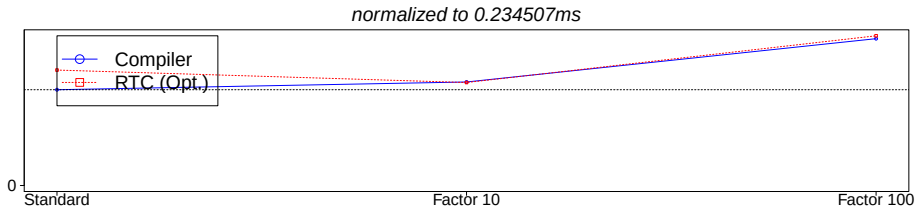
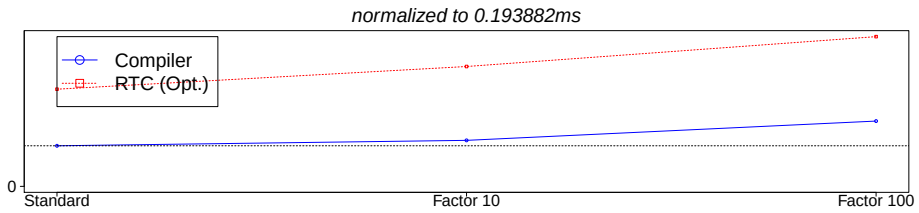
	arraynested, 2. Gruppe
1.6 Server	
1.6 Jamvm	
1.6 Cacao	
1.7 Server	
1.7 Jamvm	
1.8 Server	

	setnested, 2. Gruppe
1.6 Server	
1.6 Jamvm	
1.6 Cacao	
1.7 Server	
1.7 Jamvm	
1.8 Server	

	mapnested, 2. Gruppe
1.6 Server	
1.6 Jamvm	
1.6 Cacao	
1.7 Server	
1.7 Jamvm	
1.8 Server	

	arraynested, 3. Gruppe
1.6 Server	
1.6 Jamvm	
1.6 Cacao	
1.7 Server	
1.7 Jamvm	
1.8 Server	

	setnested, 3. Gruppe
1.6 Server	
1.6 Jamvm	
1.6 Cacao	
1.7 Server	
1.7 Jamvm	
1.8 Server	

	mapnested, 3. Gruppe
1.6 Server	
1.6 Jamvm	
1.6 Cacao	
1.7 Server	
1.7 Jamvm	
1.8 Server	

Da die Daten einiger Graphen relativ nahe beieinander liegen, folgen hier noch einmal die wichtigsten Verhältnisse der ersten und zweiten Gruppe als Zahlen aufgeschrieben (dabei werden nur jeweils die beiden Mediane verglichen, außerdem betrachten wir nur die Platform JDK 1.7, Server). In der ersten Gruppe vergleichen wir dabei den Laufzeitübersetzer mit dem Compiler:

* *threegrep.* Der Compiler ist ca. 3,49-mal schneller im Vergleich zum Laufzeitübersetzer.
* *bonus.* Der Compiler ist ca. 1,26-mal schneller im Vergleich zum Laufzeitübersetzer.
* *webgraph.* Der Compiler ist ca. 1,06-mal schneller im Vergleich zum Laufzeitübersetzer.
* *idf.* Der Compiler ist ca. 2,33-mal schneller im Vergleich zum Laufzeitübersetzer.

In der zweiten Gruppe vergleichen wir zuerst RTC mit RTC (Opt.):

* *arraynested.* RTC (Opt.) ist ca. 59,02-mal schneller im Vergleich zu RTC.
* *setnested.* RTC (Opt.) ist ca. 198,37-mal schneller im Vergleich zu RTC.
* *mapnested.* RTC (Opt.) ist ca. 52,03-mal schneller im Vergleich zu RTC.

Dann vergleichen wir noch RTC (Opt.) mit dem Compiler:

* *arraynested.* RTC (Opt.) ist ca. 71,54-mal schneller im Vergleich zum Compiler.
* *setnested.* RTC (Opt.) ist ca. 229,21-mal schneller im Vergleich zum Compiler.
* *mapnested.* RTC (Opt.) ist ca. 0,57-mal schneller (also ca. 1,75-mal langsamer) im Vergleich zum Compiler.

6.4 Interpretation

Ganz allgemein fallen zuerst einmal die ganz unterschiedlichen Zeiten innerhalb der verschiedenen Platformen auf. Grundsätzlich wurden die besten Laufzeiten dabei auf den virtuellen Maschinen *Server* der unterschiedlichen JDKs erzielt. *Cacao* schnitt meist ein Stück schlechter ab, *JamVM* praktisch immer am schlechtesten. Dies ist wohl mit den unterschiedlichen Optimierungs-Graden der einzelnen VMs zu erklären.

Das Ziel der PQL-Implementierung war es, eine zu Java-Code vergleichbare Laufzeit zu erreichen. Schauen wir uns die Messergebnisse an, so ist dies größtenteils auch gelungen. Zwar sind fast alle Messungen im Vergleich ein Stück langsamer, aber im Großen und Ganzen sind die Unterschiede nicht so gravierend. Im Vergleich zum manuellen Java-Code war der Benchmark *idf* sogar als einzige Ausnahme auf einer Platform effizienter (*JDK 1.6, Cacao*). Vergleichen wir mit dem Compiler, so ist der Laufzeitübersetzer zwar auch im Schnitt ein kleines bisschen langsamer, aber in ein paar Messungen (4 Stück) auch performanter.

Damit kommen wir zu den Messergebnissen der zweiten Gruppe. Sowohl das *Array-*, als auch das *Set-Benchmark* zeigen dabei sehr gut das, was wir erreichen wollten: Zuerst einmal führt die dynamische Optimierung beim Laufzeitübersetzer zu sehr viel schnelleren Ergebnissen im Vergleich zu jenen Messungen des Laufzeitübersetzers mit rein statischer Optimierung. Zum anderen kann man beobachten, dass der Compiler immer ungefähr auf der Stufe des Laufzeitübersetzers nur mit statische Optimierung ist und der Laufzeitübersetzer mit dynamischer Optimierung immer performanter als der Compiler ist. Der Interpreter mit dynamischer Optimierung ist überraschend performant (im Vergleich zu Compiler und Laufzeitübersetzer), dies mag aber auch an den konkreten Benchmarks liegen.

Bei dem *Map-Benchmark* beobachten wir nun eine sehr hohe Laufzeit beim Laufzeitübersetzer ohne Optimierungen. Dies kann aber ganz klar mit einem implemtierungstechnischen Detail erklärt werden: Bei der *Lookup-Methode* des Map-Objects, welches jeweils die innere Schleife initiiert, macht der Laufzeitübersetzer einiges an Zusatzaufwand, obwohl es sich jedes Mal um das gleiche Map-Object handelt, unabhängig vom ersten *Lookup*. Hier ist für die Zukunft also noch Optimierungsbedarf vorhanden. Auch im Vergleich zum Compiler schneidet dann aufgrund dieses Mangels der Laufzeitübersetzer schlecht ab, der Vergleich zwischen Laufzeitübersetzer mit dynamischer und jenem mit statischer Optimierung ist aber ähnlich zu denen der beiden Bench-

marks zuvor (*arraynested* und *setnested*).

Bevor wir zur dritten Gruppe kommen, betrachten wir nun erst noch die Erstellungszeiten des Laufzeitübersetzers. Dabei sollte man beachten, dass die Zeiten relativ zur Codegröße zustande kommen und praktisch nicht an die endgültige Laufzeit gekoppelt sind. Schaut man sich die Werte an, erkennt man, dass die Erstellungszeit nur dann nennenswert ins Gewicht fällt, wenn die Gesamtlaufzeit des Programms sehr kurz ist (beispielsweise bei *threegrep*, das mit einer Gesamtlaufzeit von teilweise unter zehn Millisekunde selbst kaum Zeit in Anspruch nimmt). Bei den zwei aufwendigsten Benchmarks (*webgraph* und *idf*) ist die Erstellungszeit meist mehr als 1000-mal schneller im Vergleich zur Ausführungszeit.

Die Messungen der dritten Gruppe belegen nun auch größtenteils genau das, was demonstriert werden sollte: Hat man vergleichsweise kleine Datenstrukturen, fallen die dynamischen Optimierungen kaum ins Gewicht, bei ungünstigen Größenverhältnissen aber kann es passieren, dass die Compiler-Laufzeiten stark steigen (teilweise bis fast ans 200-fache), währenddessen der Laufzeitübersetzer dank günstiger dynamischer Optimierung die Laufzeiten praktisch auf einer Stufe halten kann. Diese Beobachtungen können nun im *Map-Benchmark* nicht angestellt werden, dies könnte am gleichen Problem liegen, welches auch schon in der zweiten Gruppe der Messungen bei *mapnested* auftrat.

6.5 Aufruf der Benchmarks in der Kommandozeile

Möchte man die Benchmarks selbst auf einem Rechner ausführen, so kann man dies über die Kommandozeilen-Parameter problemlos machen. Der Aufruf ist dabei folgender:

```
1 benchmarks.Bench run idf [Liste der Benchmarks] [
      Ausführungsart] [optionale zusätzliche Argumente]
```

Die auszuführende main-Methode befindet sich also im Paket *benchmark* in der Klasse *Bench*. Die Liste der Benchmarks wird mit einem Komma getrennt und kann folgende Inhalte haben: *threegrep, bonus, webgraph, idf, arraynested, setnested* und *mapnested*. Die Ausführungsart kann dabei folgende Werte annehmen: *manual*, *runtime_pql* und *pql-1*. Das Argument *manual* brauchen Sie dabei nur für die Benchmarks der ersten Gruppe um den manuellen Java-Code auszuführen. Die Ausführungsart *runtime_pql* steht für den Lauf-

zeitübersetzer und mit *pql-1* rufen Sie den Compiler auf (die 1 bezeichnet die nicht-parallele Ausführung). Die optionalen zusätzlichen Argumente sind hingegen nur für die zweite und dritte Gruppe interessant: *-use-select-path* stellt die effizientere Datenstruktur-Aufteilung mittels dynamischer Optimierung ein und *-use-standard-interpreter* sorgt dafür, dass der normale Interpreter statt dem Laufzeitübersetzer genutzt wird. Dann gibt es noch die Optionen *factor10* und *factor100*, diese modifizieren die Größe der verwendeten Datenstrukturen. Dies wurde für die dritte Gruppe der Messungen verwendet, allerdings handelt es sich dabei nicht jeweils um den Faktor 10 und 100, sondern jeweils um den Faktor $\frac{1}{10}$ und $\frac{1}{100}$.

Zu beachten ist auch noch, dass die meisten Benchmarks mehr Ram-Speicher brauchen, als Java standardmäßig bereitstellt. Das zusätzliche Argument *-Xmx1024m* sollte dabei in den meisten Fällen für genug verfügbaren Speicherplatz sorgen.

7 Fazit

Die grundlegende Zielsetzung bestand darin, für PQL einen Laufzeitübersetzer zu schreiben, der auf der einen Seite von seiner Performanz sowohl mit dem Interpreter, als auch mit dem Compiler mithalten kann und auf der anderen Seite zur Laufzeit vorhandene Information für Performanzverbesserungen messbar nutzen kann. Die Messergebnisse zeigten: Die Laufzeit-Größenordnungen zwischen den PQL-Übersetzern Compiler und Laufzeitübersetzer war jedes Mal — zumindestens bei hinreichend umfangreichen Benchmarks — ausreichend ähnlich, auch wenn der Laufzeitübersetzer immer ein klein bisschen schlechter abschnitt. Die Verbesserung durch die Nutzung von Laufzeitinformationen konnte sehr gut beobachtet werden.

Zusammenfassend kann man festhalten, dass der Laufzeitübersetzer in der Funktion als Übersetzer-Prototyp seine Praktizität mehr als ausreichend demonstriert hat um somit zukünftig als Basis für weitere Optimierung dienen zu können.

8 Literaturverzeichnis

Literatur

[1] Reichenbach, C. (2011): PQIL revision 0.5.2 (`http://creichen.net/pql/IL`) [4.10.2013]

[2] Gyori, A. & Franklin L. & Lahoda J. & Dig D. (2013): Crossing the Gap from Imperative to Functional Programming through Refactoring (Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering Pages 543–553, `dig.cs.illinois.edu/papers/lambdaRefactoring.pdf`) [15.09.2013]

[3] Rehm, W. (1998): Ausarbeitung zum Proseminar IBM-PC, Kapitel 2.1.3 (`http://www.tu-chemnitz.de/informatik/RA/news/stack/kompendium/vortraege_98/grafik/index.html`) [15.09.2013]

[4] Leitenberger, B.: Die Cray 1 — Architektur eines Supercomputers (`http://www.bernd-leitenberger.de/cray-1.shtml`) [15.09.2013]

[5] Sheard, T. (2001, Springer Verlag): Accomplishments and Research Challenges in Meta-programming (`http://link.springer.com/chapter/10.1007%2F3-540-44806-3_2`) [23.09.2013]

[6] Jefferson Computer Museum (2004): What is the UCSD P-System? (`http://www.threedee.com/jcm/psystem/`) [15.09.2013]

[7] Preußer, T. (2011): Increasing the Performance and Predictability of the Code Execution on an Embedded Java Platform (`http://www.qucosa.de/fileadmin/data/qucosa/documents/7742/dissertation_thpreusser.pdf`) [19.09.2013]

[8] Lindholm, T & Yellin, F. (1999): The Java™ Virtual Machine Specification — Second Edition (`http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html`) [15.09.2013]

[9] Reichenbach, C. & Smaragdakis Y. & Immerman N. (2012): PQL: A Purely-Declarative Java Extension for Parallel Programming (Proceeding ECOOP'12 Proceedings of the 26th European conference on Object-Oriented Programming Pages 53–78)

[10] Bruneton, E. (2007, 2011): ASM 4.0 A Java bytecode engineering library (http://download.forge.objectweb.org/asm/asm4-guide.pdf) [15.09.2013]

[11] Duffy J. & Essey E. (2007): Parallel LINQ: Running queries on multi-core processors. (MSDN Magazine)

[12] Code Generation Library Developer Team (2002 – 2004): cglib description (http://cglib.sourceforge.net/) [15.09.2013]

[13] Mac Developer Library (2011): Grand Central Dispatch (GCD) Reference (https://developer.apple.com/library/mac/documentation/performance/reference/gcd_libdispatch_ref/Reference/reference.html) [2.10.2013]

[14] Ullenboom, C. (2011, Galileo Press): Java ist auch eine Insel, 9.Auflage (http://openbook.galileocomputing.de/javainsel9/index.htm#_top) [2.10.2013]

[15] National Instruments (2013): Multicore-Programmierung mit NI LabVIEW (http://www.ni.com/white-paper/14565/de/) [15.09.2013]

[16] Chappel, D. (2006): Introducing the .NET Framework 3.0 (http://msdn.microsoft.com/en-us/library/aa479861.aspx) [3.10.2013]

[17] Fischereder, W. (2003): The Common Intermediate Language (CIL) (http://www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2003/reports/Fischereder/Fischereder.pdf) [7.10.2013]

[18] Gamma, E. & Helm, R. & Johnson, R. E. & Vlissides J. (1994, Addison-Wesley Longman, 1. Auflage): Design Patterns. Elements of Reusable Object-Oriented Software.

9 Anhang

9.1 Quellcode

Der komplette Quellcode ist auf der beigefügten CD zu finden.

9.2 Beispiel: Von PQL zu Bytecode

Zur Veranschaulichung stelle ich hier den ganzen Übersetzungsweg eines PQL-Programms zu Bytecode vor. Dabei wird das bereits aus den Benchmarks bekannte Testprogramm *setnested* genommen. Das dazugehörige PQL-Programm lautet:

```
1 query(Set.contains(x)): generatedSet1.contains(x) && x <
      10 && generatedSet2.contains(x)
```

Dies wird dann in folgendes PQIL-Programm übersetzt, dabei steht *o0r* für das erste generierte Set, *o1r* für das zweite generierte Set und in *o2w* wird das Ergebnis-Set gespeichert. In *i0r* ist die Konstante 10 gespeichert. Die Kodierung der Variablen besteht aus drei Ziffern: Object-Typ (in unserem Fall o (*Object*) oder i (*Integer*)), Index und Attribut (r (lesen) oder w (schreiben)).

```
1 PQLFactory.Reduction(
2     new Reductor [] {PQLFactory.Reductors.SET(i1r, o2w)},
3     PQLFactory.ConjunctiveBlock(
4         PQLFactory.CONTAINS( o0r, i1w),
5         PQLFactory.LT_Int(i1r, i0r ),
6         PQLFactory.CONTAINS( o1r, i1r))
```

Anschließend übersetzt der Laufzeitübersetzer das ganze in folgenden Byte-Code (zur besseren Lesbarkeit sind hierbei noch Kommentare drinnen, diese sind im finalen Code nicht drin):

```
1  // class version 50.0 (50)
2  // access flags 0x1
3  public class Evaluator {
4
5    // compiled from: Evaluator.java
6
7    // access flags 0x1
8    public <init>()V
9     L0
10     LINENUMBER 1 L0
11     ALOAD 0
12     INVOKESPECIAL java/lang/Object.<init> ()V
13     RETURN
```

```
14      L1
15       LOCALVARIABLE this LEvaluator; L0 L1 0
16       MAXSTACK = 1
17       MAXLOCALS = 1
18
19     // access flags 0x9
20     public static evaluate(Ledu/umass/pql/Env;)Z
21       GOTO L0
22      L1
23       LDC "START_REDUCTION"
24       POP
25       NEW edu/umass/pql/container/PSet
26       DUP
27       INVOKESPECIAL edu/umass/pql/container/PSet.<init> ()V
28       ASTORE 3
29       LDC "START_CONJUCTIVE"
30       POP
31       LDC "START CONTAINS"
32       POP
33       ACONST_NULL
34       ASTORE 4
35       ICONST_0
36       ISTORE 5
37       ICONST_0
38       ISTORE 6
39       ALOAD 7
40       DUP
41       INSTANCEOF edu/umass/pql/container/PSet
42       IFNE L2
43       CHECKCAST java/util/Set
44       INVOKEINTERFACE java/util/Set.toArray ()[Ljava/lang/Object;
45       DUP
46       ASTORE 4
47       GOTO L3
48      L2
49       CHECKCAST edu/umass/pql/container/PSet
50       INVOKEVIRTUAL edu/umass/pql/container/PSet.getRepresentation ()[Ljava/lang/Object
             ;
51       DUP
52       ASTORE 4
53      L3
54       ARRAYLENGTH
55       ISTORE 5
56      L4
57       ILOAD 5
58       ILOAD 6
59       IF_ICMPLE L5
60       ALOAD 4
61       ILOAD 6
62       AALOAD
63       DUP
64       IFNULL L6
65       POP
66       ALOAD 4
67       ILOAD 6
68       AALOAD
69       DUP
70       INSTANCEOF java/lang/Number
71       IFEQ L7
72       CHECKCAST java/lang/Number
73       INVOKEVIRTUAL java/lang/Number.intValue ()I
74       GOTO L8
```

```
75      L7
76       DUP
77       INSTANCEOF java/lang/Boolean
78       IFEQ L9
79       CHECKCAST java/lang/Boolean
80       INVOKEVIRTUAL java/lang/Boolean.booleanValue ()Z
81       GOTO L8
82      L9
83       DUP
84       INSTANCEOF java/lang/Character
85       IFEQ L10
86       CHECKCAST java/lang/Character
87       INVOKEVIRTUAL java/lang/Character.charValue ()C
88       GOTO L8
89      L10
90       GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
91       LDC "the specified object cannot be converted in a numerical value."
92       INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
93       NEW java/lang/Exception
94       DUP
95       LDC "the specified object cannot be converted in a numerical value."
96       INVOKESPECIAL java/lang/Exception.<init> (Ljava/lang/String;)V
97       ATHROW
98      L8
99       ISTORE 8
100      GOTO L11
101      LDC "Reductor_iteration"
102      POP
103      ALOAD 3
104      ILOAD 8
105      INVOKESTATIC edu/umass/pql/Env.canonicalInteger (I)Ljava/lang/Integer;
106      ALOAD 3
107      INSTANCEOF edu/umass/pql/container/PSet
108      IFNE L12
109      INVOKEVIRTUAL java/util/Set.add (Ljava/lang/Object;)Z
110      GOTO L13
111     L12
112      INVOKEVIRTUAL edu/umass/pql/container/PSet.add (Ljava/lang/Object;)Z
113     L13
114      POP
115      GOTO L14
116      GOTO L11
117     L6
118      POP
119     L15
120      IINC 6 1
121      GOTO L4
122     L5
123      GOTO L14
124     L11
125      ILOAD 8
126      LDC 10
127      IF_ICMPLT L16
128      GOTO L15
129      GOTO L17
130     L16
131     L17
132      LDC "START CONTAINS"
133      POP
134      ACONST_NULL
135      ASTORE 9
136      ICONST_0
```

```
137      ISTORE 10
138      ICONST_0
139      ISTORE 11
140      ALOAD 12
141      DUP
142      INSTANCEOF edu/umass/pql/container/PSet
143      IFNE L18
144      CHECKCAST java/util/Set
145      INVOKEINTERFACE java/util/Set.toArray ()[Ljava/lang/Object;
146      DUP
147      ASTORE 9
148      GOTO L19
149     L18
150      CHECKCAST edu/umass/pql/container/PSet
151      INVOKEVIRTUAL edu/umass/pql/container/PSet.getRepresentation ()[Ljava/lang/Object
             ;
152      DUP
153      ASTORE 9
154     L19
155      ARRAYLENGTH
156      ISTORE 10
157     L20
158      ILOAD 10
159      ILOAD 11
160      IF_ICMPLE L21
161      ALOAD 9
162      ILOAD 11
163      AALOAD
164      DUP
165      IFNULL L22
166      DUP
167      INSTANCEOF java/lang/Number
168      IFEQ L23
169      CHECKCAST java/lang/Number
170      INVOKEVIRTUAL java/lang/Number.intValue ()I
171      GOTO L24
172     L23
173      DUP
174      INSTANCEOF java/lang/Boolean
175      IFEQ L25
176      CHECKCAST java/lang/Boolean
177      INVOKEVIRTUAL java/lang/Boolean.booleanValue ()Z
178      GOTO L24
179     L25
180      DUP
181      INSTANCEOF java/lang/Character
182      IFEQ L26
183      CHECKCAST java/lang/Character
184      INVOKEVIRTUAL java/lang/Character.charValue ()C
185      GOTO L24
186     L26
187      GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
188      LDC "the specified object cannot be converted in a numerical value."
189      INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
190      NEW java/lang/Exception
191      DUP
192      LDC "the specified object cannot be converted in a numerical value."
193      INVOKESPECIAL java/lang/Exception.<init> (Ljava/lang/String;)V
194      ATHROW
195     L24
196      ILOAD 8
197      IF_ICMPEQ L27
```

```
198      GOTO L28
199      GOTO L29
200     L27
201     L29
202      GOTO L30
203     L22
204      POP
205     L28
206      IINC 11 1
207      GOTO L20
208     L21
209      GOTO L15
210     L30
211      LDC "Reductor_iteration"
212      POP
213      ALOAD 3
214      ILOAD 8
215      INVOKESTATIC edu/umass/pql/Env.canonicalInteger (I)Ljava/lang/Integer;
216      ALOAD 3
217      INSTANCEOF edu/umass/pql/container/PSet
218      IFNE L31
219      INVOKEVIRTUAL java/util/Set.add (Ljava/lang/Object;)Z
220      GOTO L32
221     L31
222      INVOKEVIRTUAL edu/umass/pql/container/PSet.add (Ljava/lang/Object;)Z
223     L32
224      POP
225      GOTO L15
226     L14
227      GOTO L33
228     L0
229      ALOAD 0
230      GETFIELD edu/umass/pql/Env.v_object : [Ljava/lang/Object;
231      ICONST_2
232      AALOAD
233      ASTORE 3
234      ALOAD 0
235      GETFIELD edu/umass/pql/Env.v_object : [Ljava/lang/Object;
236      ICONST_0
237      AALOAD
238      ASTORE 7
239      ALOAD 0
240      GETFIELD edu/umass/pql/Env.v_int : [I
241      ICONST_1
242      IALOAD
243      ISTORE 8
244      ALOAD 0
245      GETFIELD edu/umass/pql/Env.v_object : [Ljava/lang/Object;
246      ICONST_1
247      AALOAD
248      ASTORE 12
249      GOTO L1
250     L33
251      ALOAD 0
252      GETFIELD edu/umass/pql/Env.v_object : [Ljava/lang/Object;
253      ICONST_2
254      ALOAD 3
255      AASTORE
256      ALOAD 0
257      GETFIELD edu/umass/pql/Env.v_object : [Ljava/lang/Object;
258      ICONST_0
259      ALOAD 7
```

```
260     AASTORE
261     ALOAD 0
262     GETFIELD edu/umass/pql/Env.v_int : [I
263     ICONST_1
264     ILOAD 8
265     IASTORE
266     ALOAD 0
267     GETFIELD edu/umass/pql/Env.v_object : [Ljava/lang/Object;
268     ICONST_1
269     ALOAD 12
270     AASTORE
271     ICONST_1
272     IRETURN
273     MAXSTACK = 0
274     MAXLOCALS = 0
275 }
```