How to optimize Rscript comprehensions?

Menno Bredenoord

1 September 2006

One Year Master Program Software Engineering

Thesis Supervisor: Prof. Dr. Paul Klint
Internship Supervisor: Prof. Dr. Paul Klint
Company or Institute: Centrum voor Wiskunde en Informatica (CWI)

Universiteit van Amsterdam,
Hogeschool van Amsterdam,
Vrije Universiteit

Universiteit van Amsterdam

Centrum voor Wiskunde en Informatica

Table of Contents

Summary

Rscript is a small scripting language based on relational calculus. Its main purpose is to analyze and query source code from software. In this thesis we will give an answer to the following two research questions:

I. How can Rscript comprehensions be optimized?

II. What are the effects of the optimizations?

We will investigate two methods which can be used to optimize Rscript comprehensions. Comprehensions are constructions used for iterating over one or more sets or relations, while matching one or more of their elements to a boolean-valued expression.

The first method considers several algebraic optimization techniques we found in the literature on comprehensions. These are: Qualifier Interchange, Filter Hiding, Product Elimination, Common Subexpression Elimination and Index Introduction. Next we discuss optimization techniques which are used for optimizing relational databases, and can be used to optimize Rscript. These are: Commuting Selections – Cartesian Product, Commuting Selections – set-difference, Commuting Selections – union, Semantic Query Caching – subsumption, Semantic Query Caching – overlap and Peephole Optimization. For each of the techniques mentioned above it is discussed which transformations they perform, how the technique realizes a performance increase, which problems occur when applying them to Rscript and some examples of use in Rscript.
In order to determine the most efficient algebraic optimization technique, we evaluated the techniques by 1) performing measurements on them by using test cases, 2) checking the probability that the optimization can be applied to a Rscript file and 3) interpreting what the literature claims about them. Finally we concluded that the optimization techniques Qualifier Interchange and Filter Hiding are the most efficient.

The second method is based on the optimization of the set operators by using other data structures for set representation. The current implementation of Rscript uses a Linear data structure to perform set operations. In order to optimize this, we investigate several data structures, such as: Hash Table, Binary Search Tree, Red-Black Tree, Binomial Tree and Judy Arrays. For each of the data structures we obtained their theoretical performance. For the two theoretically best performing data structures, Hash Table and Red-Black Tree, we performed several measurements on their operators.
The results of these measurements show us that the Red-Black Tree is the best performing data structure.

Finally we give an advice on which method to use best for optimizing Rscript comprehensions. We advice *not* to focus on algebraic optimization techniques to optimize the Rscript comprehensions, because the applicability of these optimizations is very low and an answer about the performance increase is hard to give. We *do* advice to change the current data structure and thereby improving the performance of the entire comprehension.

Preface

This thesis is written as a master project of the one year Master program Software Engineering at the University of Amsterdam. The project is executed at the Centrum voor Wiskunde en Informatica (CWI).

I worked with a lot of enthusiasm on this project, and would like to thank all of my colleagues at the CWI for keeping me enthusiastic and helping me whenever they could. A special word of thanks goes to my supervisor Paul Klint for suggesting this interesting topic and helping me to bring this project to a good end. I would also like to say thanks to my fellow students with whom I shared a room during the project: it has been fun.
Finally I would like to thank my family and Céline for their support.

In the first chapter we give a description of the problem that this thesis tries to solve and we introduce our research questions. The second chapter gives some background information about the problem. In the third chapter we discuss our research plan, which is divided into four iterations. In the fourth and fifth chapter we discuss several algebraic optimization techniques. In the sixth chapter we evaluate these algebraic optimization techniques and perform measurements on them, also several possible implementation methods are discussed and one of them is chosen to implement a prototype. The seventh chapter discusses several data structures and performs several measurements on them. The eighth chapter we present the results of the project and give an answer to the research questions. In chapter nine, the final chapter, we evaluate the entire project.

1. Problem Description

Rscript is a small scripting language based on relational calculus. Its main purpose is to analyze and query source code from software. Relevant information from the source code, in the form of relations, has to be extracted first. This can be done with, for example, ASF+SDF. Subsequently Rscript can use these relations to derive additional information from the software like, for example, the procedures which call each other directly, but also indirectly (with Transitive closure[1]). When discussing Rscript, henceforth will be referred to Rscript version 0.3.

1.1. The problem

Rscript is mainly used at the CWI, VU and the HvA. Recently it became apparent that the performance of Rscript has become a problem. When Rscript is performing thorough analysis on software applications, the performance of the relational calculations becomes a problem, especially on larger applications. The time needed for these analyses to complete, has to be in proportion with the extra amount of information that is retrieved. However this is not always the case and therefore this is a problem for the application of Rscript. When Rscript will become more efficient its application can be expanded. And this is what we want to achieve.

The performance issue described above is not solely caused by the computational power of Rscript, but also by the syntax used to formulate the analyses. When complex situations have to be calculated, their performance can only be maintained when the use of expensive operators like, for example, the Cartesian product[2], is limited. Since Rscript is not a very large language, and its syntax is fairly limited, the entire analysis will profit from the optimization of even one of the operators.

It turns out that, performance wise, most of the execution time is spent in calculating comprehensions. Therefore the emphasis of this thesis will be on optimizing comprehensions. In order to optimize them, research will be conducted in the relevant literature about which techniques are available for optimizing comprehensions. A prototype will be created to test the relevant optimizations.

The results of this research can be used to determine which optimizations are available and how to further optimize Rscript. The research is also interesting for other companies and software developers who work with a language which uses comprehensions.

This results in the following two research questions for this thesis:

I. **How can Rscript comprehensions be optimized?**

II. **What are the effects of the optimizations?**

1.2. Sub questions

In order to be able to answer the research questions, several sub questions have to be answered first.
For the first research question, the following questions are raised:

I.1 *Has there already been conducted some research in the literature about the optimization of comprehensions in general? And if so, which techniques are used? In what way is an optimization performed? Is the optimization applicable to Rscript? How does the optimization work in Rscript?*

I.2 *Is it possible to apply the techniques used for the optimization of relational database queries to Rscript comprehensions? If this is not the case, why not?*

I.3 *Which data structures exist for set representation? In what way do they perform better than the current data structure of Rscript?*

For the second research question, the following questions are raised:

II.1 *To what extent do the techniques really perform an optimization of the comprehensions?*

II.2 *Which of the optimization techniques is considered the most efficient one?*

Finally, we raise the questions:

1 http://en.wikipedia.org/wiki/Transitive_closure - the tuple <a,c> if <a,b> and <b,c> exists within the relation
2 http://en.wikipedia.org/wiki/Cartesian_product - AxB is {<1,2>,<3,4>} if A is {1,2} and B is {3,4}

II.3 *How can the optimization process be automated?*

II.4 *How to determine the effect of using another data structure for set representation?*

1.3. Objectives

In order for this project to be successful, the following objectives have been raised:

- *A better understanding of how comprehensions can be transformed into a more efficient form.*
- *A better understanding of different data structures and their performance.*
- *Detailed knowledge of several algebraic optimization techniques.*
- *An overview of the efficiency of the algebraic optimization techniques.*
- *To produce a working prototype which (semi-)automatically applies optimization techniques to Rscript files.*
- *To produce a prototype which can compare the efficiency of different data structures.*

2. Background and Context

In this chapter several important issues are discussed. First we briefly consider the relevant literature which focuses on optimization techniques, data structures and the evaluation of optimizations. Afterwards the notation of the comprehensions of Rscript is discussed and some assumptions are made about the environment in which the optimization techniques will be evaluated.

2.1. Relevant literature

Here we briefly discuss the relevant literature that we have found.

2.1.1 Optimization techniques

In the literature the following three types of areas in which optimizations can be used to optimize Rscript are found.

1) Comprehension optimization.

In the literature on optimization techniques that are specifically focused on comprehensions, several algebraic transformations are identified which state that they perform an optimization. For example: Qualifier Interchange[1,2,5] enables the possibility to switch qualifiers within a comprehension. By placing less expensive qualifiers as foremost as possible in the comprehension, the more expensive qualifiers will need to be evaluated fewer times and hereby increasing the performance. This approach appears to be a promising technique, which will certainly need some more detailed investigation. Another algebraic optimization is called Common Subexpression Elimination[1,11] and states that expressions which are used multiple times, should be calculated once and subsequently the result should be reused. In larger applications the possibility of expressions being used multiple times, is more than present. Therefore this optimization technique is also worth some further investigation. This will be done in the first iteration (see §3.1.1).
Other algebraic optimizations found are Product Elimination[1], Filter Hiding[1,5], Evaluating Options[1,4] and Index Introduction[1,2]. Each of these techniques will be studied and documented in detail in this thesis.

2) Relational Database optimization

It is well known that comprehensions show a great similarity with the operation of queries in relational databases. The possibility exists that the techniques which are used for optimizing relational queries, can be adapted in order to optimize Rscript comprehensions. Several interesting relational database optimization techniques are found and will be discussed, such as Commuting Selections[6], Semantic Query Caching[8,10] and Peephole optimization[12]. This will also be done in the first iteration. The optimization power of these techniques is based on different methods. Some techniques are replacing expensive operators with less-expensive operators, others are creating indices in order to speed-up the retrieval process.

Whenever a certain optimization technique, whether algebraic or not, uses a specific function, method or operator in order to perform its optimization, the question is whether it is possible to express this function in Rscript. The possibility exists that Rscript will fall short on this part. Some of these optimization techniques will be discussed, although they cannot be applied to Rscript in its present state. However in the future, when Rscript will be expanded, there might be a possibility that these functions can be expressed.

3) Data structures

The current implementation of Rscript uses a Linear set implementation. Several data structures have been found in the literature which can be used to represent a set as used in Rscript and to optimize the current implementation. We have identified the following: Hash Tables[15], Binary Search Tree[15], Red-Black Tree[15] , Binomial Heap[15] and Judy Arrays[16,17,18]. Each of these techniques will be investigated and their performance will be discussed.

2.1.2 Evaluation of optimizations

The following information about measurements, performance gain and comparisons will be used to answer the second research question (II) and the first (*II.1*) and second (*II.2*) sub questions.

Every optimization technique that is suitable for Rscript, whether algebraic or based on techniques used in relational databases, has to be tested in order to determine to what extent the technique does perform an optimization. To determine the extent of the optimization, a measurement of the execution time will be conducted as done in [8,11]. This measurement will done on test scripts which will be specifically designed for each of the techniques found. This will be done in the second iteration (see §3.1.2).

2.2. Comprehension notation

Comprehensions are constructions for iterating over one or more sets or relations while matching one or more of their elements to a boolean-valued expression. The form and notation of the comprehensions are discussed here, in order to describe the transformations of the optimization techniques in the subsequent sections. Comprehensions take the following form:

$$\{ E_1, \ldots, E_m \mid Q_1, \ldots, Q_n \}$$

The return values of the comprehension are specified by $E_1, \ldots, E_m$. The qualifiers $Q_1, \ldots, Q_n$ within the comprehension can be divided in generators (G) and filters (F). Generators take the form:

```
p : e
```

Where p introduces one or more new variables of different types (e.g.: int, str) and e is a collection (e.g.: set, rel) which is enumerated and its results are assigned to the variables. If e is a set then p introduces a variable and when e is a relation, a tuple of variables is introduced.
Filters are boolean-valued expressions that specify the conditions which the variables of the generators must satisfy in order to be included in the result.

Comprehensions show a lot of similarities with (SQL) queries in Relational Databases. Example:

SQL: $\quad$ SELECT $v_1, \ldots, v_n$ FROM R WHERE P_1 AND ... P_n

and the corresponding comprehension:

Comprehension: $\quad \{ v_1, \ldots, v_n \mid v_1, \ldots, v_n : R, P_1, \ldots, P_n \}$

A more thorough introduction into comprehensions and Rscript in general can be found in [3].

2.3. Assumptions

When researching algebraic optimization techniques, several assumptions have to be made about the environment in which they will be evaluated. This is necessary in order for some techniques to be able to work.

The comprehensions will be evaluated using caching[1]. This states that when an expression is calculated for the first time, it is read from (slower) secondary storage and the expression has to be calculated. Afterwards, when the expression is needed again, the result of the expression can be directly read from (faster) primary storage and it does not have to be calculated again.

For some optimization techniques, such as Qualifier Interchange, the principle of bag equality[1,5] has to apply. Bag equality states that the order of tuples in a bag is not significant. So two collections are bag equal if they contain the same elements, although possibly in a different order. Rscript only has the collections *set* and *relation*. Since the definition of a *set* already states that the order of the elements is non-significant and in Rscript a *relation* uses the same implementation as a *set*[3,7], this principle already applies. Therefore it will be no problem for the principle of bag equality to apply in Rscript.

3. Research Plan

In this chapter, the plan for answering the research questions is discussed.

3.1. Iterations

In order to answer the research questions, the project is separated into four iterations. In each of the iterations an attempt will be made to answer one or more of the raised sub questions.

3.1.1. Iteration 1

As briefly mentioned in chapter 2, the first iteration of this project consists of a research in the literature of finding optimizations techniques. Not only will we discuss the optimization techniques especially designed for comprehensions, but also the techniques that are used for optimizing relational databases. An attempt will be made to answer sub questions *I.1* and *I.2*.

Several optimization techniques will be studied and for each of these technique the following important issues will be addressed:

- *Transformation rules.*
  Each of the algebraic optimization techniques consist of one or more transformations from the original form to a more efficient form. These transformations and their rules are discussed in this issue.
- *(possible) Performance gain.*
  We will discuss whether the use of the optimization technique increases the performance and what the improvement will be.
- *Possible problems when applying the technique to Rscript.*
  We will discuss whether the optimization technique is suitable for applying on Rscript and what the eventual problems are when doing so.
- *Examples of use.*
  Some examples will be given of how the optimization technique is expressed in Rscript.

3.1.2. Iteration 2

The second iteration tries to answer the sub questions *II.1* and *II.2*.

The performance increase of the optimizations is determined by applying each of the optimizations separately to a test file. In order to make sure that every optimization technique is fully tested, a unique test case is created for each optimization. Afterwards an overview will be constructed of the performance of each separate optimization and when possible, of several optimizations used together. Eventually a comparison will be made between the performance of the original execution and the optimized execution.

In order to determine which optimization technique is considered the most efficient, several factors will be taken into account, like 1) the extent of the optimization improvement, 2) the possibility that an optimization can be applied to a Rscript file and 3) what the literature tells us about this optimization. Finally a ranking of optimizations will be given.

Eventually we conclude which techniques will be best suited for implementation in the prototype.

3.1.3. Iteration 3

The third iteration is meant for developing a small prototype application which enables us to *automatically* apply optimization techniques to Rscript files. The result can be used to validate the result obtained by *manually* applying the techniques to Rscript files. This answers sub question *II.3*.

First of all, a small study will be conducted in order to determine the best way to construct a prototype. There are several possible methods of implementing the automation of optimizing Rscript comprehensions. Three methods will be introduced here:

- *Standalone application.*
  This application is a standalone Java program that, given a Rscript file, performs the optimizations and returns an

optimized Rscript file.

- *Extending Rscript in ASF+SDF.*
  The current Rscript implementation in ASF+SDF will be extended in order to automatically apply the optimization techniques.
- *Complete re-creation of Rscript.*
  The last method is based on creating an entire new version of Rscript in Java, which automatically applies the optimization techniques.

From the results of this research a conclusion will be drawn about which method to choose. Afterwards, the prototype will be constructed according to the chosen method. Before the optimization transformations are implemented, a pseudocode algorithm of each optimization will be created first. Finally, when the application is functional, it will be applied to the same test cases as used in the second iteration. A comparison can then be made between the optimization of the manual results and the automated results.

3.1.4. Iteration 4

The fourth and last iteration will be used to investigate different data structures and to create another prototype which will enable us to compare the efficiency of different data structures. This answers sub question *I.3* and *II.4.*

First the operations of Rscript which are most frequently used on the current data structure will be determined. Subsequently several data structures which we found in the literature will be discussed and their performance of the operations will be determined.
After the discussion we will determine the best candidate data structure for Rscript, which will be used in the prototype to be developed. This prototype will enable us to compare the running time of different data structures. We shall import an implementation of the best candidate and create a default Java version for comparison.

In order to determine the efficiency of the candidate data structure, we shall use an existing Rscript file and extract its operations on the data structure. These operations will be used as a test case in the prototype and the results of the comparison will be discussed.

4. Comprehension optimization techniques

In this chapter several optimization techniques are discussed. These techniques are specifically used for the optimization of comprehensions (in general). This chapter answers sub question *I.1*.

For each technique several important points are discussed: transformation rules, performance gain and suitability for Rscript.

Several techniques which are found in the literature (as mentioned in the background chapter), like: memoising[2] and Evaluating Options[1,4] are not discussed here. This is because their power for optimization is based only on constructions that are not possible in Rscript. The following techniques are discussed here: Qualifier Interchange, Filter Hiding, Product Elimination, Common Subexpression Elimination and Index Introduction.

4.1. Qualifier Interchange

Qualifier Interchange [1,2,5] is an algebraic optimization technique based on the possibility to swap any two qualifiers inside a comprehension. It is also referred as Selection Promotion[6].
When Qualifier Interchange is applied to a comprehension containing two or more generators which each iterate over different relations, the order of the generators within the comprehension will also be influenced by the size of these relations. The relations should be ordered in ascending order of their size. This is because when the smaller relation is in front, the larger relation can be processed as a whole.

Transformation rules

Here the transformations of Qualifier Interchange are explained.

Condition: f_2 does not use variables which are declared in f_1.
qi/1 { v | q_0 , q_1 , f_1 , f_2 } == { v | q_0 , q_1, f_2 , f_1 }

The above transformation states that the position of two filters (f_1 and f_2) within the comprehension can be swapped. This transformation is only possible if the stated conditions are met. Here the only condition is that the second filter (f_2) does not use variables which are declared in the first filter (f_1), because otherwise the second filter will use non existing variables and the comprehension will be incorrect. This transformation has to be used in several situations:

- The second filter uses less variables than the first filter does. This filter is less-expensive to compute, and therefore it will decrease the intermediate result.
- If the second filter is expected to decrease the size of the intermediate result more than that the first filter can.

Example:
```
C1    { a | <int a, int b> : AB , <int c, int d> : CD , b == c , d == 99 }
```
The above example shows a comprehension with two filters at the end. On this comprehension `qi/1` can be applied and will be transformed into `C2`:
```
C2    { a | <int a, int b> : AB , <int c, int d> : CD , d == 99 , b == c }
```
Above we show that the filters `b == c` and `d == 99` are swapped. Now the comprehension will be more efficient because the filter `d == 99` will decrease the intermediate result more than `b == c`.

Condition: (p_2:e_2) does not use variables declared by p_1.
qi/2 { v | q_0 , p_1 : e_1 , p_2 : e_2 , q_1 } == { v | q_0 , p_2 : e_2 , p_1 : e_1 , q_1 }

The second transformation states that the position of two generators (p_1 : e_1 and p_2 : e_2) within the comprehension can be swapped. This is only possible if the second generator (p_2 : e_2) does not uses variables declared by the variables of the first generator (p_1), because otherwise the second generator will use non existing variables. This transformation is useful in combination with one of the other transformations of Qualifier Interchange, in order to decrease the size of the intermediate result as early as possible.

Example:
We continue the result of the previous example (`C2`) because it contains a generator followed by another generator. On this comprehension, `qi/2` can be applied and will be transformed into `C3`:
```
C3    { a | <int c, int d> : CD , <int a, int b> : AB , d == 99 , b == c }
```
Here we show that the generators `AB` and `CD` are swapped. On itself this has no performance gain, however the current form

is useful for other transformations to be able to be applicable.

Condition: f_l does not use variables declared by p_l.
qi/3 { v | q_0 , p_1 : e_1 , f_1 , q_1 } == { v | q_0 , f_1 , p_1 : e_1 , q_1 }

The last transformation states that a generator (p_1 : e_1) and a filter(f_1) can be swapped. This is only possible if the filter does not use variables which are declared in the variables of the generator (p_1), because otherwise the filter will use non existing variables. This transformation is useful to decrease the size of the intermediate result as early as possible.

Example:
Here we will continue with the previous example C3. Since there is a generator followed by a filter, this comprehension can be transformed using qi/3, resulting in:

```
C4    { a | <int c, int d> : CD , d == 99 , <int a, int b> : AB , b == c }
```

This shows that the generator AB and the filter d == 99 are swapped. This will make the comprehension more efficient because the generator CD and the filter d == 99 will decrease the intermediate result and thereby resulting in less enumerations of the generator AB.

Performance gain

The performance gain of Qualifier Interchange is based on switching places of qualifiers (filters/generators) within the comprehension in order to decrease the size of the intermediate result. By placing filters as foremost as possible, they restrict the relation by discarding elements which do not apply and therefore the succeeding generators and filters will be used less. The performance gain is different for each comprehension since not every switch of qualifiers ensures an equal performance increase.

Suitability
The technique Qualifier Interchange is perfectly suited for optimizing the comprehensions in Rscript. The transformations are easy to apply to comprehensions. However, before transforming a comprehension using one of the above transformations, information has to be gathered for each of the qualifiers about where their used variables are declared. This information is necessary to determine whether a transformation can be applied or not.

4.2. Filter Hiding

Filter hiding [1,5] brings a generator with its accompanying filters outside the original comprehension and places them inside a new expression. By doing so, it transforms the comprehension into a more suitable form. Most of the time the new form is clearer and more intuitive than the original. By placing a part of the comprehension outside the comprehension, there is an increased possibility that other optimization techniques can be applied, such as Common Subexpression Elimination.

Transformation rules

Here the transformations of Filter Hiding are explained.

Condition: f_e is only using variables declared in p or outside the comprehension
fh/1 { v | q_0 , p : e , f_e , q_1 } == { v | q_0 , p' : e' , q_1 } where e' = { p' | p : e , f_e }

The above transformation states that a generator (p : e) and a filter (f_e) can be brought outside the comprehension into a new expression(e'). The generator is adapted to iterate over this new expression. This transformation is possible if the filter is only using variables which are declared in the variables of the generator (p) or to variables which are declared outside the comprehension. The variables of the generator which are not used anymore in the rest of the original comprehension (q_1), can be removed from the return variables of the new expression (p'). This transformation has to be used whenever possible since the new form will enable other transformations to be applicable.

Example:

```
C1    { s1 | < int s1, int grade > : grades , grade < 5.5 , < int s2, int times > :
      absence , s1 == s2 , times < 2 }
```

The above example shows a comprehension that contains a generator (grades) and a filter (grade) which is only using

variables declared in this generator. On this comprehension `fh/1` can be applied and will be transformed into `C2`:

```
C2     { s1 | < int s1, int grade > : lowGrades , < int s2, int times > : absence ,
       s1 == s2 , times < 2 }
where rel[int,int] lowGrades = { <s1, grade> | < int s1, int grade > : grades ,
       grade < 5.5 }
```

Here we show that the generator `grades` and the filter `grade < 5.5` are brought outside the original comprehension and inside a newly created comprehension called `lowGrades`. Note that on `C2` the transformation `fh/1` can be applied again, resulting in `C3`:

```
C3     { s1 | <int s1, int grade> : lowGrades , < int s2, int times > : lowAbsence ,
       s1 == s2 }
where rel[int,int] lowAbsence = { <s2, times > | < int s2, int times > : absence ,
       times < 2 }
```

Here we show that the generator `absence` and the filter `times < 2` are brought outside the original comprehension and inside the created comprehension called `lowAbsence`. The comprehension `lowGrades` is not shown here, but is the same as shown in `C2`.

Note that the variable `grade` and `times` are not used anymore in `C2` and `C3` and therefore they could be removed. Before this is done, there has to be sure that no other qualifier uses these variables. In this example no other qualifier uses these variables and therefore resulting in:

```
C4     { s1 | int s1 : lowGrades , int s2 : lowAbsence , s1 == s2 }
where set[int] lowGrades = { s1 | < int s1, int grade > : grades , grade < 5.5 }
where set[int] lowAbsence = { s2 | < int s2, int times > : absence , times < 2 }
```

In `C4` we show that `grade` and `times` are removed from the return values of the newly created comprehensions and also of the variables in the original comprehension.

Condition: `p` contains variables which are not used in the comprehension		
`fh/2`	$\{ v \mid q_0 , p : e , q_1 \}$	$== \{ v \mid q_0 , p' : e' , q_1 \}$ where $e' = \{ p' \mid p : e \}$

The second transformation states that a generator (`p : e`) can be brought outside the comprehension into a new expression (`e'`). If `p` declares variables which are not used in the rest of the original comprehension (q_1), they can be removed from the return variables of the new expression (`p'`). This transformation has also be used whenever possible.

Example:

```
C5     { a | < int a, int b > : AB , < int c, int d > : CD , a == c , d > 5 }
```

The above example shows a generator (`AB`) from which only one of the two variables is used (`a`). Therefore this comprehension can be transformed using `fh/2`, resulting in `C6`:

```
C6     { a | int a : onlyA , < int c, int d > : CD , a == c , d > 5 }
Where set[int] onlyA { a | <int a , int b> : AB }
```

Above we show that the generator `AB` has been brought outside into a new comprehension. Since variable `b` is not used in the original comprehension, it is removed from the return values of `onlyA`.

Performance gain

The performance gain of Filter Hiding is based on placing one or more qualifiers outside the original comprehension into a new comprehension. This ensures that this new comprehension is cached before it is used in the original comprehension. However, the literature claims that this technique does not really performs an optimization, but by transforming the comprehension using one of the transformations of Filter Hiding it increases the possibility for other optimizations, such as Common Subexpression Elimination, to be effective.

Suitability

Filter Hiding can be applied very well to the comprehensions in Rscript since it does not use any techniques that are not supported by Rscript. However before transforming, information has to be gathered about where the variables used in all of the qualifiers are declared.

4.3. Product Elimination

Product Elimination [1] will transform a Cartesian product into a natural join. If the Cartesian product is followed by a equality test between two relations, the result of the natural join is a set of all the combination of tuples that are equal on

their common attribute. The natural join from two relations with arity[3] *a* and *b*, will construct a new relation of arity *a* + *b* – *1*:one of their common attributes will be deleted.
If the Cartesian product is followed by a equality test between one relation and a value, it can be transformed into a comprehension.

Transformation rules
Here the transformations of Product Elimination are explained.

Condition: The form is a Cartesian product followed by an equality between variables.
pe/1 { v | q_0 , <a,b> : AxB , a_i == b_j , q_1 } == { v[a_i/b_j] | q_0 , ab : AB , q_1[a_i/b_j] } where AB = $jmerge_{ij}$ ($sort_i$ A) ($sort_j$ B)

The above transformation states that the Cartesian product (AxB) followed by an equality between variables (a_i == b_j) can be transformed into a natural join. The natural join is computed as follows: first the relations which are joined are sorted on their common attribute, where i and j indicate the position of their common attribute inside the relation. Afterwards a sort-merge function is called ($jmerge_{ij}$) as described in [1]. This will return a new set containing A and B joined together. Since the equality (a_i == b_j) is removed b_j is no longer available. Therefore, wherever b_j is used it has to be changed into a_i because they are equal. This is notated like [a_i/b_j].

Example:
```
C1    rel[int,int] ABCDc = { <a,c> | < <int a, int b>,<int c, int d> > :
      ABxCD , b == c }
```
The above comprehension shows a Cartesian product (ABxCD) and an equality filter (b == c). Therefore, this comprehension can be transformed using pe/1, resulting in C2:
```
C2    rel[int,int] CDc = { <a,d> | < int a, int b, int d > : ABCD }
where rel[int,int,int] ABCD = jmerge21 (sort2 AB) (sort1 CD )
```
The $jmerge_{21}$ function will merge the relations AB and CD on their common attribute, which are b and c (b == c). This will return a relation with all of the elements combined from AB and CD, with one of their common attributes removed (c). Since c is removed and it is equal to b, wherever c is used it has to be replaced by b. This is shown by changing the return values into <a,d>.

Condition: The form is a Cartesian product followed by an equality test.
pe/2 { v_1 | q_0 , <a,b> : AxB , b == v_2 , q_1 } == { v_1 | q_0 , <a,b> : AB , q_1 } where AB = { <a, b> | int b : B , b == v_2 , int a : A }

The second transformation states that a Cartesian product (AxB) followed by an equality test between one of the result values of the product against a hard coded value(b == v_2), can be transformed into a comprehension. This transformation is useful because it will reduce the intermediate result and thereby improving the performance.

Example:
```
C3    rel[int,int] NM28 = { <n,m> | <int n, int m> : N x M , m == 28}
```
The above example shows a Cartesian product (N x M) and a equality test (m == 28). Therefore this example will be transformed using pe/2 into C4:

```
C4    rel[int, int] NM28 = { <n,m> | <int n, int m> : NM }
Where rel[int,int] NM = { <n,m> | int m : M, m == 28, int n : N }
```
This shows that the Cartesian product is replaced by a comprehension. Now the efficiency will be increased because the generator N will be enumerated fewer times.

Performance gain
Product Elimination transforms the Cartesian product into a natural join. Since a natural join can be calculated in a more efficient way than a Cartesian product, an optimization is realized. The degree of optimization depends on the size of the

3 http://en.wikipedia.org/wiki/Arity - the number of domains within the relation

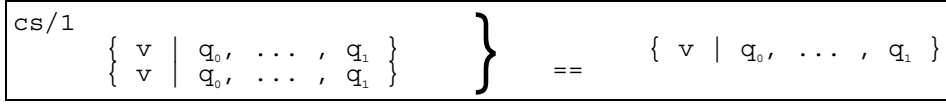
relations that are joined.

Suitability
Rscript does not have a (built in) sort functionality and also the `jmerge` function has to be implemented manually. Therefore `pe/1` is not suitable for Rscript. When this functionality becomes available, either built in or created manually, it can be applied. However, `pe/2` can be applied to Rscript comprehension and is very suitable for optimizing them.

4.4. Common Subexpression Elimination

Common Subexpression Elimination[1,11] is an optimization technique based on removing duplicate expressions within a file. The advantage is that once an expression has been calculated, the next time it is needed it does not have to be calculated again but the result can be obtained directly from primary storage (cache).

Transformation rules
Here the transformations of Common Subexpression Elimination are explained.

Condition: `Two or more equal expressions are present inside the Rscript file.`
`cs/1` 

This transformation states that the duplicate comprehension can be eliminated, and the original can be re-used. This transformation has to be used whenever possible since it will ensure that expressions will not be calculated more than once.

Example:
```
C1    { c | <int c, int d> : CD , d == 99 }
C2    { c | <int c, int d> : CD , d == 99 }
```
The examples above shows two exactly the same comprehensions. This example will be transformed using `cs/1`, resulting in:
```
C3    { c | <int c, int d> : CD , d == 99 }
```
By using Common Subexpression Elimination one of the duplicate relations is removed and the original comprehension (`C1`) is used.
Note that when two expressions are not exactly the same but they do share common qualifiers that `cs/1` *cannot* be applied. However, by using Filter Hiding to place these common qualifiers outside of the expressions into their own expression, the transformation `cs/1` *can* be applied.

Performance gain
Common Subexpression Elimination makes sure that every unique expression is calculated only once. So when the expression is used again, the already calculated result can be reused. The probability that two or more exactly the same expressions exist inside an file increases as the size of the application grows. Especially after a transformation of Filter Hiding has been applied, when each generator with its filters is placed in a separate expression, the odds will increase that a match will occur. The obtained performance gain depends on the number of times a comprehension is reused and the time needed to calculate the expression.

Suitability
Common Subexpression Elimination can be used very well to optimize Rscript. However before applying Common Subexpression Elimination to two or more comprehensions, there has to be determined whether they are identical or not.

4.5. Index Introduction

Index Introduction[1,2] is based on creating an index over a relation. Once the index has been created, every time there is a comprehension that iterates over this relation, the index can be used instead.
A disadvantage of Index Introduction is that it can only be used with the comparison operator '==' and not with ranges (like $<, \leq, \geq$ or $>$) since an index cannot work with ranges.
The creation of the index and the implementation of how the index should be used falls outside the scope of this project.

Transformation rules

Here the transformations of Index Introduction are explained.

Condition: An index over relation e is available.
ii/1 { e | q_0 , p : e , p == v , q_1 } == { e | q_0 , p : $index_e$v , q_1 }

This transformation shows that an index is used over the relation (e). After the transformation, this index can be used to get the values (p) of this relation .

Example:
C1 { a | <int c, int d> : CD , d == 99 , <int a, int b> : AB , b == c }
The above transformation shows that in C1 the relation CD is enumerated in order to match the filter d == 99. After applying ii/1 an index is created to retrieve all the tuples from CD which satisfy the filter d == 99, result:
C2 { a | <int c, int d> : $index_{CD}$99 , <int a, int b> : AB , b == c }

Performance gain
With index introduction a performance gain is obtained since the use of an index is a lot faster than enumerating over a relation. If a relation is being enumerated lots of times, then the costs of creating an index can be regained. The eventual performance gain depends on the number of times the index is used and the amount of time needed for creation of the index.

Suitability
Rscript does not have the ability to use or create real indices. There is the possibility that a relation can be created in such a way that it can act like a index. This will not have the (direct access) advantages of a real index, although this is an implementation issue. Therefore this technique is currently not suitable for application on Rscript. When the index functionality is included in Rscript, it can be applicable.

4.6. Conclusion

There has been done quite some research about optimization techniques which are especially designed for optimizing comprehensions in general. None of them is designed to optimize comprehensions specifically for Rscript, but only for comprehensions in general. Not all of these techniques are suitable for application on the comprehensions of Rscript, partially because of the limited functionality of Rscript. An overview of each of the techniques and their suitability is given in the table below.

#	Optimization technique	Suitable
1	Qualifier Interchange	yes
2	Filter Hiding	yes
3	Product Elimination	partially
4	Common Subexpression Elimination	yes
5	Index Introduction	No/future

Table 1 – Suitability overview for Comprehension optimization techniques.

We have shown that the power of each optimization is based on different methods: Qualifier Interchange, Filter Hiding and Common Subexpression Elimination are based on algebraic transformations in order to obtain a more efficient comprehension. Other techniques are based on creating and using an index over a relation (Index Introduction) or are removing expensive operators by cheaper ones (Product Elimination). For each of the techniques, an example has been given of how these techniques can be applied to Rscript and what their potential performance increase can be. However, the practical performance increase still has to be determined. This will be done in chapter 6.

The next chapter will describe several techniques that are used to optimize Relational Databases and can be applied to Rscript.

5. Database optimization techniques

In this chapter several optimization techniques which are originally used for optimizing relational databases, are discussed. Here, an answer is given to the sub question *I.2*.

For each technique several important points are discussed: transformation rules, performance gain and suitability for Rscript.

Several techniques found in the literature (as mentioned in the background chapter), such as: Copy Optimization[13] and Code Motion[13] are not discussed here since their power for optimization is based only on constructions that are not possible in Rscript. The following techniques are discussed: Commuting Selections, Semantic Query Caching and Peephole Optimization.

5.1. Commuting Selections

Commuting Selections [6] is based on performing selection as early as possible. Hereby the intermediate result is smaller and the operation between the relations is calculated in less time. It can be applied to several operators, such as Cartesian Product, Union and Set Difference.

Transformation rules
Here the transformations of Commuting Selections are explained.

Condition: The form is a selection over a set difference of two relations.			
co/1	$f_1(R_1 \setminus R_2)$	==	$f_1(R_1) \setminus f_1(R_2)$

The above transformations shows that the selection(f_1) over a set difference between two relations is changed into performing this selection on both of the relations before performing the set difference. However when the selection is only using variables which are declared in one relation, for example R_1, then the selection only has to be applied to this relation. This transformation is useful because by performing selections earlier, the amount of intermediate results will decrease and thereby increasing the performance.

Example:
C1 { result | int result : (AB \ CD)[-,3] }
This example shows that the selection is performed over the result of the set difference. After applying co/1 the comprehension is transformed into C2:
C2 { result | int result : (AB[-,3] \ CD[-,3]) }
Here the selection has been performed first over each of the relations, afterwards the set difference is calculated. This will ensure that the comprehension will be calculated more efficiently.

Condition: The form is a selection over a union of two relations.			
co/2	$f_1(R_1 \text{ union } R_2)$	==	$f_1(R_1) \text{ union } f_1(R_2)$

This transformation shows that the selection(f_1) over a union between two relations is changed into performing this selection on both of the relations before performing the union. However when the selection is only used to select over one relation, for example R_1, then the selection only has to be applied to this relation.

Example:
C3 { result | int result : (AB union CD)[-,3] }
This example shows that the selection is performed over the result of the union. After applying co/2 the comprehension is transformed into C4:
C4 { result | int result : (AB[-,3] union CD[-,3]) }
Here the selection has been performed first of all over the relations, afterwards the union is calculated. By first performing the selection, the intermediate result is decreased and the union operator can be performed more efficient.

Condition: The form is a selection over a Cartesian product of two relations.			
co/3	$f_1(R_1 \times R_2)$	==	$f_1(R_1) \times f_1(R_2)$

The last transformation is based on performing a selection not over the result of the Cartesian product, but on each relation individually. However when the selection is only used to select over one relation, for example R_1, then it only has to be

applied to this relation.
Example:
```
C5    { <n,m> | <int n, int m> : N x M , m > 28}
```
Here the selection is performed over the result of the Cartesian product. After applying `co/3` the comprehension is transformed into `C6`:
```
C6    { <n,m> | set[int] M28 <- { m | int m : M , m > 28 }, <int n, int m> : N x
      M28 }
```
Here a new collection is introduced (`M28`) in order to perform the selection over `M`. Afterwards the Cartesian product is performed.

Performance gain
The performance gain of Commuting Selections depends on the size of the relations and the type of operation that is performed. When applying one of the transformations the intermediate result becomes smaller because the selections are performed first on each of the relations. Afterwards the more expensive operations, such as: set difference, union and Cartesian product, is applied to the (smaller) intermediate result and thereby decreasing the amount of time needed to calculate the final result.

Suitability
Although these transformation rules are not specifically designed for comprehensions, but for specific operators (union, set difference and Cartesian product) they can be used very well for Rscript. Since these operators can be present as a qualifier within a comprehension, the performance of the comprehension as a whole will increase when the transformation is applied.

5.2. Semantic Query Caching

Semantic Query Caching (SQC) [8,10] is based on saving one (or multiple) part(s) of query processes by using the cached results of previous queries instead. By re-using the cached result, the amount of time needed for processing the query, is reduced. This optimization is partially the same as Common Subexpression Elimination. The difference is that the origin of this technique lies in the relational database world and that other methods of re-use are possible.
There are four possible situations that a result can be re-used:

- *Nothing in common*
  The result contains nothing. Example: `x == 4` has nothing in common with `x == 5`. In this situation the technique cannot be applied.
- *Identical*
  The result contains everything. Example: `x > 5` is identical with `x > 5`. This is the same as Common Subexpression Elimination and is therefore not discussed here.
- *Subsumption*
  The result contains everything and even more than necessary. Example: `x > 5` subsumes `x > 10`. This is also referred as a subset.
- *Overlap*
  To a certain degree the result overlaps with the necessary information. Example: `x > 10` overlaps `x < 20`. This is also referred as an intersection.

Originally SQC is designed for the use in a client/server environment. Here the cached result(s) can be stored client side and this saves client-server connectivity. To identify what information is cached, a semantic description is used.

Transformation rules
Here the transformations of Semantic Query Caching are explained.

Condition: `p:e and` f_1 `subsumes p:e` f_2.	
`sqc/1` e_1 `= { v | p : e ,` f_1 `}` e_2 `= { v | p : e ,` f_2 `}`	`==` e_1 `= { v | p : e ,` f_1 `}` e_2 `= { v | p :` e_1 `,` f_2 `}`

This transformation applies to the subsumption situation and states that when two almost identical comprehensions the generator (`e`) and its filter (f_1) of the first expression (e_1) subsumes the generator (`e`) a its filter (f_2) of the second expression (e_2), the generator of the second comprehension can be changed to enumerate over the first comprehension.

Example:

```
C1    rel[int,int] CDsmaller99 = { <c,d> | <int c, int d> : CD , d < 99 }
C2    rel[int,int] CDsmaller55 = { <c,d> | <int c, int d> : CD , d < 55 }
```

Here we show that in the examples `C1` and `C2` the same relation `CD` is iterated, only that the filters differ. Since `d < 99` subsumes `d < 55`, `sqc/1` can be applied, result:

```
C3    rel[int,int] CDsmaller99 = { <c,d> | <int c, int d> : CD , d < 99 }
C4    rel[int,int] CDsmaller55 = { <c,d> | <int c, int d> : CDsmaller99 , d < 55 }
```

Condition: `p:e` and f_1 overlaps `p:e` f_2.

`sqc/2`

$$\left.\begin{array}{l} e_1 = \{\ v \mid p : e\ ,\ f_1\ \} \\ e_2 = \{\ v \mid p : e\ ,\ f_2\ \} \end{array}\right\} \quad == \quad \begin{array}{l} e_1 = \{\ v \mid p : e\ ,\ f_1\ \} \\ e_2 = \{\ v^{*1} \mid p : e_1\ ,\ f_2\ ,\ p' : e\ ,\ f_1^{*2}\ \} \end{array}$$

The last transformation applies to the overlap situation and states that when two almost identical comprehensions the generator (`e`) and its filter (f_1) of the first expression (e_1) overlaps the generator (`e`) and its filter (f_2) of the second expression (e_2), the second comprehension can be transformed. Several changes are made by the transformation:
The generator is changed to enumerate the first expression with its current filter (f_2). Next a second generator is added which enumerates over the original expression (`e`) and stores its variables with different names (`p'`). On this generator, the inverse of the filter of the first expression is added(f_1^{*2}) while using the different names. Finally the return value(v^{*1}) is changed in order to include the variables with the different names.

Example:

```
C5    rel[int,int] CDsmaller55 = { <c,d> | <int c, int d> : CD , d < 55 }
C6    rel[int,int] CDlarger30 = { <c,d> | <int c, int d> : CD , d > 30 }
```

Here we show that in `C5` and `C6` the same relation `CD` is iterated, only the filters are different. Since `d < 55` overlaps `d > 30` the transformation `sqc/2` can be applied, resulting in:

```
C7    rel[int,int] CDsmaller55 = { <c,d> | <int c, int d> : CD , d < 55 }
C8    rel[int,int] CDlarger30 = { <c,d>,<c2,d2> | <int c, int d> : CDsmaller55 , d
> 30 , <int c2, int d2> : CD , d >= 55 , d > 30 }
```

In the example above we see that the filter `d < 55` is changed into `d >= 55`, also the return value is changed in order to include all of the results.

Performance gain
The performance gain of this optimization depends on the number of parts that can be re-used and the extent of the similarity between the part and the re-usable query. When an expression subsumes another expression, this expression can be re-used. Its performance gain lies in the fact that this expression probably has less values to iterate and it is already in cache. When an expression overlaps, its performance gain lies in the fact that a part of the result is already in the cache.

Suitability
Semantic Query Caching is suitable for Rscript, since it is based on a transformation with no new operators. However it will be difficult to determine whether a query is (remotely) similar to another one. Since Rscript does not have the possibility to store semantic information about a relation, the identification has to be done otherwise.

5.3. Peephole Optimization

Peephole optimization [12] is a method which optimizes by inspecting the code of a file to identify and modify inefficient sequences of instructions. Its origin does not only lie inside the relational database world, but also in the optimization of compilers.

Transformation rules
Here the transformations of Peephole Optimization are explained.

Condition: -
po/1 $e_1 = e_2$, $\{ e \mid q_0 , p : e_1 , q_1 \ldots q_n \}$ == $\{ e \mid q_0 , p : e_2 , q_1 \ldots q_n \}$

This transformation is based on the elimination of an extra variable (e_1). However when this transformation is applied, the variable (e_1) has to be changed to its value (e_2) everywhere in the entire Rscript file. This technique is also referred to as constant propagation.

Example:
```
C1    rel[str,str] closure = calls+
      set[str] calledTotal = closure[ top(calls) ]
```
Here we show that in `C1` the variable `closure` is used to identify `calls+`. Therefore the transformation `po/1` can be applied, result:
```
C2    set[str] calledTotal = calls+[ top(calls) ]
```
The example above shows that the variable `closure` is removed.

Condition: The expression is a collection of hardcoded values.
po/2 $e_1 = int_1 * int_2$ == $e_1 = int_{12}$

The second transformation is based on calculating hardcoded values once, instead of every time the script is executed.

Example:
```
C3    int result = ( 50 / 2 ) * 4
```
Here we show that in `C3` the several hard coded values are present, therefore `po/2` can be applied, result:
```
C4    int result = 100
```
This shows that the hard coded values have been calculated into one value.

Performance gain
The performance gain of Peephole optimization is based on limiting the amount of variables and a form of pre-processing. Since the creation of variables is not an expensive operation, the performance gain will be fairly limited. Also the improvement of preprocessing variables will not be very impressive.
Although the performance gain is not very high, this optimization can be useful when it is automated. While writing the code, there is not any need anymore to limit the use of variables and of calculations between hardcoded values: the optimization will improve this.

Suitability
Peephole optimization can be used for optimizing Rscript. It is fairly easy to apply these optimizations, although the impact is very limited. Also it is not improving the readability of the file, since every variable that is used for increasing the readability is removed. Note that it is not wise to use Peephole optimization together with Filter Hiding, since Filter Hiding creates extra variables and Peephole will remove them and thereby canceling the optimization gain caused by Filter Hiding.

5.4. Conclusion

We have only found three optimization techniques which are designed for optimizing relational database queries and which are suitable for application to the comprehensions of Rscript. An overview of these optimization techniques is given in the table below.

#	Optimization technique	Suitable
1	Commuting Selections	yes
2	Semantic Query Caching	yes
3	Peephole Optimization	yes

Table 2 – Suitability overview of database optimization techniques

We have shown that the techniques in the table above can be used to optimize Rscript comprehensions, since their power to

optimize is based on using operators or performing transformations that are available in Rscript. If this is not the case, the technique cannot be used in Rscript.
The performance increase Commuting Selections lies in an algebraic transformation into a more efficient form. Other optimizations, like Semantic Query Caching, are based on re-using the result of previous calculated queries. Peephole Optimization is based on pre-processing files.

In the next chapter each of the optimization techniques discussed in the current and the previous chapter will be evaluated.

6. Evaluation of the optimizations studied

In the previous chapters, several optimization techniques have been considered. This chapter answers the sub questions *II.1* and *II.2*.

In this chapter we evaluate each of the algebraic optimizations discussed above. First the efficiency of each of the algebraic optimization techniques will be determined. Next the prototype for automatically applying optimization techniques to Rscript shall be discussed. Also the method of validating the efficiency of the two most efficient determined optimization techniques, shall be discussed.

6.1. Efficiency of the algebraic optimization techniques

Several criteria are selected in order to determine which optimization is considered the most efficient. Each of these criteria will be discussed separately. The following criteria are selected:

1. To what extent does the technique perform an optimization?
2. What is the possibility that a technique can be applied to a Rscript file?
3. What does the relevant literature say about the performance of the optimization?

In the following section, each of the algebraic optimizations will be tested against these criteria. The information needed for the first criteria will be obtained by determining the difference in execution time between a non-optimized and a optimized test script. In order to obtain the information needed by the second criteria, a practical test will be conducted to determine the number of times each of the optimization techniques can be used. For the third criteria the relevant literature will be used.

6.1.1 Evaluation of testcases

For each of the optimizations, a unique test case has been created[4]. Each of the test cases will be executed before – and after the optimization. In order to measure the effect of the optimization, the execution time is measured in milliseconds(ms).

A program has been created, named *runRscript*, which enables us to successively run Rscript files multiple times whilst measuring the average execution time. In order to get a representative average value of the execution time, each test case is executed 20 times. In the table below the results of the measurements are shown. First the not-optimized execution time of the test script is given, next the execution time of the optimized test script. Next the difference between the execution times of these two executions is given. Finally the performance gain of the optimized script is given in percentages.

Optimization technique	Not-optimized	Optimized	Difference	Performance increase
Qualifier Interchange	17746 ms	13526 ms	4220 ms	31.2 %
Filter Hiding	17807 ms	13547 ms	4260 ms	31.5 %
Qualifier Interchange + Filter Hiding	17807 ms	13570 ms	4237 ms	31.2 %
Common Subexpression Elimination	13493 ms	13482 ms	11 ms	0.1 %
Commuting Selections - set-difference	20632 ms	13454 ms	7178 ms	53.4 %
Commuting Selections - union	14285 ms	13531 ms	754 ms	5.6 %
Commuting Selections – Cartesian product	7603 ms	4867 ms	2736 ms	56.2 %
Product Elimination – pe/2	5094 ms	274 ms	4820 ms	1759,1 %
Semantic Query Caching - subsumption	14072 ms	13624 ms	448 ms	3.3 %
Semantic Query Caching - overlap	14618 ms	14348 ms	270 ms	1,9%
Peephole optimization – po/1	17807 ms	50635 ms	-32828 ms	-64,8 %
Peephole optimization – po/2	179 ms	174 ms	5 ms	2,9 %

Table 3 - Optimization measurements. [5]

4 The test cases are documented in Appendix A.
5 The test scripts are executed on a Athlon 64, 3500 MHz, 1GB RAM on Fedora v4.

For every optimization technique discussed in chapter four and five, the table above shows the extent of which they perform an optimization and thereby answering sub question *II.1*.
Although each optimization has only been applied to a test file instead of being applied to an existing application, it gives a good indication of the extent of optimization that each technique performs.

Some of the techniques are capable of working together with other techniques, especially Filter Hiding is a good example. By doing so, the performance gain might be increased even more. In the measurements above, Filter Hiding is combined with Qualifier Interchange. A bit disappointing is that the measurements show that their combined result does not perform better than they do separately. The question arises whether this disappointing result was obtained because the test case used was not designed very well for the combined optimizations or that the techniques really do not perform better combined, as they do separately. This will have to be investigated in future work, because the time available is not sufficient to do this here.

6.1.2. Occurrences of optimizations

In order to determine the probability that an optimization can indeed be applied to a Rscript file, several Rscript files are checked whether optimization techniques can be applied. During the Master Program, a group of students of the course Software Evolution has done several exercises with Rscript and these results will be used here. In order to determine the probability that an optimization technique can be applied, all of the students files will be scanned manually. Every time that a technique could be applied, it is notated. We have obtained thirteen different sets (=78 files) of answers for all of the exercises, and they will all be checked.

The original exercise consists of twelve assignments. We have selected six of these assignments to check for the possible application of optimization techniques.

Exercise number	Qualifier Interchange	Filter Hiding	Common Subexpression Elimination	Commuting Selections	Product Elimination	Semantic Query Caching	Peephole Optimization
1	0	0	0	0	0	0	*
5	1	1	0	0	0	1	*
6	0	0	0	0	0	0	*
8	1	2	0	0	0	2	*
11	2	2	0	0	0	2	*
12	5	3	0	0	0	3	*
Total	9	8	0	0	0	8	*

Table 4 – Occurrences of optimization techniques in student files

*Peephole Optimization is found, although only *Po/1* is found and therefore the exact amount of occurrence is hard to determine.

We can see that only Qualifier Interchange, Filter Hiding and Semantic Query Caching can be applied several times. Common Subexpression Elimination, Commuting Selection and Product Elimination could not be applied at all. The outcome is rather disappointing, we expected that the optimization techniques could have been applied more often. The question arises whether the student files used, are suitable to determine the amount of times an optimization can be applied. In order to get a better view, we checked another Rscript file created by a colleague. This file is used in the process of determining dead code within software systems and is considerable larger than the student files: 500 LOC.

Rscript file	Qualifier Interchange	Filter Hiding	Common Subexpression Elimination	Commuting Selections	Product Elimination	Semantic Query Caching	Peephole Optimization
Dead code file	4	8	0	0	0	8	-

Table 5 – Occurrences of optimization techniques in dead code Rscript file.

As we see in the table above only Qualifier Interchange, Filter Hiding and Semantic Query Caching are found to be applicable. These results are not very different from the results obtained by checking the student files.

After checking almost eighty different code fragments we must conclude that the algebraic optimization techniques cannot be applied very often. Only Qualifier Interchange, Filter Hiding and Semantic Query Caching are found to be applicable several times.

6.1.3. Performance discussed in literature

In the literature several statements are made about the performance of the optimizations. In [1,2,6] is claimed that Qualifier Interchange is the most important algebraic improvement. In [6] is stated that if a Cartesian Product is actually used as a join, it should be transformed into one, this is what Product Elimination does. Together with Common Subexpression Elimination, also discussed in [6], these two optimization techniques are mentioned as very well performing. Negative statements are also found: in [1] it is claimed that Filter Hiding does not improve the efficiency at all. We have found no claims about the performance of the techniques Commuting Selections and Peephole Optimization.

6.1.4. Conclusion

Several findings have bee made while evaluating the optimizations. First of all, the execution time of the performed optimization techniques are very divergent: some techniques barely perform an optimization, such as: Common Subexpression Elimination, Commuting Selections – union, Semantic Query Caching and Peephole Optimization – po/2. Peephole Optimization – po/1 even showed a negative influence on the performance. Other techniques performed very well: Qualifier Interchange, Filter Hiding, Commuting Selections – set-difference, - Cartesian product and Produce Elimination. In this respect Produce Elimination realized the largest increase in performance.

While considering the possibility that an optimization technique can be applied to a Rscript file, the results were rather disappointing. Only Qualifier Interchange, Filter Hiding, Semantic Query Caching and Peephole Optimization were found to be applicable several times, from which Qualifier Interchange could be applied most often in the student files. The other optimizations could not be applied at all.

The relevant literature claimed Qualifier Interchange to be the most efficient, second best performing are Product Elimination and Common Subexpression Elimination. Remarkably, Filter Hiding was said to not be efficient at all: however, our own measurements proved otherwise.

In order to determine which optimization technique can be considered the most efficient for us, all the three factors (the evaluation, the possibility of appliance and the statements made in the literature) are taken into account. When doing so, we can conclude that Qualifier Interchange is the most efficient optimization. Its performance is not measured as the highest of all the optimizations, although it is one of the best performing. It is determined to have the greatest possibility to be applied to a Rscript file. Also the literature claims it is the most important algebraic improvement. The next most efficient optimization is Filter Hiding. Although this optimization is considered by the literature as not improving, the evaluation of its test case shows otherwise and it is determined to be second best applicable. Product Elimination performed the best in the measurements but was not found to be applicable at all, therefore it is ranked third.

6.2. Automation Prototype

In the previous section, all of the optimizations that are suitable for Rscript have been evaluated and the two most efficient optimization techniques have been identified.
In this section the development of the automation prototype will be discussed. First the implementation method that will be used to create the prototype will be determined. Subsequently, the two most efficient optimizations are implemented.

As determined earlier, the optimization techniques that are considered the most efficient are Qualifier Interchange and Filter Hiding. For these optimization techniques, a pseudocode description of the techniques algorithm will be given first. Next it will be added to the prototype.

6.2.1. Implementation method

This paragraph will discuss several possible implementation methods for the automation prototype which were mentioned before (see §3.1.3). Eventually we will determine which method to use further along.

Before discussing the methods, we declare that the main target of this prototype is to prove that an automatic application of optimization techniques is possible and to allow us to optimize the test case Rscript files.

When discussing each of the methods, several issues are addressed: 1) how each of the methods works and of what tasks it

consists, 2) advantages and disadvantages, 3) personal experience with the programming environment and 4) since the time available is limited, the chosen method will have to be able to be constructed in little time.

Extending Rscript in ASF+SDF

This method is based on extending the current implementation of Rscript in order to apply the optimizations. This will be done in ASF+SDF[7]. The advantages of this method are that 1) the syntax definition of Rscript is already available and that 2) a parser for the Rscript files is already available. Although the information needed for each transformation can be obtained reasonably easy, our personal experience and knowledge of ASF+SDF is limited to such a degree that the time needed to obtain this information will be very high. Therefore this method will be hard to realize in the time available.

Complete re-creation of Rscript
With this method, Rscript will be completely recreated in Java. Several key components will have to be created, such as a parser and a relational calculus library. The usage of this method will have several disadvantages: 1) The syntax definition of Rscript is not available and has to be constructed, 2) a parser will have to be created in order to process a Rscript file and 3) because this method will require a lot of work, the amount of time needed for implementing this method will be very high. An advantage is that the information needed for applying transformations will be easily to obtain since the structure will be perfectly adapted to do so.

Standalone application
This method contains of creating a standalone Java application in which Rscript files can be formulated and subsequently be optimized. This method has two disadvantages: 1) The syntax definition of Rscript is not available outside Rscript, this will have to be implemented and inserted into the application and 2) a Rscript parser will not be included, the Rscript files will have to be inserted manually into the application. The implementation of a parser is a reasonably time consuming activity and falls outside the scope of this project, therefore it is discarded.
An advantage of this method is that the information that is necessary for applying transformations, is easy to obtain.

My experience and knowledge of Java is reasonably high, therefore the time needed to use this method for implementing a prototype is limited which is an advantage here.

Conclusion
After discussing each of the proposed methods for implementing a prototype, a decision has to be made about which method shall be used here.

Although the method to extent Rscript in ASF+SDF has the most advantages in comparison with the other methods, it will take too much time to implement this method because of our limited knowledge of ASF+SDF. The complete re-creation of Rscript in Java will be too time-consuming for the main target of the prototype. Therefore the method of a standalone Java application for optimizing Rscript will be used to develop the prototype.

6.2.2. Prototype framework

The main purpose of the framework of the prototype is to validate the results which were obtained by performing the optimization techniques manually to the test cases. Therefore the main target of the prototype is to automatically perform optimizations on a Rscript file and return the optimized file. First we determine what the prototype should be able to do and what it will not have to do:

Should be able to	Will not have to do
• Manually input Rscript files • Express most of the Rscript syntax • Perform automatic optimizations • Combine optimization techniques • Produce an optimized Rscript file	• Parse Rscript files • Execute Rscript files • Completely contain the Rscript syntax

Table 6 – Overview possibilities of prototype.

Secondly the prototype is created. While creating the framework of the prototype, the implementation of Rscript in ASF+SDF is used as the main reference. The architecture of the prototype is discussed in appendix C.

6.2.3. Pseudocode algorithms

For both of the optimization techniques that will be implemented into the framework, a pseudocode algorithm is constructed. The goal of these pseudocode algorithms is 1) to identify the information that is necessary in order to perform

the optimization, 2) to simplify the implementation of these optimization techniques and finally 3) explaining the optimizations for future use by the CWI.

Filter Hiding

In order to create a working algorithm for Filter Hiding, we have to determine when a generator or a filter can be placed outside of a comprehension. In order to obtain this information, the following situations have been identified in which generators and filters can occur:

(1) A generator uses one or more variables which are declared inside the comprehension.

(2) A generator uses one or more variables which are declared outside the comprehension

(3) A generator uses one or more variables which are declared inside and outside the comprehension

(4) A generator uses no variables

(5) A filter uses variables which are declared in one generator

(6) A filter uses variables which are declared in more generators

The generators from situation 2 and 4 are candidates for Filter Hiding because the generator should not be dependent of anything within the comprehension, since it will be placed outside the comprehension. The filter from situation 5 is also a candidate for Filter Hiding because the filter should not be dependent of multiple generators. The rest of the candidates are not suitable for Filter Hiding.

The pseudocode for the Filter Hiding algorithm is given below.

```
foreach ( comprehension c ) {

  //all declaration variable in the entire comprehension
  collection declVarsInComprehension

  foreach ( qualifier in comprehension c ) {
    determine variables which are used in the qualifier (used vars)
    determine variables which are declared in the qualifier (decl vars)
    add decl vars to declVarsInComprehension
  }

  //get all generators who match situations 2 and 4
  collection candidateGenerators
  foreach ( generator g in comprehension c)
    if( g.usedVars == 0 ) { add g to candidateGenerators } //situation 2
    elseif( all g.usedVars notIn declVarsInComprehension ) { //situation 4
      add g to candidateGenerators
    }
    else {} //rest of the situations
  }

  //iterate over all the candidateGenerators and add the filters who match situations 5 and 8.
  foreach( generator g in candidateGenerators ) {

    //a collection of generator g and the filters which shall be applied to Filter Hiding
    collection gFilters

    foreach( filter f in comprehension c) {
      //all declaration variables in the entire comprehension who are used by filter f
      declVarsInComprehensionUsedByF = f.usedVars inter declVarsInComprehension

      if( #declVarsInComprehensionUsedByF != 0 ) {
        //if the variables used by filter f from inside the comprehension
        //are all in the declaration variables of generator g, add it to the collection
        if( all declVarsInComprehensionUsedByF in g.declVars ) { //situation 5
          add filter f to gFilters
        }
      }
    }

    if( #gFilters > 0 ) {  // fh/1
      apply fh/1 on the generator g with the filter collection gFilters )
    } else { // fh/2
      apply fh/2 on the generator g
    }
  }
}
```

We will now discuss the algorithm. The main target of the algorithm is for each of the comprehensions to select the generators, with or without filters, which can be applied to `fh/1` and `fh/2`.
There is iterated over each comprehension individually. First of all it is determined for each of the qualifiers in the comprehension which variables they use and which they declare. All of the declaration variables are stored in the collection `declVarsInComprehension`. Subsequently there is iterated over all of the generators of the comprehensions, in order to obtain the generators which are a candidate for Filter Hiding. If the generator matches situation 2 or 4 it is added to the collection `candidateGenerators`. These generators can already be applied to `fh/2`, however we have to check first for each of the generators whether there are filters who only use variables which are declared in the generator. If this is the case `situation 5` is matched, and the filter can be brought outside. At the end of the algorithm, we check whether there are filters who can be brought outside, if this is the case then the transformation `fh/1` can be applied, else `fh/2` can be applied.

This algorithm can be expanded in such a way that it is also possible to bring out groups of generators with their filters. However because of the limited time, this possibility is discarded here.

Qualifier Interchange
Before creating an algorithm for Qualifier Interchange, we have to realize that there is no algorithm of Qualifier Interchange that can transform every comprehension into the optimal form. The transformations of Qualifier Interchange describe several possibilities to swap: two filters(`qi/1`), two generators(`qi/2`) and a generator with a filter(`qi/3`). These can be abstracted to the swapping of any two qualifiers. This abstract view is used in this algorithm.
Since there are too many possibilities to determine the order of the qualifiers within the comprehension, we will choose one of the following options:

(1) To re-arrange the qualifiers in ascending order of the amount of their used variables

(2) To re-arrange the qualifiers in order to ensure that they are situated as close as possible to the qualifier which declares its used variables.

(3) A combination of (1) and (2).

In this algorithm option 3 is used because we suspect that this option will probably give the best results because 1) the evaluation of a qualifier with less variables is cheaper than a qualifier with more variables and 2) when a qualifier only uses variables which are declared outside the comprehension, these variables are already cached and therefore they are cheaper to evaluate. In this algorithm only filters will be checked on these points and the generators who declare their used variables are added when the filter is added.

In order to obtain the order of the qualifiers, first we have to determine where the used variables of a qualifier can be declared. Two cases can be identified: 1) outside the comprehension, which can also be in the arguments of the variable which contains the comprehension and 2) within of the comprehension itself.

The pseudocode algorithm for Qualifier Interchange is given below.

```
foreach ( comprehension c ) {

  //all declaration variable in the entire comprehension
  collection declVarsInComprehension

  foreach ( qualifier in comprehension c ) {
    determine variables which are used in the qualifier (used vars)
    determine variables which are declared in the qualifier (decl vars)
    add decl vars to declVarsInComprehension
  }

  //new comprehension based on comprehension c (same return values)
  Comprehension newC

  // the number of generators in the comprehension c
  int generatorsInComprehension

  for ( nr = 0 to generatorsInComprehension ) {
    foreach( filter f in comprehension c ) {

      // all the used variables of filter f.
      collection usedVarsF
      // all the generators of c who declare one or more of the variables in usedVarsF
      collection generatorsF

      // add the filters (+ the generators who declare its used variables) to the new
      // comprehension, in order of the amount of generators its variables are declared in.
      if( #generatorsF == nr ) {                                    // option 1
        foreach( generator g in generatorsF ) {
          add generator g to comprehension newC
        }
        add filter f to comprehension newC
      }

      // add the filter to the new comprehension if the variables used
      // by the filters are declared outside the comprehension
      if( usedVarsF not in declVarsInComprehension ) {              // option 2
        add filter f to comprehension newC
      }

      // add the filters whose used variables are declared in more than nr generators,
      // but these generators are already placed inside of the new comprehension
      foreach( filter f2 in the comprehension c ) {                 // option 3

        // all the used variables of filter f2.
        collection usedVarsF2
        // all the generators of c who declare on or more of the variables in usedVarsF2
        collection generatorsF2

        if( comprehension newC contains all the generators in generatorsF2 ) {
          add filter f2 to comprehension newC
        }
      }
    }
  }
  replace comprehension c with comprehension newC
}
```

The main target of the algorithm is to order the qualifiers within a comprehension in the way described above, for each comprehension.

The algorithm iterates over every comprehension. First of all it is determined for each of the qualifiers in the comprehension which variables they use and which they declare. All of the declaration variables are stored in the collection `declVarsInComprehension`. Then a new comprehension is created in which the optimized order of filters will be constructed. All of the filters are iterated `nr` times, where `nr` is the number of generators within the comprehension. This is done to determine when a filter can be added to the new comprehension: when the variables used by a filter are declared in `nr` generators from within the comprehension, its generators and the filter are added to the new comprehension (`option 1`). When a generator or filter is added to the new comprehension, it is checked if it is not already present. When the used

variables of the filter are not declared in the comprehension, the filter can be added to the new comprehension (`option 2`). Subsequently another loop is started where it is checked for every filter whether the generators which declares their used variables are already placed inside the new comprehension. If this is the case, then the filter is added (`option 3`). Eventually the original comprehension is replaced with the newly constructed one.

Note that it is not sufficient to arrange the filters in ascending order of the amount of generators in which their used variables are declared. We will explain this with an example: when there is a comprehension with three generator each with one filter and a fourth filter who uses variables which are declared in two generators. The fourth filter will be placed at the end of the comprehension if the filters are solely arranged by the amount of generators in which their used variables are declared. When the above algorithm is used, the filter will be placed directly after the generators who declares its used variables.

6.2.4. Implementation details

The prototype developed is created in Java and consists of 102 classes with a total of 5000 LOC. The Rscript language definition is expressed in 73 classes. The optimization algorithms are expressed in 8 classes. The rest of the classes are used for testing purposes, test case Rscript files and unit tests.
A total of twelve days has been used to develop the prototype.

6.3. Evaluation of automation prototype

The algorithms created for the optimization techniques have been added to the prototype. Afterwards the test cases of Filter Hiding and Qualifier Interchange have been inserted manually into the prototype and subsequently each optimization has been applied automatically.
It is expected that, to a large extent, the results will be the same as the ones obtained by applying the optimizations manually to these test cases.
Each of the test cases will be executed before – and after the optimization and this will be repeated 20 times. The results are shown in the table below.

Optimization technique	Not-optimized	optimized	Difference	Performance increase
Qualifier Interchange	17965 ms	13803 ms	4162 ms	30,2%
Filter Hiding	17965 ms	13879 ms	4086 ms	29,4 %
Qualifier Interchange + Filter Hiding	17965 ms	13911 ms	4054 ms	29,1 %

Table 7 - Measurements test cases after optimization with the automation prototype.[6]

When we compare the results above with the results of optimizing test cases manually in table 3 (see §6.1.1), we can see a great similarity in the way each of the optimization techniques performs. When comparing the outcome of the test cases after applying them to the prototype, we can see that this is almost identical to the test cases after manually optimization. The only difference is that for the Filter Hiding and Qualifier Interchange + Filter Hiding test cases, the prototype does not remove the unused qualifiers in the comprehension which is taken outside of the original comprehension. However, this was expected since the algorithm for Filter Hiding did not include this removal.

6.4. Validation

In order to determine that the prototype does not only work on the test scripts but also on 'real' existing Rscript files, the student files used earlier will be inserted into the prototype. The results from this optimization are also used to validate the extent of the performance increase of Qualifier Interchange and Filter Hiding (and both together). Since these optimization techniques were only found applicable to exercises five, eleven and twelve (see §6.1.2), these exercises are used.

6 The test scripts are executed on a Athlon 64, 3500 MHz, 1GB RAM on Fedora v4.

Measurements	Exercise number		
	5	11	12
Not Optimized	402 ms	392 ms	409 ms
Qualifier Interchange + Filter Hiding	380 ms	386 ms	389 ms
Difference	22 ms	6 ms	20 ms
Performance increase	5,8 %	1,6%	5,1 %

Table 8 - Measurements of student files after optimization with prototype.[7]

We can see that a performance increase is achieved by using the automation prototype on the student files. However, the performance increases are not nearly as high as the optimization results of the test cases, either manually or automatically. The performance increase of these optimization techniques depends on the situation on which they can be applied, therefore the question arises whether these student files are not suited very well for the optimization techniques used or that the optimization techniques are not as powerful as the test cases showed? Since almost every case where these optimization techniques can be applied is unique, each situation is different and its performance will also be different every time. We suspect that the constructions within these student files are just not that suitable for application of these optimization techniques. Further, we suspect that the performance increase of Qualifier Interchange and Filter Hiding will almost never be higher in regular Rscript files than the increase we observed with the test cases, since these situations were ideal.

6.5. Conclusion

In this chapter several algebraic optimization techniques are evaluated. First we have determined the efficiency of each of the optimization techniques by looking at their performance increase, their occurrences of application in several Rscript files and their performance increase claimed by the relevant literature. We concluded that Qualifier Interchange and Filter Hiding are the two most efficient techniques.

In order to create a prototype to automatically apply optimization techniques to Rscript files, there has been determined which method to use for creating the prototype. Earlier we selected three different methods, from which we selected the standalone application to be the method to be used here. Next the automation prototype has been created and implemented according to the proposed architecture and all of the requirements which were stated earlier are fulfilled. The algorithms of Qualifier Interchange and Filter Hiding have been thoroughly discussed and are also added to the prototype. Although these techniques are not very complicated, the construction of a pseudocode algorithm was. A lot of information is needed about dependencies from each qualifier to every other qualifier, in order to correctly apply the transformation techniques. In the future, these algorithms can be used to optimize similar situations.
Although the optimizations can now be applied automatically, the Rscript files still have to be inserted manually into the prototype. In order to make the prototype better applicable in the future, it might be desirable to extend it with a parser in order to automate this process.

The test cases have been inserted into the prototype and have subsequently been executed. The results are shown in table 7 (see §6.3). These results show a great similarity to the results obtained by the manual application of optimization techniques to the test scripts. Therefore we concluded that for the test cases, the prototype does its work properly.

In order to validate the extent of the optimization of the techniques, the student files, which where also used earlier, have been inserted into the prototype. The obtained results are disappointing: the optimized files only perform a maximum of five percent better than the original files, while the test cases showed an increase of thirty percent. Not every file is equally suited for the application of these optimization techniques and in order to give a better validation, other Rscript files will have to be optimized using the prototype and subsequently be evaluated.

The next chapter describes several optimization techniques in order to achieve set optimization and determines their effect.

7 The test scripts are executed on a Athlon 64, 3500 MHz, 1GB RAM on Fedora v4.

7. Set Optimization

In this chapter an answer is given to the sub questions *I.3* and *II.4*.

First several data structures which can be used for set representation are discussed. Afterwards we will discuss the development of a prototype which enables us to determine the effect of using other data structures.
Eventually several measurements will be performed on this prototype and an advice is given about which data structure to use in order to maximize the performance gain.

7.1. Data Structures

When in the current implementation of Rscript a collection of values is stored, a set implementation is used in order to create this collection. When a set with, for example, integers is stored it will have the following (Linear) data representation:

`Set[int] = i1 … in`

When a new integer is added to a set, first it will be determined whether this (integer) value is already present within this set, which is achieved by using the Rscript operator `in`. If the value is not present, it is added in front of the other values from the set. So before adding, a lookup is required which is in total done in O(n) time.

Almost all of the Rscript functionality that depends on the data structure of a set, can be implemented using one or more of the following operations: insert, delete, search, maximum, minimum and union. Therefore the emphasis will be on data structures who perform these operations efficiently, such as: Hash Tables, Binary Search Tree, Red-Black Trees, Binomial Heap, and Judy Arrays.

7.1.1 Hash Tables

With hashing[14,15] elements are stored inside a hash table at a certain position. The location of an element inside this table is determined by a hashing function which delivers an almost unique key. With this key, the element can be retrieved from the table in constant time O(1), independent of the size of the table. There are two conditions to this hashing function. The first condition is that for a given element it always delivers the exact same key. The second condition is that this key can be interpreted as a natural number in order to perform a lookup in the table.

When constructing a hash table, an array is created of a certain size. The question of how large the (hash table) array should be can be answered in different ways[14]: 1) If the data set is of known size and a perfect hashing function can be used, then we make the table the same size as the data set, 2) if a perfect hashing function is not available or practical but the size of the data set is known, as a rule the table is made 150 percent of the size of the data set and 3) If we do not know the size of the data set, dynamic resizing can be used. Dynamic resizing of a hash table is based on creating a new hash table and inserting all of the elements of the original table into the new table and subsequently removing the original table. To decide when to resize a hash table, the load factor is introduced. The load factor is the percentage of how many elements in the table are filled in comparison to its size. A hash table can have a load factor of 0.7 – 0.8 and still perform well[8]. Java uses a load factor of 0.75.

The hashing function is said to be a perfect hashing function when it maps each element to an unique position inside the table. However when the function is not perfect, collisions may occur. A collision occurs when the hashing function of two different elements delivers an identical key. Two techniques are most commonly used to handle collisions: chaining and open addressing. With chaining all the elements in the hash table that have the same key are put in a linked list(either singly- or doubly-linked). Inserting elements takes O(1), deletion can also be done in O(1) when doubly-linked lists are used. Searching an element is dependent of the length of the linked-list, however when we assume simple uniform hashing[9][15] it takes O(1 + *load factor*) which can be rewritten to O(1), see [15].
With open addressing all the elements are stored in the hash table itself and collisions are avoided by determining another open position in the hash table to place the element in. This open position is discovered by probing the hash table. Three techniques are commonly used to probe the table: linear probing, quadratic probing and double hashing. The main problem with open addressing is the deletion of elements. When an element is deleted, the position it occupied in the hash table has to be specially marked as deleted since the search algorithms have to stop probing when an empty slot is found. For this reason, chaining is used more commonly as a collision handler than open addressing.

8 http://en.wikipedia.org/wiki/Hash_table - a hash table can contain about 70%–80% as many elements as it does table slots and still perform well.
9 The assumption or goal that items are equally likely to hash to any value

The main problem with hashing is to find a uniform hashing function that will lower the risk that a collision will occur by distributing the keys over the table as dispersed as possible. Also the percentage of used elements within the reserved array can be very low and thereby making it space inefficient.

7.1.2 Binary Search Tree

A Binary Search Tree(BST)[15] is represented as a binary tree, therefore each node in the tree has two child nodes: left and right. Each node also knows who its parent is. The BST must satisfy the *binary-search-tree properties*[15] which states that for each node *n*, every node in its left-subtree is smaller or equal to *n* and that every node in its right-subtree is larger or equal to *n*. Note that the BST does not have to be balanced in order to operate.
The operations insert and delete cause the representation to change and then it has to be modified. By doing so, it must continue to satisfy the BST-properties. Searching and inserting elements into the BST can be done in O(*h*) where *h* is the height of the tree. However the deletion of an element is a bit more complex operation. When deleting element *e*, there are three cases that can occur: 1) If *e* has no children, the parent of *e* must be modified to replace its child *e* with no child. 2) If *e* has one child, *e* is removed by creating a new connection between its parent and its child. 3) If *e* has two children, we have to obtain the successor of *e*, called *s*, which has no child and replace *e* with *s*. In this case the successor (the left-most child of the right subtree) can also be the predecessor (the right-most child of the left subtree). When deleting an element, this element has to be found first and depending on the situation also the successor has to be found. However, all these actions can be done in O(h) and therefore the deletion of an element can also be done in O(h).

It is clear is that all the discussed operators can be done in O(h). However, since the BST does not have to be balanced, its worse case representation is in a way that the height is equal to the number of elements. In this case the operations on BST are done in linear time. The lack of guarantee that the tree is balanced is considered the main disadvantage of BSTs.

7.1.3 Red-Black Trees

A Red-Black Tree (RBT)[15] is a form of a BST where the nodes are colored red or black and the entire tree is balanced. For each node that has only one or none children the missing nodes are added which are called leaves. The RBT must satisfy the *red-black-properties*: 1) every node is red or black, 2) the root is black, 3) every leaf is black, 4) if a node is red, then both its children are black and 5) every path from a node to a descendant leaf contains the same number of black nodes, this is called black-height.
Stated in [15] is that the height of the RBT is at most: log2 (n+1), where n is the amount of internal nodes. Since the RBT is a form of a BST, the operations search and mix/max can be performed in O(h), where *h* is the height of the tree. Hereby can be concluded that these operation can be performed on a RBT in O(log n).
When inserting or deleting elements, the structure of the tree will change and therefore it has to be balanced. This can be done by rotating the subtrees. There are two rotations possible: left rotation and right rotation[10]. These rotations can be performed in constant time O(1). After the rotation it has to be checked whether the RBT still satisfies the red-black-properties, this can be done in O(log n) time[15].

All the operations on an RBT can be performed in O(log n). A disadvantage of RBT is that a lot of extra steps are necessary in order to keep the tree balanced. However according to the literature, when they are done properly, they do not influence the performance of the RBT.

7.1.4 Binomial Heap

A Binomial Heap(BH)[15] is a collection of Binomial Trees (BTs). The BT B_0 contains one node, the BT B_k consists of two BTs with one B_{k-1} and one B_{k-2} that are linked together: the root of the one (k-1) is the leftmost child of the root of the other (k-2)[15]. The BT must satisfy the *binomial-trees-properties*[15] for a BT B_k: 1) there are 2^k nodes, 2) the tree is of height *k*, 3) there are exactly

$\binom{k}{i}$ nodes at depth *i* for *i* = 0, 1, ... *k* and 4) the root has degree *k*, which is greater than that of any other node. If the children of the root are numbered (*i*) from left to right by *k-1*, *k-2*, ... , *0*, the child *i* is the root of a subtree B_i. Example:

10 http://en.wikipedia.org/wiki/Tree_rotation - changes the structure without interfering with the order of the elements.

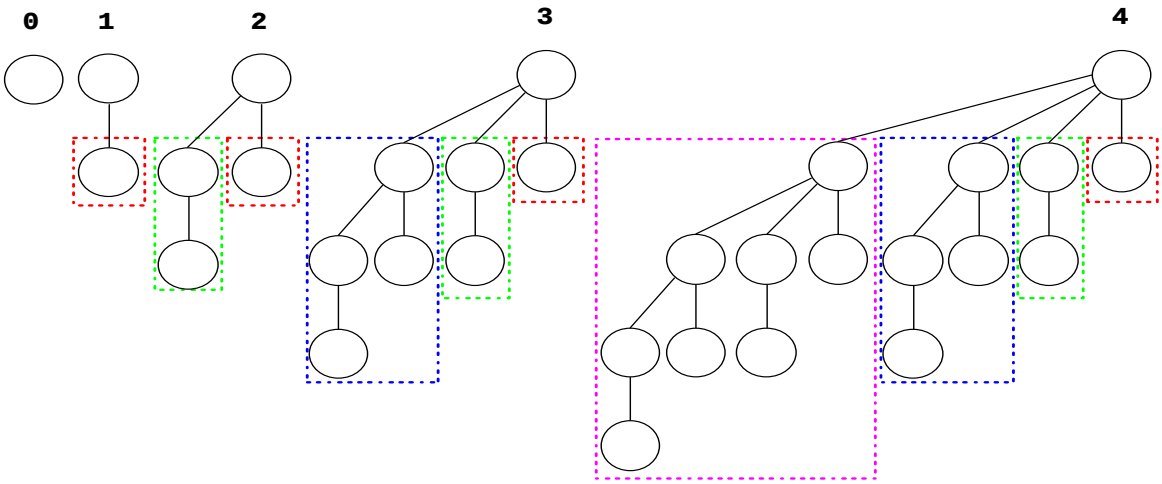


Figure 1 – Binomial Heap construction example.

In the figure above, we show several BHs of different degrees. Its shown that a BH of depth 3 consists of a BH of depth 2, a BH of depth 1 and of 0.
A BH must satisfy the *binomial-heap-properties*[15]: 1) Each BT in a BH is *heap-ordered*: the key of a node is greater or equal to the key of its parent and 2) there is at most one BT in a BH whose root has a given degree. Most of the operations can be done by using the operation union. The union of two BHs can be done very efficient if they are of equal depth k: the united BH becomes of depth k+1 with its root the minimum of both BHs. The other BH can be directly attached to the new BH, this is also shown in figure 1. Merging BHs with different depths can also be done efficient, in O(log n). The main disadvantage of BHs is that the search of an element takes O(n).

7.1.5. Judy Arrays

Judy[16] has been brought to our attention with the question whether it can be used as a data structure for Rscript. Judy is a library for C that can be used to replace different types of data structures. The reason that it is taken along in this investigation is that it claims to be faster than tree-form data structures and even faster than hashing. This would mean that it should be able to perform operations in O(1) time or better. It supports the following operations[16,18]: insert, delete, find, find first/last, find previous/next and count. A disadvantage of Judy is that it does not support all of the operations needed for Rscript, especially the absence of the operator Union makes the usability a problem.
The Judy Array uses a Digital Tree (trie) as the underlying data structure. An advantage of tries is that the memory necessary is much less than with other tree structures or hashing.
In the relevant literature, we found a paper that compares the performance of Judy against Hash Tables[17]. This paper has several important findings and we will discuss a few here: First of all it shows and confirms our earlier findings that Judy is far more space efficient than hashing. Further along, it states that Judy is specifically aimed at situations where a very large data structure is necessary (1 million entries or more) or for strictly sequential data. For smaller data structures and non-sequential data, it is shown that a hash table is faster. The measurements performed in this paper points out that Judy is not optimized for insertions and deletions: hash tables are fare more efficient in these operations.
The manual of Judy [18] shows by performing measurements, that Judy is faster than hash tables. However, in this paper the hash tables are implemented with a fixed-size hash table with external chaining, which does not uses the full performance gain of hashing. In [17] new measurements are done with a more efficient hash table algorithm and this shows that hash tables are, in most situations, faster. A final note of the author of [17] is that there are not very much ways to optimize Judy however there *is* a lot of room to improve hash tables.

7.1.6. Conclusion

We have discussed several data structures which can be used to represent a set as used in Rscript. Below, an overview is given of their theoretical performance with respect to the most important operators. If the performance of an operator is unknown, this is indicated with a question mark. If the operation cannot be performed, this is indicated with an x.

Technique	Insert	Delete	Search	Max	Min	Union	Remarks
Hash Table	O(1)[*2]	O(1)[*1]	O(1)[*2]	O(n)	O(n)	?	[*1]: When double-linked lists are used. [*2]: Simple uniform hashing is assumed
Binary Search Tree	O(h)	O(h)	O(h)	O(h)	O(h)	?	h: height of the tree
Red-Black Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	?	
Binomial Heap	O(log n)	O(log n)	O(n)	?	O(log n)	O(log n)	
Judy Arrays	*	*	*	?	?	x	*: Claims to be faster than Hash tables, no specific data available.

Table 9 – Theoretical performance of basic operations on data structures

Several things can be observed from the table above. First of all, the performance of the Union operator is not discussed in the literature concerning Hash Table, Binary Search Tree and Red-Black Tree. For Judy Arrays, the Union operator is not supported at all. Only for Binomial Heaps, the performance of Union is known. Therefore we cannot compare these data structures on base of their performance of the Union operator.
When looking at the operators Insert, Delete and Search, we can compare the different data structures and we see that the Hash Table performs the best and the Red-Black Tree data structure performs second-best. The Red-Black Tree is the best with respect to the Min and Max operators. The Judy Array data structure claims to be faster than Hash Tables. However, the research in [17] proved otherwise.

Overall we can conclude that theoretically the Hash Table is the best performing data structure for the discussed operators, although its performance for Union is unknown. Second best is the Red-Black tree. In order to determine which data structure will have to be used by Rscript to get the best performance, these data structures will be tested in detail.

7.2. Prototype

A data structure prototype will be developed in order to determine the effect of using other data structures for set representation (sub question *II.4*). It will be implemented in Java and contains three data structures: 1) a linear implementation based on the ASF+SDF set-implementation of Rscript, 2) the data structure which is theoretically determined to be the best performing: a Hash Table and 3) the theoretically second best performing data structure: a Red-Black Tree.

The main purpose of this prototype is to measure the time needed for basic operations on data structures. This will enable us to compare the performance of different data structures. The prototype cannot apply entire Rscript files onto the data structure(s), but the set-operations of the Rscript files will first have to be extracted manually from these files and subsequently they can be inserted into the prototype. This process is not automated because of the limited time available .

The architecture of the data structure prototype is described in appendix D.

In order to validate the correct working of each of the implementations, we created several test cases for each operator. This ensures us that when an operation is executed on any of the tested data structure, it is performed correctly.

7.2.1. Implementation details

The prototype is developed in Java and consists of 17 classes with a total of 2200 LOC. The data structures are expressed in 7 classes. The measurements on operators are expressed in 5 classes. The rest of the classes are used for testing purposes. A total of eight days has been used to develop the prototype.

7.3. Results

In order to determine the performance of each of the three data structures in the prototype, an existing Rscript file is used to approach reality. We have received a file called *TestSpeed.Rscript*[11] from a colleague, which is originally used to test the speed of Rscript. In order to use this file with the prototype, the operations performed on a set within this file have been extracted (manually). Subsequently, these operations are inserted into the prototype. Finally, each of the data structures has

11 The content of the file TestSpeed.rscript is placed in Appendix E.

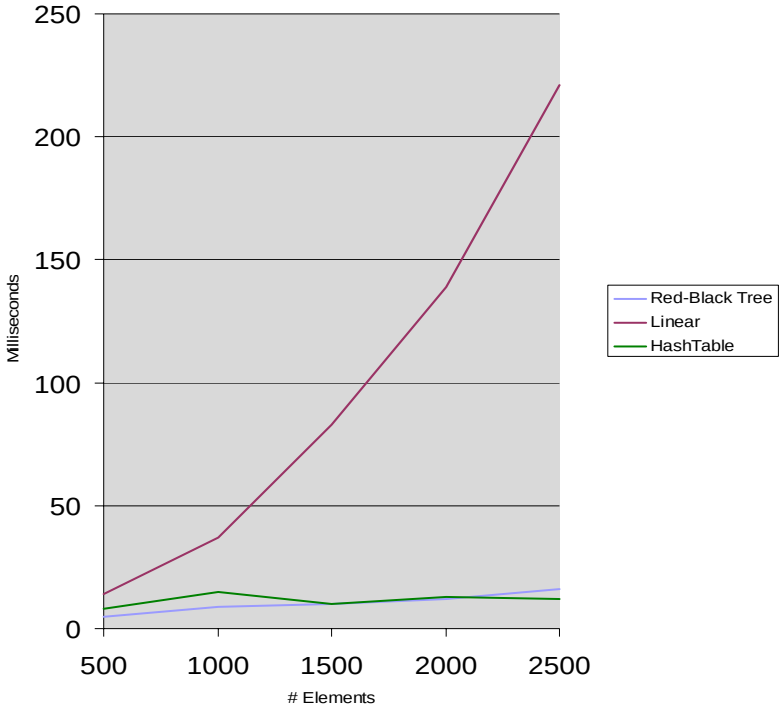
been used in order to perform these operations. The measurements are repeated 20 times in order to get a representative value[12]. The following table shows the results.

Data structure	Time
Linear implementation	32 ms
Java Hashtable implementation	20 ms
Red-Black Tree implementation	14 ms

Table 10 – Measurements of data structures while executing TestSpeed.Rscript

The measurements above give an impression of how each of the data structures performs while executing the set operations of a specific Rscript file. We can see that the Linear data structure performs the worst. Interestingly, the theoretically best performing data structure Hash Table is performing worse than the Red-Black Tree. Since these measurements are specific for the tested file, we cannot give an advice based on these measurements only. In order to get a better understanding of how each of the data structures performs, we will look at the performance of each of the operators: Insert, Min/Max, Union and In(search) separately. These measurements are discussed below.

Insert operator



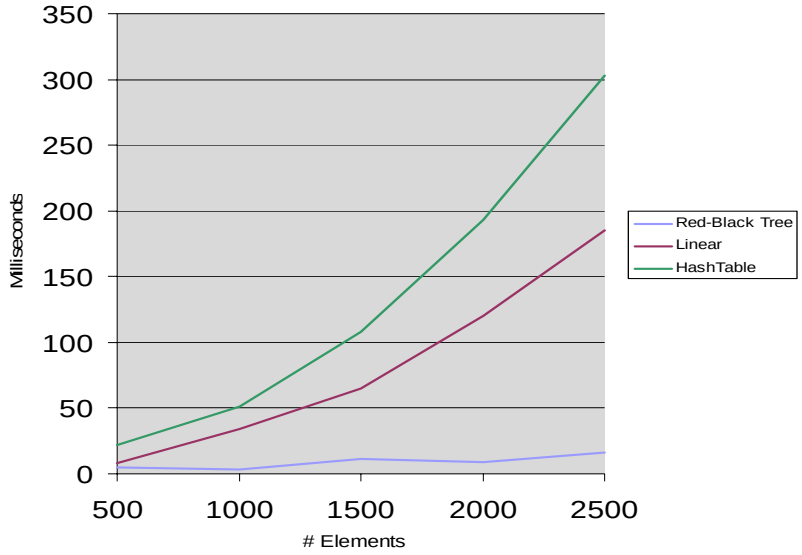
# Elements	Linear	RBT	Hash
500	14	5	8
1000	37	9	15
1500	83	10	10
2000	139	12	13
2500	221	16	12
…			
30000	24187	193	194

Figure 2 – Performance of the Insert operator, times are in milliseconds

The figure above shows the time needed for inserting elements into each of the data structures. A lower graph indicates that the data structure can execute the insert operator faster. We can see clearly that the Linear data structure performs the worst, since its graph is the highest. Curiously, the results of the Hash Table and the Red-Black Tree are almost similar, which is different from their theoretical performance as showed in table 9, in which the Hash Table is superior. Our first thought as to why the Hash Table does not perform better than the Red-Black Tree here, was that the hash function performed poorly. However when we adapted our hashing function to perform better, the results were still similar. As yet, we have no explanation for this similarity. Therefore we have to conclude that for the Insert operator, both the Hash Table and the Red-Black Tree are equally fast performing.

12 The measurements are executed on a Athlon 64, 3500 MHz, 1GB RAM on Fedora v4.

Min operator

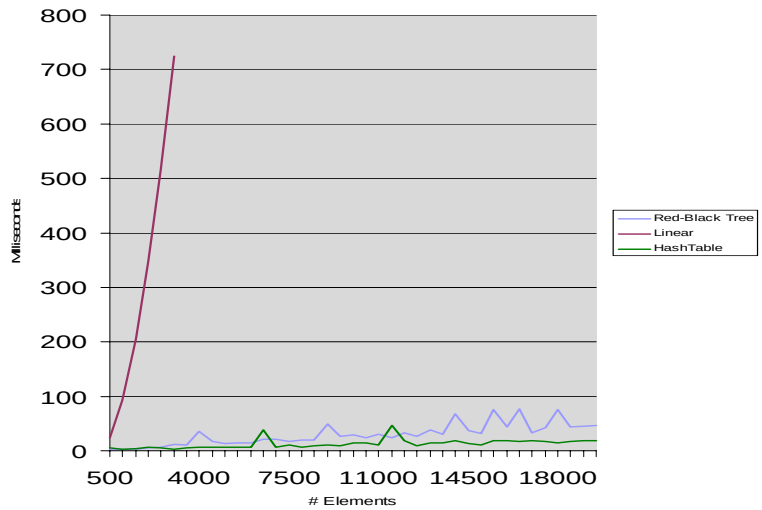


# Elements	Linear	RBT	Hash
500	8	5	22
1000	34	3	51
1500	65	11	108
2000	120	9	193
2500	185	16	303
…			
30000	20928	157	47861

Figure 3 – Performance of the Min operator, times are in milliseconds

The figure above shows the time needed to execute the Min operator over a collection of different sizes. A lower graph stands for a better performance. We can see that the Hash Table has the worst performance, this is because in order to determine the minimal element, it has to iterate over all its table slots and for each table slot it has to check the entire linked list. Second best performing is the Linear data structure, which also has to iterate over each element. We can see that the performance difference between the Hash Table and the Linear data structure is getting larger when the number of elements are increasing. This is caused by the hashing-function, which does not disperse the elements very well throughout the table and then the table slots get more and more filled when the number of elements increases: this lowers the performance.
The best performing data structure for the Min operator is the Red-Black Tree, which is also claimed by the literature (see table 9). Because the Red-Black Tree is ordered, this operator can be executed very fast.
We have also looked at the Max operator: its results are similar to the Min operator – the Red-Black Tree performs best.

Union operator



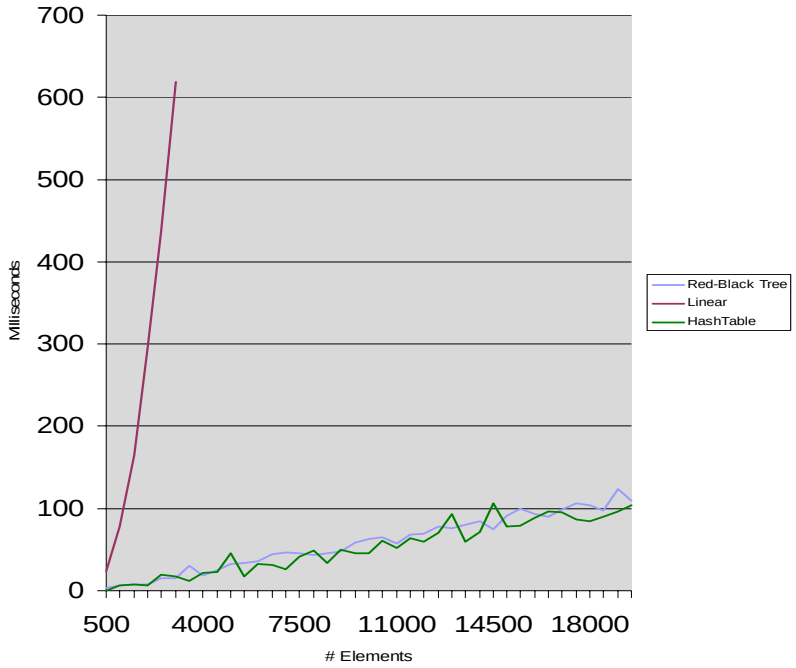
# Elements	Linear	RBT	Hash
500	24	2	5
1000	93	2	3
1500	206	3	4
2000	348	5	7
2500	516	6	5
…			
30000	33384	65	44

Figure 4 – Performance of the Union operator, times are in milliseconds

In the figure above, the performance of a Union of two data structures each of a certain size is shown. We can see that the Union of two Linear data structures is very expensive. We restricted the visibility of the Linear data structure graph in figure

4 to 3000 elements, because otherwise the diagram would be distorted. However in the table next to the graph we can see that with 30000 elements, the time necessary for the Linear algorithm to perform a Union is more than 30 seconds.
We can see that the performances of the Red-Black Tree and the Hash Table are very close to each other. However, the Hash Table performs slightly better for this Union operator. The current Red-Black Tree implementation of the Union operator does not use the fact that the Red-Black Tree is ordered. Therefore, there is still room for improving the Red-Black Tree.

In operator



# Elements	Linear	RBT	Hash
500	24	3	0
1000	78	7	6
1500	165	8	8
2000	296	8	7
2500	437	15	19
…			
30000	43248	178	169

Figure 5 – Performance of the In operator, times are in milliseconds

In the figure above, the performance of an In operation executed within data structures of a certain size is shown. Here we can also see that the Linear data structure is performing the worst. The performances of the Red-Black Tree and the Hash Table are very similar for this operator. This is different from what the literature claims. The use of a more efficient hashing function did not produce better results for the Hash table.

7.4. Conclusion

In this chapter, we discussed several data structures which can be used to replace the current (Linear) data structure used by Rscript. For each data structure we studied their theoretical performance for the operators Insert, Delete, Search, Max, Min and Union in the literature. On the base of this theoretical performance we concluded that the Hash Table and the Red-Black Tree are the best performing.

In order to determine the practical performances of the Hash Table and the Red-Black tree, we created a prototype which enabled us to perform measurements on these data structures. To compare these data structures we inserted a Linear data structure into this prototype. First we obtained the set operations from a Rscript file called Testspeed.rscript and inserted these operations into the prototype. Subsequently, we executed these operations for each of the data structures. Next we executed each of the operators separately for different sizes of data sets and determined which of the data structures performed the best for each operator.

We summarize the results of these measurements in the table below.

Part	Best performing data structure
Testspeed.rscript	RBT
Insert operator	RBT / Hash table
Min operator	RBT
Union operator	Hash table
In operator	RBT / Hash table

Table 11 – Best performing data structures.

In the table above we can see that the Red-Black Tree (RBT) is the best performing data structure for every measured part, except for the Union operator. However, as discussed in §7.3 the performance difference between the Hash Table and the Red-Black Tree is very small. We can see that the Linear data structure which is currently used, does not perform best in any of the tested parts. This already indicates that either one of the two tested data structures already performs better. However, we can conclude that to maximize the optimization increase of the way Rscript handles its set operations, the best data structure to use is the Red-Black Tree.

The next chapter will discuss the results obtained by this thesis.

8. Results

This chapter presents the results obtained by this research project and related them to the research questions raised in chapter 1.

8.1. Research Questions

In this paragraph we will give an answer to the main raised research questions.

I. How can Rscript comprehensions be optimized?

Rscript can be optimized in two ways: 1) algebraic transformations and 2) to use a different data structure for set representation.

1) We have discussed several algebraic optimization techniques, which were found in the relevant literature. Some of these techniques are based on techniques used for optimizing relational database queries, others are based on optimizing comprehensions in general. Each of the techniques performs one or more transformations in order to change the original comprehension into a semantically equivalent and more efficient form. The content and purpose of these transformations are different for each of the optimization techniques and have been discussed in chapter four and five.
While discussing each of the techniques, we have determined whether the techniques are suitable or not for applying to Rscript. Most of the discussed techniques are found to be applicable, such as: Qualifier Interchange, Filter Hiding, Common Subexpression Elimination, Commuting Selections, Semantic Query Caching, Product Elimination and Peephole Optimization. Other techniques are found to be not applicable, such as: Index Introduction. The reason that these techniques are not applicable lies in their use of operators which are not supported by Rscript.

2) Rscript currently uses a Linear data structure in order to store sets, and consequently the operations on this data structure are also done in linear time. By improving the calculation time necessary for these operations, the over-all time of comprehensions can be improved. Therefore we investigated several data structures which could be used by Rscript, in order to increase the performance of set operations, such as: Hash Tables, Binary Search Tree, Red-Black Trees, Binomial Heap, and Judy Arrays.

II. What are the effects of the optimizations?

We answer this question for 1) algebraic optimizations and for 2) data structures.

1) In order to determine the effect of the algebraic optimization techniques, a test case in the form of a Rscript file has been created for each of these techniques. The effect of the optimization has been measured by determining the calculation time of the test case with - and without an optimization.
As shown in table 3 (see §6.1.1) the results are diverse. Some optimization techniques are causing a substantial increase, such as Qualifier Interchange, Filter Hiding, Commuting Selections – set difference, Commuting Selections – Cartesian product and Product Elimination. Other techniques barely perform an optimization, such as: Common Subexpression Elimination, Commuting Selections – union, Semantic Query Caching and Peephole Optimization – po/2.
It was remarkable that although the literature describes Filter Hiding as a techniques which does not perform an optimization, our results show otherwise. The opposite was also found: literature describes Common Subexpression Elimination as a noticeable technique, however our results show almost no increase at all.
In order to validate these results, we have performed the two most efficiently determined optimization techniques (Qualifier Interchange and Filter Hiding) on three different Rscript files. In order to do so, we have created a prototype application which enabled us to automate the optimization process for these optimization techniques. First, we used this prototype to optimize the earlier created test cases: the results obtained were similar to the manual results. Next we used the prototype on the three Rscript files. Although a performance increase is seen, the results are disappointing: the increase does not match the increase obtained by optimizing the test cases. We concluded that the performance gain is different for each situation in which the optimizations occur.

2) In order to determine the effect of using other data structures to store sets, the relevant literature has been studied to determine which data structure has the best performance. An overview of the theoretical performance of several operations

for each of the data structures has been created in table 9 (see §7.1.6). We concluded that the data structures Hash Table and Red-Black Tree theoretically performed the best.

To determine the practical performance of these data structures, a prototype application has been developed which enabled us to measure the running time of different data structures while performing several operations. We used the operations that were extracted from an existing Rscript file and inserted them into the application. For comparison a Linear data structure has been added to the application. The results are shown in table 10 (see §7.3) and we saw that both the Hash Table and the Red-Black Tree performed better than the Linear data structure. Since these measurements only show the performance of the used Rscript file, we also determined the performance for each of the operators separately (see §7.3). This showed us that the Insert and In operator was performed equally fast by both the Hash Table and the Red-Black Tree. The Min/Max operator could be performed faster by the Red-Black Tree and the Union operator could be performed faster by the Hash Table. Finally, we concluded that to optimize the way Rscript handles its set operations, the best data structure to use is the Red-Black Tree.

8.2. Validation

To validate the results obtained by applying the algebraic optimization techniques manually to the test cases, the same test cases have been applied to the automation prototype. Since this prototype can only optimize for two techniques (Qualifier Interchange and Filter Hiding), not all of the techniques have been validated. However, the results for these two techniques after applying them to the prototype, are similar to the results obtained manually.
Another validation phase has been done by optimizing the student files with the prototype. The purpose was to check how *real* Rscript applications would react when they are optimized. Subsequently, the performance increase of these files has been determined before and after optimization. The results were disappointing. The observed performance increase of the student files did not show any similarity with the increase we observed by optimizing the test cases.

In order to validate the measurements obtained by performing operations on data structures, we made sure that when an operation is performed on a data structure, it is performed correctly. This is done by creating unit tests for each of the operators.

8.3. General Conclusion

Taken together the answers to research questions I and II, we can say that the best way to optimize the Rscript comprehensions is not to algebraically transform the comprehension into a more efficient form, but to change the underlying data structure. According to the results of this thesis, a transformation of a comprehension will not always guarantee a performance increase. Also the performance gain is unique for every situation. However, when a better performing data structure is used the performance of the entire comprehension will benefit.
The most efficient algebraic optimization techniques are Qualifier Interchange and Filter Hiding. These techniques also have the biggest possibility to be applicable. Theoretically the data structure Hash Table is the best performing. However, our own measurements showed that the data structure which operators are performing best is a Red-Black Tree.

8.4. Future Work

Although a clear answer has been given to our main research questions, a number of points have not been addressed yet and remain to be studied in future work. We will briefly summarize these points here.

Automation Prototype

- The automation prototype in this thesis is created only to automate the optimization of the test cases and is not very well suited to optimize other Rscript files. Someone with enough experience of ASF+SDF should use the optimization techniques and pseudocode algorithms which are discussed in this thesis, in order to change the ASF+SDF implementation of Rscript to automatically apply these optimizations. Although the best performance increase is not realized by algebraically optimizing a comprehensions, the performance does improve.

Questions raised by our results

- To fully determine which technique is considered the most efficient, we considered three criteria (see § 6.1). However, a fourth criterium should be considered also. A general concern for every optimization technique is that the optimization phase is not free: it takes time to perform the necessary (transformation) steps. The fourth criterium should be looking at the time that is needed for this optimization phase. Although this criterium is not

used here because the optimizations are done manually, it actually should be used in order to determine which optimization technique can be considered the most efficient. However, we do not suspect that by using this fourth criterium that the results will be altered significantly.

- We have measured the performance increase of all of the discussed optimization techniques (see § 6.1.1). Also the optimization techniques Qualifier Interchange and Filter hiding have been tested together. It showed us that their combined result does not perform better than each of the techniques does separately. A question which is yet unanswered is whether this is caused by using a bad test case, or that the techniques really do not perform better when they are combined?
- When validating the results obtained by optimizing the test cases, we used the automation prototype to optimize several existing Rscript files: the student files (see § 6.4). However, the optimization of the student files did not realized the a performance increase similar to what we observer with the test cases. We have been unable to answer the question whether this is because of bad Rscript files or that is this caused by other factors?

Data Structure

- We have determined which data structure has to be used in order to achieve the largest performance increase. With this information, Rscript can be adapted in order to use this data structure. When this has been realized, there has to be determined if the different data structure really does performs better.

9. Evaluation

In this chapter will be discussed how the project went and if the results were satisfying. Also will be discussed how this project could have been performed in a better way.

Results

We have identified several algebraic optimization techniques which can be used to optimize Rscript comprehensions (see chapter 4 and 5). Each of the techniques has been discussed and for those which are found to be applicable to Rscript, a test case has been created. The results of these test cases are, to a large extent, in agreement to what we expected. Some techniques were expected to be performing better, such as Common Subexpression Elimination. Other techniques performed better than we thought at first, such as Qualifier Interchange.
Several student files have been scanned in order to check how often optimization techniques could be applied. This is shown in table 4 and 5 (see §6.1.2). The applicability was rather disappointing. Only three optimization techniques were found to be applicable: Qualifier Interchange, Filter Hiding and Semantic Query Caching. We would have expected that more optimization techniques would be found to be applicable.
We also identified several data structures which can be used to replace the current data structure for set representation (see chapter 7). After measuring the performance of several important operators (insert, min/max, union and in) for each of the data structures, we concluded that overall the use of Red-Black Tree will give the best performance (see 7.4).
Overall we concluded that the best way to optimize Rscript comprehensions is not to apply algebraic optimization techniques, but to change Rscript to use another data structure.

What went wrong/Problems encountered

Nothing really went 'wrong', although we did expect to find more algebraic optimization techniques which could be studied. Also, it took too much time to narrow down the exact target of this project. This should have been done earlier, in the literature study, in which case we would have more time during the project which we could spend on realizing the targets raised.

Usability

The results of this thesis can be used by the CWI in several ways. The automation prototype can be used to optimize Rscript files, although it probably has to be extended with a parser first in order to be useful. When CWI chooses not to continue with this prototype, the identified optimization techniques can be implemented in the current ASF+SDF implementation of Rscript. In this case the produced pseudo code algorithms can be reused.
The data structure prototype can be used in the future to compare other data structures. And the data structure which showed, according to our measurements, the largest increase in performance can be used with Rscript.

The project in retrospect

When starting the project, it became apparent that the path that we chose beforehand, to fully create a Java version of Rscript which could apply optimization techniques automatically, could not be realized. Therefore the projects has been slightly altered several times and finally converted into the current form.

When looking back at the entire project several things could have been done better. We mention a few:

- The data structures described in the chapter of Set Optimization should have been investigated earlier in the project. Since changing the data structure can optimize Rscript completely, it is very interesting to study this in more detail. We would have liked to see this optimization technique implemented, however the time available was not enough.
- The automation prototype has been developed in Java. It would probably have been more useful to CWI if it was implemented in ASF+SDF. Although our experience in this language is fairly limited and it would have taken us longer to implement it, this would have been a better choice.

Bibliography

[1] Trinder, P., Wadler, P. *Improving List Comprehension Database Queries*. Proceedings of TEN-CON'89, Bombay, India (November 1989), pages 186-192.

[2] Trinder, P. *Comprehensions, a Query Notation for DBPLs*. Glasgow University, Department of Computing Science.

[3] Klint, P. *A Tutorial Introduction to Rscript – a Relational Approach to Software Analysis* -. May 2005.

[4] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D. *System R: Relational Approach to Database Management.* IBM Research Laboratory. ACM Transactions on Database Systems, Vol 1, No. 2, June 1976, pages 97-137.

[5] Poulovassilis, A., Small, C. *Algebraic query optimisation for database programming languages.* The VLDB Journal, 1996.

[6] Ullman, J. D. *Database and Knowledge-base Systems*, Volume II: The New Technologies. Computer Science Press, 1989.

[7] van den Brand, M., Klint, P. *ASF+SDF Meta-Environment: Guided Tour.* Revision 1.17. CWI, 2005.

[8] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, M.Tan. *Semantic Data Caching and Replacement*. In proceedings of the 22nd International Conference on Very Large Data Bases(VLDB), Bombay, India, Sept. 1996. To appear. URL: http://www.cs.umd.edu/projects/dimsum/papers/semantic_caching.ps.gz

[9] Godfrey, P., Gryz, J. *Intensional Query Optimization*. Dept. of Computer Science, University of Maryland. Sept. 1996

[10] Godfrey, P., Gryz, J. *Semantic Query Caching for Heterogeneous Databases*. Proceedings of the 4th KRDB Workshop, Athens. August 1997.

[11] Chen, F. F., Dunham, M.H. *Common Subexpression Elimination Processing in Multiple-Query Processing*. IEEE Transactions on knowledge and data engineering, Vol. 10, No 3. June 1998.

[12] McKeeman, W. M. *Peephole Optimization*. Stanford University, California. 1965.

[13] Freudenberger, S.M., Schwartz, J.T., Sharir, M. *Experience with the SETL Optimizer*. New York. 1983.

[14] Lewis, J., Chase, J. *Java Software Structures – designing & using data structures, second edition.* Addison Wesley 2005.

[15] Cormen, T.H., Leiserson C.E., Rivest, R.L. *Introduction to Algorithms*. The MIT Press, fifth printing, 1991.

[16] Doug B., Hewlett-Packard. *Judy Arrays web page.* Website, http://judy.sourceforge.net/

[17] Sean B. *A Performance Comparison of Judy to Hash Tables*. Website, http://www.nothings.org/computer/judy/ August 2003

[18] Alan S. *Judy IV Shop Manual*. An internal technical description of Judy. Hewlett-Packard, August 2002.

Appendix A – Test Cases

Here all of the testcases are documented. Since the relations AB and CD and the set N and M are used in multiple test scripts, they are not repeated every time but only documented once.

```
rel[int,int] AB = { <1,2> , T2, ... T1000 }        where T = Tuple<int,int>
rel[int,int] CD = { <5,6> , T2, ... T1000 }
```

```
set[int] N = { 1 , I1 , ... I200 }                 where I = int
set[int] N = { 5 , I1 , ... I200 }
```

Qualifier Interchange – not optimized

```
set[int] result = { A | <int A, int B> : AB, <int C, int D> : CD, B == C, D == 23
}
```

Qualifier Interchange – optimized manually

```
set[int] result = { A | <int C, int D> : CD, D == 23, <int A, int B> : AB , B == C
}
```

Qualifier Interchange – optimized automatically

```
yield result
set[ int ] result = { A | < int C , int D > : CD , D == 23 , < int A , int B > :
AB , B == C }
```

Filter Hiding – not optimized

```
set[int] result = { A | <int A, int B> : AB, <int C, int D> : CD, B == C , D == 23
}
```

Filter Hiding – optimized manually

```
set[int] CD23 = { C | <int C, int D> : CD, D == 23 }
set[int] result = { A | <int A, int B> : AB , int C : CD23, B == C }
```

Filter Hiding – optimized automatically

```
yield result
rel[ int , int ] FH1 = { < C , D > | < int C , int D > : CD , D == 23 }
set[ int ] result = { A | < int A , int B > : AB , < int C , int D > : FH1 , B ==
C }
```

Qualifier Interchange and Filter Hiding – not optimized

```
set[int] result = { A | <int A, int B> : AB, <int C, int D> : CD, B == C, D == 23
}
```

Qualifier Interchange and Filter Hiding – optimized manually

```
set[int] CD23 = { C | <int C, int D> : CD, D == 23 }
set[int] result = { A | int C : CD23, <int A, int B> : AB, B == C }
```

Qualifier Interchange and Filter Hiding – optimized automatically

```
yield result
rel[ int , int ] FH1 = { < C , D > | < int C , int D > : CD , D == 23 }
set[ int ] result = { A | < int A , int B > : AB , < int C , int D > : FH1 , B ==
C }
```

Common Subexpression Elimination – not optimized

```
set[int] selD-one = { D | <int C, int D> : CD, D == 23 }
set[int] resOne = { A | int C : selD-one, <int A, int B> : AB , B == C }

set[int] selD-two = { D | <int C, int D> : CD, D == 23 }
set[int] resTwo = { A | int C : selD-two, <int A, int B> : AB , B != C }
```

Common Subexpression Elimination – optimized

```
set[int] selD = { D | <int C, int D> : CD, D == 23 }
set[int] resOne = { A | int C : selD, <int A, int B> : AB , B == C }

set[int] resTwo = { A | int C : selD, <int A, int B> : AB , B != C }
```

Commuting Selections – set-difference – not optimized

```
set[int] result = { result | int result : ( AB \ CD )[-,3] }
```

Commuting Selections – set-difference – optimized

```
set[int] result = { result | int result : ( AB[-,3] \ CD[-,3] ) }
```

Commuting Selections – union – not optimized

```
set[int] result = { r | int r : ( AB union CD )[-,3] }
```

Commuting Selections – union – optimized

```
set[int] result = { r | int r : AB[-,3] union CD[-,3] }
```

Commuting Selections – Cartesian Product – not optimized

```
rel[int,int] result = { <n,m> | <int n, int m> : N x M , m > 28}
```

Commuting Selections – Cartesian Product – optimized

```
rel[int,int] result = { <n,m> | set[int] M28 <- { m | int m : M , m > 28 }, <int
n, int m> : N x M28 }
```

Semantic Query Caching – subsumption – not optimized

```
set[int] CD25 = { C | <int C, int D> : CD, D <= 25 }
set[int] oldResult = { A | <int A, int B> : AB , int C : CD25, B == C }

set[int] CD15 = { C | <int C, int D> : CD, D <= 15 }
set[int] result = { A | <int A, int B> : AB , int C : CD15, B == C }
```

Semantic Query Caching – subsumption – optimized

```
set[int] CD25 = { D | <int C, int D> : CD, D <= 25 }
set[int] CD15 = { D | int D : CD25, D <= 15 }

set[int] oldResult = { A | <int A, int B> : AB , int D : CD25, B == D }
set[int] result = { A | <int A, int B> : AB , int D : CD15, B == D }
```

Semantic Query Caching – overlap – not optimized

```
set[int] CD25 = { C | <int C, int D> : CD, D <= 25 }
set[int] oldResult = { A | <int A, int B> : AB , int C : CD25, B == C }

set[int] CD15 = { C | <int C, int D> : CD, D > 15 }
set[int] result = { A | <int A, int B> : AB , int C : CD15, B == C }
```

Semantic Query Caching – overlap – optimized

```
set[int] CD25 = { D | <int C, int D> : CD, D <= 25 }
set[int] CD15 = { D,D2 | int D : CD25, D > 15, <int C, int D2> : CD, D2 > 25 }

set[int] oldResult = { A | <int A, int B> : AB , int D : CD25, B == D }
set[int] result = { A | <int A, int B> : AB , int D : CD15, B == D }
```

Peephole Optimization – po/1 – not optimized

```
int result = 5 * 10 * 20 * 30 * 40 * 50
```

Peephole Optimization – po/1 – optimized

```
int result = 60000000
```

Peephole Optimization – po/2 – not optimized

```
set[int] result = { A | <int A, int B> : AB , <int C, int D> : CD, B == C, D == 23
}
```

Peephole Optimization – po/2 – optimized

```
set[int] result = { A | <int A, int B> : { <1,2> , T2, ... T1000 } , <int C, int D>
: { <5,6> , T2, ... T1000 }, B == C, D == 23 }
```

Appendix B – Student Files

Here the student files are documented, both the not optimized file and the file which is produced by the prototype.

Question 5 – not optimized

```
yield Q5

rel[str,str] calls = { <"R", "A">, <"R", "B">, <"A", "D">, <"B", "A">, <"B", "D">,
<"B", "E">, <"C", "F">, <"C", "G">, <"D", "L">, <"E", "H">, <"F", "I">, <"G",
"I">, <"G", "J">, <"H", "K">, <"H", "E">, <"I", "K">, <"K", "I">, <"L", "H"> }

set[str] Q2 = top(calls)
rel[str, str] Q4 = calls+


set[str] indirectCalls(str X, rel[str,str] R1, rel[str,str] R2) = { Z | <str Y,
str Z> : R1, <Y, Z> notin R2, X == Y }

rel[str,set[str]] Q5 = {<X, indirectCalls(X, Q4, calls)> | str X : Q2 }
```

Question 5 – optimized automatically

```
yield Q5

rel[str,str] calls = { <"R", "A">, <"R", "B">, <"A", "D">, <"B", "A">, <"B", "D">,
<"B", "E">, <"C", "F">, <"C", "G">, <"D", "L">, <"E", "H">, <"F", "I">, <"G",
"I">, <"G", "J">, <"H", "K">, <"H", "E">, <"I", "K">, <"K", "I">, <"L", "H"> }

set[str] Q2 = top(calls)
rel[str, str] Q4 = calls+


set[str] indirectCalls(str X, rel[str,str] R1, rel[str,str] R2) = { Z | <str Y,
str Z> : R1, X == Y, <Y, Z> notin R2 }

rel[str,set[str]] Q5 = {<X, indirectCalls(X, Q4, calls)> | str X : Q2 }
```

Question 11 – not optimized

```
yield DegenerateIn

rel[str,str] Closure = INHERITANCE+
rel[str,set[str]] NonTransitive = {<PARENT, INHERITANCE[PARENT]> | str PARENT :
domain(INHERITANCE)}
rel[str,set[str]] Transitive = {<PARENT, Closure[PARENT]> | str PARENT :
domain(Closure)}

rel[str,str, set[str]] DegenerateIn =
{<PARENT,CHILD,SETOFPARENT inter SETOFCHILD> |
str PARENT : domain(NonTransitive),
str CHILD : domain(Transitive),
set[str] SETOFPARENT : NonTransitive[PARENT],
set[str] SETOFCHILD : Transitive[CHILD],
CHILD in SETOFPARENT,
PARENT != CHILD,
SETOFPARENT inter SETOFCHILD != { } }
```

Question 11 – optimized automatically

```
yield DegenerateIn

rel[str,str] INHERITANCE

rel[str,str] Closure = INHERITANCE+
rel[str,set[str]] NonTransitive = {<PARENT, INHERITANCE[PARENT]> | str PARENT :
domain(INHERITANCE)}
rel[str,set[str]] Transitive = {<PARENT, Closure[PARENT]> | str PARENT :
domain(Closure)}

rel[str,str, set[str]] DegenerateIn =
{<PARENT,CHILD,SETOFPARENT inter SETOFCHILD> |
str PARENT : domain(NonTransitive),
str CHILD : domain(Transitive),
PARENT != CHILD,
set[str] SETOFPARENT : NonTransitive[PARENT],
CHILD in SETOFPARENT,
set[str] SETOFCHILD : Transitive[CHILD],
SETOFPARENT inter SETOFCHILD != { } }
```

Question 12 – not optimized

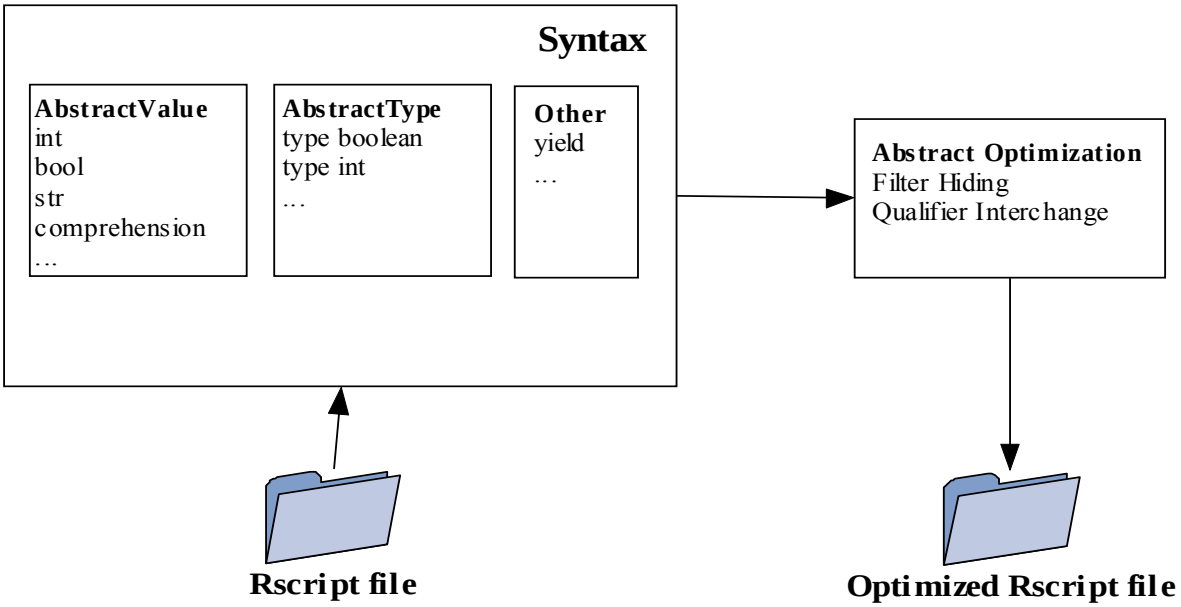
```
Yield Clones

rel[str,set[str]] InherSet = {<PARENT, INHERITANCE[PARENT]> | str PARENT :
domain(INHERITANCE)}

rel[str, str] InherGelijk = {<X,Y> |
str X : domain(InherSet),
str Y : domain(InherSet),
set[str] TEMPSETX : InherSet[X],
set[str] TEMPSETY : InherSet[Y],
X != Y,
TEMPSETX == TEMPSETY}

rel[str,set[str]] ContainSet = {<PARENT, CONTAINMENT[PARENT]> | str PARENT :
domain(CONTAINMENT), PARENT in domain(InherGelijk)}

rel[str,str] Clones = {<X,Y> |
str X : domain(ContainSet),
str Y : domain(ContainSet),
set[str] TEMPSETX : ContainSet[X],
set[str] TEMPSETY : ContainSet[Y],
X != Y,
TEMPSETX == TEMPSETY}
```

Question 12 – optimized automatically

```
yield Clones

rel[str,set[str]] InherSet = {<PARENT, INHERITANCE[PARENT]> | str PARENT :
domain(INHERITANCE)}

rel[str, str] InherGelijk = {<X,Y> |
str X : domain(InherSet),
str Y : domain(InherSet),
X != Y,
set[str] TEMPSETX : InherSet[X],
set[str] TEMPSETY : InherSet[Y],
TEMPSETX == TEMPSETY}

rel[str,set[str]] ContainSet = {<PARENT, CONTAINMENT[PARENT]> | str PARENT :
domain(CONTAINMENT), PARENT in domain(InherGelijk)}

rel[str,str] Clones = {<X,Y> |
str X : domain(ContainSet),
str Y : domain(ContainSet),
X != Y,
set[str] TEMPSETX : ContainSet[X],
set[str] TEMPSETY : ContainSet[Y],
TEMPSETX == TEMPSETY}
```

Appendix C – Automation Prototype Architecture

Here the architecture of the prototype used for automatic application of optimizations is given.



The architecture of the prototype consists of four separate areas with each their own properties. Each of the areas will be discussed here:

The Syntax area consists of the language definition of Rscript, represented in Java. It consists of three main types: AbstractValue, AbstractType and Other.
Within AbstractValue all of the different values that exist in Rscript are placed. Examples are: `int number = 5`, `str name= "Jaap"` but also comprehensions, transitive close operators, Cartesian product, etc.
Situated within AbstractType are all the different types that Rscript contains. Examples are: `bool`, `int`, `set`, `rel`, etc.
In Other is the left over functionality of Rscript situated, such as `yield`.
The following syntax is implemented into the prototype:

AbstractValue	***AbstractType***	***Not implemented***
Roundbrackets, bool, int, str, tuple, set, rel, carrier exclusion, domain exclusion, range exclusion, reach exclusion, carrier restriction, domain restriction, range restriction, reach restriction, complement, identity, inverse, powerset0, powerset1, carrier, domain, range, bottom, top, filename, first, second, divide, min, multiply, plus, and, implies, or, equal, greater, greater equal, not equal, smaller, smaller equal, diff, inter, union, in, not in, cartesian product, composition, transitive closure, reflexitive transitive closure, not, nr of elements	Boolean, int, rel, set, string, tuple, user types.	Equations, Asserts, locations

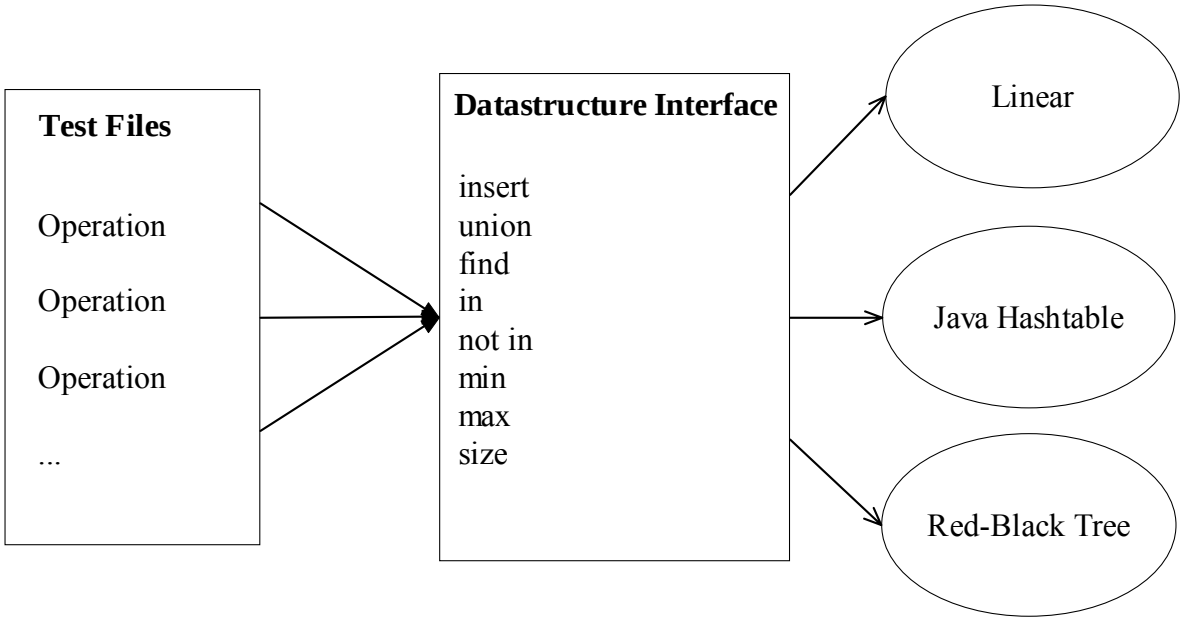
The Rscript file area is basically the Rscript file that has to be optimized. It has to be expressed manually in the syntax described above.

The (Abstract)Optimization area contains all the optimizations. Currently there are two optimizations implemented, but it can be expanded in order to support more optimization techniques. Each of the optimizations uses the Java version of the syntax definition of Rscript and performs its transformations to change the Rscript itself.

The Optimized Rscript file area shows the optimized Rscript file.

Appendix D – Data Structure Prototype

Here the architecture of the prototype used for measuring different data structures is given.



The above architecture shows three different sections. On the left there is a collection of operations within a test file. These operations have to be extracted manually from an existing Rscript file. In the center, the data structure interface is shown. This interface identifies the operations which are possible to perform on each of the data structures which are situated on the right. Each of these data structures uses the data structure interface.

Appendix E – TestSpeed.rscript

```
rel[str,str] CALL = { 656 tuples }
bag [int] B1 = { 48 different integers, repeated 5 times }

bag[int] B2 = {50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,34,33,32,31,30,
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,42,43,44,45,46,47,48,49,50}

bag[int] do(bag[int] P1, bag[int] P2)  =
       { X |
         bag[int] X1 <- P1 union P2,
         bag[int] X2 <- P1 inter P2,
         bag[int] X3 <- P1 \ P2,
         P1 >= P2,
           43 in P2,
           120 in P1,
         <"AbstractFigure_2788", "Geom_3544"> in CALL,
           int X <- 1
       }

bag [int] nrange =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,
31,32,33,34,35,36,37,38,39,40}

rel[str,str] clos = CALL+

bag[int] work = {N | int N : nrange , int M : nrange, bag[int] Z <- do(B1, B2)}
```