



Technisch-Naturwissenschaftliche
Fakultät

Efficient Rewriting by Object Reuse and Compiling

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Computermathematik

Eingereicht von:

Alexander Maletzky

Angefertigt am:

Research Institute for Symbolic Computation - RISC

Beurteilung:

A.Univ.-Prof. Dr. Tudor Jebelean

Linz, Mai 2013

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, im Mai 2013

Abstract

During the process of rewriting expressions (terms/formulae), subexpressions have to be replaced by other expressions all the time. Therefore it is of particular interest to make these subsequent replacements as efficient as possible in order to speed-up the entire process. This master's thesis pursues a new approach that was raised by the thesis supervisor for achieving precisely the mentioned increase in efficiency in connection with “replacing expressions by expressions” by working out all the subtle details that have to be taken into account and by implementing various programs (one interpreter, two compilers) that make use of the new approach.

The main idea is the following: Assume you are given an expression, represented as a tree, and a rewrite rule that is applicable to the expression/tree. Then, traditionally, when applying the rule the entire right-hand-side of the rule is created (with the variables instantiated properly), and the entire old expression is deleted. Although in principle this works pretty fine, it can be made much more efficient: Instead of creating the *entire* right-hand-side and deleting the *entire* expression, simply *reuse* all the objects that are already available in the expression and that are needed in the right-hand-side of the rule by moving them to new positions. This idea, as already pointed out above, is due to the thesis supervisor, Prof. Tudor Jebelean, and as it turns out, the new approach is indeed much more efficient w. r. t. time, as lots of timing experiments show. These experiments, which of course are presented in this document, are mainly built upon small to medium-size sets of rewrite-rules in the domains of adding/multiplying integers/polynomials and proving/disproving propositional formulae. However, not only is the old, constructive way of rewriting compared to the new, destructive one, but the different programs that have been implemented are compared to each other as well.

The implementation of the programs was done in the programming language C++, where heavy use was made of pointers, inheritance and type polymorphism. Clearly, this document does not only cover all important and subtle aspects of the algorithms and the implementation, but also contains detailed information on how the programs can be used. In fact, there are three different programs, namely an interpreter (Version 1) and two compilers (Versions 2 and 3), each capable of solving “real” rewriting tasks, such as adding Integers or proving/disproving formulae in propositional logic. The interpreter reads a set of rewrite rules and is immediately ready to solve rewriting tasks, without creating intermediate output that has to be compiled first by external compilers. This is in contrast to the other two programs, which first create intermediate C++ source- and header files that implement precisely the program that is capable of solving the rewriting tasks in the domain of the given rule set. Here, Version 3 does exactly the same as Version 2, but unlike Version 2 it can also deal with so-called *sequence variables*.

Zusammenfassung

Wird ein Ausdruck (beispielsweise ein Term oder eine Formel) im Zuge eines Rewriting-Prozesses transformiert, so geschieht dies durch sukzessives Ersetzen von Teilausdrücken durch andere Ausdrücke. Deshalb ist es von Interesse, das Ersetzen von Ausdrücken möglichst effizient zu gestalten, um den gesamten Rewriting-Prozess zu beschleunigen. Die vorliegende Arbeit verfolgt einen neuen Ansatz, der vom Betreuer der Arbeit stammt, und dessen Ziel es ist, genau jene Effizienzsteigerung in Verbin- gung mit dem Ersetzen von Ausdrücken zu erreichen. Dabei wurden nicht nur sämtliche subtilen Details, die berücksichtigt werden müssen, ausgearbeitet, sondern auch verschiedene Computerprogramme (einen Interpreter, zwei Compiler), die auf dem neuen Ansatz aufbauen, implementiert.

Diese Idee lautet wie folgt: Gegeben seien ein (syntaktischer) Ausdruck, repräsentiert durch eine Baum-Struktur, und eine Rewrite-Regel, die darauf angewendet werden kann. Wenn die Regel angewendet wird, so wird normalerweise ihre komplette rechte Seite neu erzeugt (wobei die Variablen natürlich entsprechend instanziiert werden), und der komplette “alte” Ausdruck wird gelöscht. Obwohl diese Vorgehensweise im Prinzip gut funktioniert, kann ihre Effizienz doch grundlegend gesteigert werden: Anstatt die *komplette* rechte Seite zu erzeugen und den *kompletten* Ausdruck zu löschen, verwende alle Objekte, die bereits im Ausdruck verfügbar sind und in der rechten Seite der Regel benötigt werden, indem sie einfach an neue Positionen gesetzt werden. Wie sich herausstellt ist dieser Ansatz, der vom Betreuer dieser Arbeit, Prof. Tudor Jebelean, stammt, tatsächlich viel effizienter was die Rechenzeit anbelangt, wie einige Experimente zeigen. Diese Experimente, die selbstverständlich im vorliegenden Dokument präsentiert werden, basieren hauptsächlich auf kleinen bis mittelgroßen Mengen von Rewrite-Regeln in den Bereichen Addition/Multiplikation von ganzen Zahlen/Polynomen und Beweisen von aussagenlogischen Formeln. Allerdings werden dabei nicht nur die alte, konstruktive Methode mit der neuen, destruktiven Methode verglichen, sondern auch die verschiedenen Programme untereinander.

Die Implementierung dieser Programme erfolgte in der Programmiersprache C++, wo starker Gebrauch von Zeigern, Vererbung und Typ-Polymorphismus gemacht wurde. Klarerweise behandelt dieses Dokument nicht nur alle wichtigen Aspekte der Algorithmen und der Implementierung, sondern gibt auch Aufschluss darüber, wie die Programme genutzt werden können. Tatsächlich gibt es drei verschiedene Programme, nämlich einen Interpreter und zwei Compiler, wobei jedes von ihnen in der Lage ist, “echte” Rewriting-Aufgaben zu lösen. Der Unterschied zwischen dem Interpreter und den beiden Compilern liegt offensichtlich darin, dass letztere Mengen von Rewrite-Regeln zuerst in ausführbare Programme umwandeln, wohingegen ersterer sofort mit den gegebenen Regeln arbeiten kann.

Acknowledgements

First of all, I want to thank Prof. Tudor Jebelean for giving me the opportunity to write my master's thesis under his guidance. It was him who raised my interest in mathematical logic and automated proving, as well as rewriting, and I also want to mention that he came up with many good ideas that helped me a lot in improving this thesis (not to forget the basic new idea this thesis is all about). Thank you very much!

I am also grateful for participating in the *Theorema* seminar of the Research Institute for Symbolic Computation: Thanks for many interesting talks and presentations to all members of the *Theorema* group! In particular, thanks to Prof. Bruno Buchberger, who allowed me to continue my life at the university by accepting me as his new PhD student.

And last but not least I am of course also very thankful for my family and their support, day in, day out. It is hard to express how much I am indebted to them.

Contents

1	Introduction	10
1.1	Maude	12
2	Basic Definitions	14
2.1	Formal Definition of Rewriting	14
2.2	Expressions	16
2.2.1	Subexpressions	18
2.2.2	Position of Objects	18
2.3	Integers and Natural Numbers	19
2.4	Rewrite Rules	21
2.5	Rule Sets	22
2.6	Sequence Variables	23
2.7	Trees	24
2.8	Mappings	26
2.8.1	Type- and Arity Preserving Mappings	27
2.8.2	The Notation of Mappings	29
2.8.3	The Number of Mappings	30
3	General Implementation Documentation	32
3.1	Symbols	32
3.2	Implementation of Trees	33
3.2.1	Class <code>Expression</code>	34
3.2.2	Class <code>Constant</code>	34
3.2.3	Class <code>Function</code>	34
3.2.4	Class <code>Variable</code>	34
3.3	Implementation of Rules	35
3.4	Parsing Expressions and Rules	35
3.5	Rule-Application Strategy	37
4	Version 1: The Interpreter	40

4.1	Algorithms and Implementation	40
4.1.1	Implementation Details	41
4.1.2	Loading and Storing a Rule	42
4.1.3	The Matching Algorithm	44
4.1.4	The Rewriting Algorithm	45
4.2	How To Use “Interpreter”	47
4.2.1	General Overview	48
4.2.2	Input Files	48
4.3	An Example	49
4.3.1	Step 1: Creating References	50
4.3.2	Step 2: Copying the <code>rhs</code> -Tree	51
4.3.3	Step 3: Deleting Unused Vertices	54
5	Version 2: The Compiler	55
5.1	Implementation of the Compiled Programs	56
5.2	Algorithms and Implementation of the Compiler	57
5.2.1	Class <code>Rule</code>	58
5.2.2	Constructing Function <code>match</code>	59
5.2.3	Commands	60
5.2.4	Constructing Function <code>rewrite</code>	64
5.3	How To Use “Compiler”	72
5.4	An Example	73
6	Version 3: The Compiler with Sequence Variables	82
6.1	Implementation of the Compiled Programs	83
6.2	Algorithms and Implementation of the Compiler	84
6.2.1	Commands	85
6.2.2	Constructing Function <code>match</code>	94
6.2.3	Constructing Function <code>rewrite</code>	103
6.3	How To Use “Compiler/SV”	108
6.4	An Example	109
7	Tests And Timing Experiments	115
7.1	Comparing the Different Versions	116
7.1.1	General Parameter Settings	116
7.1.2	Rule Set <code>Addition</code>	117
7.1.3	Rule Set <code>Multiplication</code>	117
7.1.4	Rule Set <code>NatAddition</code>	118
7.1.5	Rule Set <code>NatMultiplication</code>	119
7.1.6	Rule Set <code>NatSum</code>	119

7.1.7	Rule Set <code>PolyAddition</code>	120
7.1.8	Rule Set <code>SequentCalculus</code>	121
7.1.9	Summary	121
7.2	Comparing Old (Constructive) and New (Destructive) Concept	122
7.2.1	General Parameter Settings	122
7.2.2	Rule 1	124
7.2.3	Rule 2	124
7.2.4	Rule 3	125
7.2.5	Rule 4	125
7.2.6	Rule 5	126
7.2.7	Summary	126
7.3	Comparing “Compiler/SV” to <i>Mathematica</i>	127
8	Conclusion	129
8.1	Complexity Analysis	130
8.1.1	Arity Preserving	131
8.1.2	Type Preserving	132
8.1.3	Complexity: Conclusion	132
8.2	Future Work	133
8.2.1	Program Features	133
8.2.2	Implementation	134
A	Special commands	136
A.1	Closing the Program	136
A.2	Maximum Number of Rewriting Steps	136
A.3	Rewriting the previous Result	137
A.4	Integers and Natural Numbers	137
A.5	Tracing the Rewriting Process	137
A.6	Rewriting Strategy	138
A.7	Rewriting an Input File	138
A.8	Rewriting Algorithm*	138
A.9	Performing Timing Experiments	139
A.10	Printing all Symbols	140
A.11	Printing all Constants*	140
A.12	Printing all Rule Sets*	140
A.13	Printing all Rules*	140
A.14	Printing a single Rule*	141
B	Rule Sets	142

B.1	Addition	142
B.2	Multiplication	143
B.3	NatAddition	145
B.4	NatMultiplication	145
B.5	NatSum	145
B.6	PolyAddition	145
B.7	Sorting	146
B.8	SequentCalculus	146
B.9	SC	148

Chapter 1

Introduction

Generally speaking, *rewriting* is the art of transforming objects of any kind into other objects. Those objects could be, for instance, states of systems, logical formulae, mathematical expressions, or anything else. The way they are transformed is by applying so-called *rewrite rules*, which are rules that specify how an object of a certain class may be transformed into another object. For example, there could be a rule that specifies that every mathematical expression (object) of the form $(a + b)^2$, where a and b are arbitrary expressions, may be transformed into $a^2 + 2ab + b^2$. The emphasis is on the word *may*: If such a rule exists, then it is in general not guaranteed that it really is applied; There might be another rule which is applicable, too. A much more formal definition of rewriting can be found in Section 2.1. Please note that throughout the whole thesis the objects that are taken into account as in- and outputs of a rewriting process are arbitrary *expressions*, formally defined in Section 2.2.

From the above, intuitive definition it should be quite clear that a rewriting process, i. e. the successive application of rewrite rules, can not only be carried out by humans, but also completely automatically by computer systems, at least if some basic conditions are met (which is the case when rewriting expressions). A *rewriting system* is a program that does exactly this job. More formally, it is a method that takes as an input the object O_{in} that should be rewritten, together with a set of rewrite rules, and produces again an object O_{out} as the output. O_{out} is related to O_{in} in the following way: It can be obtained by successively applying rewrite rules to O_{in} , and there is no rule that can be applied to O_{out} any more (i. e. O_{out} is in *normal form*). This, however, leads to some problems: First of all, O_{out} is in general not unique. As already pointed out, at each stage, there might be several rules that could be applied, and depending on the one which is eventually chosen, the final results may differ. Secondly, it is not guaranteed that the method

terminates; A normal form does not have to exist in general. See, for instance, [15, 4] for more information on uniqueness/termination.

Let us now restrict ourselves to the case when it is *expressions* that are rewritten. As will be made clear in Sections 2.2 and 2.7, expressions can be represented as *trees*. Trees are, in principle, directed, acyclic graphs, consisting of *vertices* and *edges*. Vertices can be used for representing the symbols occurring in the expression, and edges for arranging the vertices in the right way. Now, whenever a rewrite rule is applied, it is not applied to the expression itself, but rather to the corresponding tree, and the result is again another tree. But how is this achieved? Traditionally, the new tree is entirely created from new vertices and the old tree is entirely deleted. This, of course, works pretty fine, but in some cases is inefficient: If the new tree shares some of its vertices with the old one, then why not just use them instead of creating new ones? This approach would save some creations/deletions and hence speed-up the rewriting process, at least in theory. This idea is due to my supervisor, Prof. Jebelean, and to work it out in detail and also to implement it was the main task of this thesis.

The goal of my master's thesis was the implementation of various rewrite systems, preferably in the programming language C++, that make use of exactly the idea described above. The result are three different programs, "Interpreter" (Version 1), "Compiler" (Version 2) and "Compiler/SV" (Version 3), each incorporating the main idea of reusing existing objects and each usable for "real" rewriting tasks. The interpreter reads a given rule set, stores the rules in some representation, and then is immediately ready to start rewriting. In contrast to that, Versions 2 and 3 are *compilers* that transform rule sets into executable programs. Therefore, for each rule set there is an own program associated to it which is only suitable for solving rewriting tasks for only that rule set. The difference between "Compiler" and "Compiler/SV" is the capability of the latter one to treat so-called *sequence variables*, variables that can be instantiated with an arbitrary sequence of expressions, see Section 2.6. All of the programs have been tested thoroughly and seem to work fine.

The structure of this document is the following: The last part of this chapter is dedicated to the rewriting-system Maude, meaning that some general information on Maude is given and comparisons to the programs implemented in the frame of this thesis are made. The next chapter introduces some basic notions and basic definitions used throughout the document, like the already mentioned expressions. Chapter 3 gives some short, general description of the implementation of the programs; more precisely of the part of the implementation that is shared by all three versions. This description is continued in far more detail in Chapters 4.1, 5 and 6 where the focus lies on the interpreter (Version 1), the compiler (Version 2) and the compiler with sequence variables (Version 3), respectively. Each of these three

chapters also includes a short description of how the corresponding program can actually be used. Chapter 7 presents some results from timing experiments that have been performed in order to compare the different versions to each other as well as to compare the new approach to the traditional one. The last chapter draws some conclusions and also contains a short theoretical complexity analysis. The appendix contains all special commands that can be used when running the programs, as well as all the rule sets that are mentioned in this thesis (e.g. in connection with timing experiments), together with a short description.

1.1 Maude

All information about the rewriting-system Maude is taken from [2], and in particular from [7].

Maude is a high-end rewriting-system that can be used in many areas of computer science and mathematics, ranging from applications in logic over program verification and model checking. As every rewriting-system, its core part consists of rules (and equations), similar to the ones presented in this document, that can be applied to objects in order to transform them into new objects (compare Section 2.1 for a formal definition of rewriting). Hence, there are similarities between Maude and the programs that have been developed in the frame of this thesis, and that is exactly the reason of this section: Comparing, in terms of functionality, the rewriting-systems of this thesis to the established, high-end rewriting-system Maude.

In Maude, every rewrite-theory is given by a so-called *module*. Hence, modules can somehow be seen as the Maude variant of the rule sets that have to be specified in connection with the programs of this thesis. However, Maude distinguishes two kinds of modules: Functional modules and system modules. The reason is the following: In Maude there are two different types of rewrite rules. On the one hand, there are *equations* that are used to define rewrite rules that transform objects into “equal” objects, like “1+2” into “3”. On the other hand, there are *rules* that are used to model state transitions, i.e. transformations where the resulting object is not necessarily “equal” to the old one. Of course, “equal” has no predefined meaning, but rather has to be defined in some way. Functional modules may only contain equations, whereas system modules may contain both equations and rules. This is the first difference between Maude and the programs of this thesis: Here, no distinction is made between rewrite rules that preserve equality and the ones that do not.

A second, important difference results from rewrite rules, too: In Maude it is possible to equip both equations and rules with *conditions*, i. e. boolean expressions depending on the LHS of the rewrite rule, such that the rewrite rule is only applied if the condition yields “true”. This is a very powerful (and in some cases even absolutely necessary) concept, which is also supported by other rewriting-systems, like *Mathematica* [3], but not by any of the three different systems developed in the frame of this thesis.

Another very important difference are the strict typing restrictions in Maude. There, every function/operator/constant/variable explicitly needs to be assigned to a certain *sort*. Sorts themselves have no predefined meaning and must therefore be defined by the user. The purpose of sorts is helping the user forming valid expressions: An operator can only be applied to a sequence of arguments if the sorts of the arguments correspond to the signature of the operator. Please note that in the programs of this thesis there are no typing restrictions at all: Symbols are just symbols without any meaning (see Section 2.2 for more information about well-formed expressions).

One more difference is the support of sequence variables (Section 2.6): Maude, unlike *Mathematica*, does not support sequence variables. However, it provides quite elegant ways to overcome this fact: Operators can be equipped with certain *attributes*, like associativity, commutativity, and many more. This allows the user to represent arbitrary sequences of objects as nested applications of one and the same (associative+commutative) operator, and still design the rewrite rules in a very short and intuitive way, almost as if sequence variables were involved. An example on this issue can be found in the appendix, in Section B.9.

Of course, the list of differences from above is by no means complete; Rather, it covers the most important (in my view) features of Maude and compares them to the features of the rewriting-systems developed in the frame of this thesis. As already mentioned above, Section B.9 in the appendix contains a short example of a Maude-module: The Maude-version of the sequent calculus rule set **SequentCalculus**, which can be found in the appendix, too. For more information on the syntax and semantics of a module in Maude, and how one can actually work with it, please let me refer to [7].

Chapter 2

Basic Definitions

2.1 Formal Definition of Rewriting

As pointed out in the introduction, *rewriting* is the very well-known (especially in logic and computer science: [4, 5, 11]) art of transforming expressions (or “terms”, see Section 2.2) into other expressions by successively applying so-called rewrite rules. This section aims at formalizing this rather intuitive concept in order to keep the thesis self-contained. First of all, it is necessary to give various auxiliary definitions, such as *substitution* and *pattern matching*.

Definition 1. Let x_1, x_2, \dots, x_n be pairwise different variables. A set σ of the form $\{x_1 \rightarrow e_1, x_2 \rightarrow e_2, \dots, x_n \rightarrow e_n\}$ is called a *substitution* if every e_i is an expression (or a sequence of expressions, if x_i is a *sequence variable*; see Section 2.6) where none of the variables occurs.

Remark 2. The above definition is not the most general one. In principle it is also allowed that the e_i do contain variables; However, this is not needed in the frame of this thesis.

Please note that the empty set is a substitution as well, called *empty substitution*. The next definition states what can be done with substitutions.

Definition 3. Let a be an expression, $\sigma = \{x_1 \rightarrow e_1, \dots, x_n \rightarrow e_n\}$ a substitution. The application of σ to a , denoted as $a\sigma$, is again an expression, where every occurrence of x_i in a is replaced by e_i , for all $1 \leq i \leq n$.

Now that the very basic notions of substitution have been made clear, let’s move on to *pattern matching*, or in short simply *matching*. Pattern matching means

trying to make two expressions equal, by providing a suitable substitution for the variables of one of the two expressions.

Definition 4. Let a and b be two expressions. a matches b iff there exists a substitution σ such that $a\sigma = b$. This is denoted by $\text{match}(a, b, \sigma)$.

In what way is matching beneficial for rewriting? The answer is quite simple: It allows us to define the *rewrite rules*, or in short simply *rules*, that are the very basic ingredients for every rewriting process.

Definition 5. A rewrite rule r is a pair of expressions (lhs, rhs) , where every variable occurring in rhs occurs in lhs as well, but not necessarily vice versa. $\text{lhs}(r)$ denotes lhs , $\text{rhs}(r)$ denotes rhs , and $\text{var}(r)$ denotes the set of all variables occurring in lhs .

Rules can be grouped together in a so-called *rule set*, in the following always denoted as \mathcal{R} . However, since in most applications (also here in this thesis) the order of the rules is important, the word “set” is misleading; It is more a tuple of rules, but for now this does not really matter.

Definition 6. Let a and b be two arbitrary expressions. a can be rewritten in one step into b w.r.t. the rule set \mathcal{R} , denoted by $a \rightarrow_{\mathcal{R}} b$, iff there exist a subexpression a' of a , a rule $r \in \mathcal{R}$ and a substitution σ for $\text{var}(r)$ such that

1. $\text{match}(\text{lhs}(r), a', \sigma)$ (r matches a')
2. b is equal to a where a' is replaced by $\text{rhs}(r)\sigma$

Obviously, for given \mathcal{R} , $\rightarrow_{\mathcal{R}}$ defines a relation on the set of expressions. But one could also have a look at the reflexive-transitive closure of this relation:

Definition 7. Let a and b be two arbitrary expressions. a can be rewritten into b w.r.t. the rule set \mathcal{R} , denoted as $a \rightarrow_{\mathcal{R}}^* b$, iff there exist expressions b_1, b_2, \dots, b_n such that

$$a = b_1 \rightarrow_{\mathcal{R}} b_2 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} b_n = b.$$

Now it should be clear what it means to rewrite an expression a w.r.t. a rule set \mathcal{R} : It is just trying to find an expression b with $a \rightarrow_{\mathcal{R}}^* b$ and b itself cannot be rewritten any more (of course, such an expression does not need to exist in general). That is exactly what the programs implemented in the frame of this thesis do.

In the rest of this section it is explained how expressions, rules, rule sets, etc. are given in this thesis.

2.2 Expressions

Expressions, or terms, are one of the very basic ingredients of rewriting systems and thus well-known concepts in logic and computer science [6]. In fact, the formal definition of expressions given below is quite similar – though not completely equivalent – to the inductive definition of expressions in predicate logic. Since there are many other programs, as *Mathematica* [3] or Maude [7], that are rewriting systems as well, they have an own definition of expressions/terms; However, all those different definitions are very similar to each other, and therefore similar to the one shown below, too.

Let us see now how expressions are given within the frame of this thesis, i. e. both in the rule sets as well as in the programs. Indeed they are given by strings of characters, as one would expect, where function symbols have to appear *before* their argument list, which is enclosed by square brackets ($[,]$). The arguments of a function are separated by commas ($,$), followed by at most one additional whitespace character in order to increase readability. All other whitespaces are part of the next symbol, which means that any symbol may start with or even consist only of whitespace characters. This is a general paradigm: Characters have absolutely no predefined meaning (apart from square brackets, commas, and partially whitespaces) - A “+”, for example, may be the name of a function, the name of a constant, or only part of a symbol. It does surely not stand for addition, in any sense, unless defined by the user.

One could also give a more formal definition of expressions:

- Definition 8.**
1. Any symbol c whose name does not contain square brackets and commas, but at least one character, is an expression. In this case, c stands either for a constant or a variable.
 2. If f is a symbol and a_1, a_2, \dots, a_n are expressions, then also $f[a_1, a_2, \dots, a_n]$ is an expression. In this case, f stands for a function which is applied on the arguments a_1, \dots, a_n . In connection to Version 3 (“Compiler/SV”), n could also be 0.
 3. These are all.

Note that absolutely nothing is said about the names of symbols, which means that a symbol might occur both as a constant/variable and as a function. Note also that the arity of a function symbol is not fixed, which means that the same function symbol might occur with different numbers of arguments. In fact, these two properties form the main differences to expressions/terms of other systems

as well as predicate logic: There, every symbol is *either* a constant symbol *or* a variable symbol *or* a function symbol, and every function symbol has a *fixed* arity. Summarizing, the definition of expressions given in this thesis is strictly more general than that of, e.g., *Mathematica*, Maude and predicate logic, which means that an expression/term in one of those contexts is also (equivalent to) an expression according to the above definition, but not vice versa.

Please note also that there are lots of different ways to “write down” expressions, e.g. consider the expression `+ [1, 2]` which a human reader might rather consider as `1 + 2` (the so-called *infix* notation). But there are also many other ways to represent expressions, most notably Polish notation [6] and Lisp-like notation [13].

Here are some examples of well-formed expressions:

1. `+ [1,2]`
2. `+ [1, 2]`
3. `+ [1, 2]`
4. `[a function[a,b], a function[]]`
5. `4*5[<, >, ', _]`
6. `a constant`

Note that the first two expressions are exactly the same, because the whitespace after the comma is not part of the second argument. However, the third expression is different, because there are two whitespaces after the comma, and one of them belongs to the second argument (which is thus `2`).

The fourth example again illustrates the use of whitespaces. The name of the outermost function is just , the name of its first argument contains a whitespace, and its last argument is a function without any arguments (but the same name as the first one). Remember that this is only valid in Version 3.

The fifth example shows that characters apart from `[,]` and `,` have absolutely no predefined meaning. Although one might think that, for example, `<` stands for “less than”, it is just the name of a constant.

The very last example is just a constant, which itself is also an expression.

The following are some examples of strings that don’t stand for well-formed expressions:

1. `[1, 2, 3]`

2. `f[,1]`
3. `abc[def]ghi`
4. `+, -, /`

The first example is invalid because the outermost function doesn't have a name. The second example is invalid, because the outermost function doesn't have a first argument. It would be valid, however, if there was a whitespace character right after `[`.

Example three is not well-formed, because after `]` there is no `,`.

The last example is not well-formed, because there is no enclosing outermost function for the three arguments `+`, `-` and `/`.

Please be aware that the lack of rules for defining valid expressions, compared to other (programming-) languages, gives the user huge power to give meaning to symbols just as he likes, but on the other hand it is also *him* (not the program) that needs to take care that everything he writes is really what he wants.

2.2.1 Subexpressions

One important concept that has already been mentioned in Section 2.1 is the concept of *subexpressions*. A subexpression of an expression is simply a part of that expression which is itself also an expression:

Definition 9. Let the relation \sqsubseteq on expressions be defined in the following way: $a \sqsubseteq b$ iff b is of the form $f[a_1, \dots, a_{i-1}, a, a_{i+1}, \dots, a_n]$, which only means that a is an argument in expression b .

The reflexive-transitive closure of \sqsubseteq , denoted as \sqsubseteq^* , is the desired subexpression-relation:

Definition 10. For arbitrary expressions a and b : $a \sqsubseteq^* b$ (in words: “ a is a subexpression of b ”) if and only if either $a = b$ or $a \sqsubseteq b_1 \sqsubseteq b_2 \sqsubseteq \dots \sqsubseteq b_n \sqsubseteq b$ for some b_1, b_2, \dots, b_n .

2.2.2 Position of Objects

The *objects* of an expressions are all the symbols occurring in it, where the same symbol at different places stands for different objects. The position of an object

in an expression is defined in a quite natural way: Just go through the expression, from left to right, until you reach that object. Count the number of all objects you have reached in this process - This is exactly the position. The outermost object always has position 0.

It is also possible to define the position of an object/vertex in terms of the tree representing the expression (see Section 2.7). In that case, perform *depth-first-search* until you reach the vertex and again count all other vertices visited by doing so. This number is again the position of the vertex, and it is always equal to the number resulting from the algorithm above.

Here are some examples:

- c
c: 0
- f[a, b]
f: 0
a: 1
b: 2
- f[g[a], g[b], c]
f: 0
first g: 1
a: 2
second g: 3
b: 4
c: 5

2.3 Integers and Natural Numbers

This section does not aim at defining the sets of Integers and Natural numbers, but rather at how they are treated by the programs. First of all, numbers are of course just given as, like any other expressions, strings of characters, or more precisely, strings of digits (with a possible “-” in front to indicate negative numbers). But as was explained in the previous section, also these symbols have no meaning! They are just symbols, the user can give any meaning he wants to them. But still, in many applications, it would be good to be able to deal with them as numbers. This includes adding them, multiplying them, comparing them, ..., in short: Performing arithmetic on them. So, how can this be achieved?

As the experienced reader surely knows, Natural numbers can be defined inductively by means of a minimal element (0) and a successor function (`succ`, see also [12]). Thus, one could also write, say, 2, as `succ[succ[0]]`. This notation now allows to define (rewrite-) rules for arithmetic and other things. For example, one can define Addition in the Naturals just as

$$+[m, \text{succ}[n]] \rightarrow \text{succ}[+[m, n]]$$

$$+[m, 0] \rightarrow m$$

where m and n are exactly such `succ`-objects.

But this is a very inconvenient way to represent numbers. It takes longer to type all the characters and it makes the input (and of course also the output) less readable. For that reason, a method is implemented to *automatically* transform Natural numbers given in default notation (0, 1, 2, ...) into `succ`-objects. To see how this is done, let me refer to Sections 2.5 and A.4, but the result is the following: If the user types in the *constant* (not function!), say, 3, this is internally transformed into the expression `succ[succ[succ[0]]]` - fully automatically.

But not only does this work for the *input*, but also for the *output*: If the result of a rewriting process contains as a subexpression, say, `succ[succ[0]]`, the user can request this to be printed simply as 2. Again let me refer to Section A.4 to see how the user can tell the program to do so.

Now that it should be clear what can be done with Natural numbers, let us focus on Integers. Of course, the only thing that is different to Natural numbers is the possibility of having negative numbers. Hence, one could just add an additional all-enclosing function, e.g. `neg`, to denote negative numbers. However, in any case there is one problem (also for positive numbers): The representation as `succ`-objects becomes extremely memory-consuming (and also time-consuming) for big numbers (w. r. t. absolute value). The even not that big number 867 can obviously be represented by only three digits, but would require exactly 867 `succ` functions. So, why don't use this fact? Indeed, 867 could be represented by the expression `Int[7, Int[6, Int[8, 0]]]`. This idea is also used in [12], and I will not go into more detail on that in this thesis. But it can be seen (again in [12]) that such a representation allows defining arithmetic, among other things, on the whole set of Integers, in a far more efficient way than by using `succ`-objects.

The reason for explaining all that is, of course, that any Integer can be automatically transformed into such form, just like Naturals can be transformed into `succ`-objects. Again let me refer to Sections 2.5 and A.4 to see how it is done.

2.4 Rewrite Rules

Since this thesis is about rewriting, there is a need to clarify how rewrite rules are given within its context. This, however, is rather simple: Rewrite rules are again expressions, but with a very special format. The outermost function of each rule has to be `Rule`, and it needs to have exactly three arguments. Let us skip the first argument for now and immediately focus on the last two arguments: They are just the left-hand-side (LHS) and right-hand-side (RHS) of the rule, given as ordinary expressions (see Section 2.2). Now, there is only one question remaining: What about variables? Surely many rewrite rules include variables to make them more general, but how are variables defined here? There is no special syntax for specifying what is a variable and what is a constant, like, for example, adding an underscore in *Mathematica* [3]. Rather, all symbols that should be regarded as variables, whenever they don't appear as function symbols, have to be explicitly listed in a so-called *VarList*, which is now the first argument of the `Rule` function. Its name is, as expected, `VarList`, and it can have any number of arguments (even no arguments at all). In any case, all arguments have to be pairwise different symbols.

Please note that there is one more restriction: Every variable that occurs on the RHS of a rule also needs to occur on the LHS of that rule, because otherwise the whole rule would not make sense in terms of a rewrite rule (see Definition 5). But, in contrast, it is allowed that a variable does not occur in a rule at all, neither on the LHS, nor on the RHS.

The following are some examples of well-formed rules:

- `Rule[VarList[m, n], +[m, succ[n]], succ+[m, n]]`
- `Rule[VarList[m], +[m, 0], m]`
- `Rule[VarList[m, blabla], *[m, 0], 0]`
- `Rule[VarList[], power[e, *[i, pi]], -1]`

The first two rules are those that already appeared in the previous section for defining addition in the Natural numbers.

The third rule could be regarded as one rule for defining Multiplication, again in the Naturals. Please note that variable `m` only appears on the LHS, and that variable `blabla` does not appear at all, which is perfectly fine.

The last rule does not include any variables; It could be interpreted as the algebraic identity $e^{i\pi} = -1$.

How single rules are grouped together to form whole rule sets is explained in the next section, Section 2.5.

2.5 Rule Sets

Now that it should be clear how single rules are given, the next step is to show how they are grouped together to form whole rules sets. And as it turns out, this is quite simple: Rule sets are simply ordinary text files, with the rules included line-by-line.

But there are even more things that can be done. For example, recall the section about Integers and Natural numbers. As was promised there, it will now be explained how numbers occurring in *rule sets* can be automatically transformed into `succ` or `Int` objects. This is achieved by the three commands `#MAKE_NOTHING`, `#MAKE_SUCC` and `#MAKE_INT`. They can appear anywhere in the file (again in one separate line) and they affect all rules coming after them until the next such command or the end of the file. The first command, `#MAKE_NOTHING`, leaves numbers unchanged; The second one, `#MAKE_SUCC`, transforms all subsequent Natural numbers into `succ`-objects; And finally the last one, `#MAKE_INT`, transforms all subsequent Integers into `Int`-objects.

Another important issue is including rules that are already grouped in another rule set. For example, imagine again Natural numbers. First, one usually defines addition by specifying several rules, which are then put together into one rule set. Afterwards, one realizes that also multiplication would be quite useful. Thus one starts defining the rules for multiplication, but then sees that this operation cannot be defined without addition. So, what now? Of course one can again write down all the rules for addition in the rule set for multiplication, but this is very inconvenient, especially since these rules are already available (but in a different file). For that reason there is another command, `Needs[filename]`, that includes all rules of the rule set stored in the file *filename*. Again, this command can appear anywhere in the file (and of course also more than just once), and the rules are included exactly after the last rule before this command and before the next rule after this command. This is important, since the order of rules really matters in the implementation of rewriting by the programs of this thesis. Also note that in principle it is possible that there are circles of inclusions, i.e. that a rule set is included in another rule set, which itself is included in the first set. Such circles do not cause infinite loops when loading the files, since the programs keep track on which files have already been loaded and which haven't. Including a file that has already been loaded has no effect!

For increasing readability, lines can be also left empty, for example to visually separate single rules or blocks of rules. For documentation, also comments can be made. Just like all other commands, they have to appear in single lines as well, starting with two slashes (`//`).

Summarizing, the following are the ingredients of a file containing a rule set:

- `#MAKE_NOTHING`, `#MAKE_SUCC`, `#MAKE_INT`
- `Needs [...]`
- Rules themselves, as described in the previous section
- Comments
- Empty lines

Rule sets used for testing can be found in the appendix.

2.6 Sequence Variables

For some problems in the vast area of rewriting it is not enough to only have “normal” (*individual*) variables available when defining rules for solving them. For example, think of the problem of sorting a sequence of Natural numbers (possibly represented as `succ`-objects). If the length of the sequence is arbitrary and unbounded, it is almost impossible to come across reasonable rules for doing the job without using *sequence variables*. The concept of sequence variables is rather simple and taken from [10]:

Definition 11. A *sequence variable* may be instantiated not only with *one* expression, but with a whole (finite!) sequence of expressions, possibly even the empty sequence.

This means, for example: If, on the LHS of a rule, a function appears whose only argument is a sequence variable, then the argument list of that function matches *any* argument list (Of course, if the sequence variable occurs more than once, then all occurrences have to be instantiated with equal sequences of expressions).

In the frame of this thesis, sequence variables can be used only in connection with Version 3 (“Compiler/SV”). In rule sets, they are indicated by the suffix “`_`”

after the variable name. This means, every variable whose name ends with an underscore is a sequence variable, if the version mentioned above is used. In any other version they are just ordinary individual variables.

There are also other rewrite systems with such a concept, most notably *Mathematica* [3], where sequence variables are indicated by either two or three underscores. Note that three underscores are needed if the variable should also have the possibility to be instantiated with an empty sequence, just as the sequence variables in this thesis.

More information on sequence variables can be found, for example, in [10].

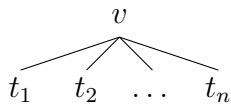
In order to see one solution for the problem of sorting sequences of Natural numbers, have a look at the rule set **Sorting** that can be found in the appendix, in Section B.7. This rule set is capable of sorting sequences of Natural numbers without holes, i.e. where no number between the sequence minimum and -maximum is missing, and where the numbers are given as **succ**-objects.

2.7 Trees

The next thing that needs to be clarified is *how* expressions are represented internal to the programs. There are many ways, and one is of course to represent them as strings of characters, just as they are given in rule sets (see Section 2.2). However, this is very inefficient: It is hard to collect all arguments of a function, to even count the number of arguments, and, last but not least, to replace subexpressions by other subexpressions, as has to be done in the rewriting process. Rather, a completely different approach is pursued: There is a natural 1-1 correspondence between expressions and *trees*: Trees are again a very well-known and well-studied data type in computer science (Section 2.3 in [9]) that is mainly used to store information with an inherent strictly hierarchical structure. Representing expressions as trees allows to efficiently retrieve information about them and manipulate them. But first of all, trees have to be formally defined.

Trees are a special kind of *directed, acyclic graphs*, i.e. graphs where all edges also have a direction, and where one cannot start in a vertex, travel through the graph by going along edges in their respective directions, and finally arrive again at the starting vertex. In addition to that, for every vertex in a tree there may be at most one edge “arriving” in it. The following is a formal definition of trees:

- Definition 12.**
1. A single vertex is a tree.
 2. If v is a vertex and t_1, t_2, \dots, t_n are trees, then also



is a tree. v is the parent of the t 's and the t 's are its children. The direction of the edges is from v to t_i , for any i .

3. These are all.

Definition 13. In a tree, vertices without children are called *leaves*, other vertices are called *nodes* or *internal vertices*. The *arity* of a node is exactly the number of its children (Hence, one could also say that leaves are nodes with arity 0). A vertex u is called *successor* of a vertex v iff it is the child of a child of a child ... of a child of v .

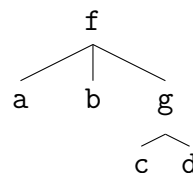
As one easily sees, in every tree there is exactly one vertex without a parent; This vertex is called *root* of the tree. Each vertex v is the root of a *subtree* denoted as $[v]$, which is the tree containing all successors of this vertex.

The previously mentioned 1-1 correspondence between expressions and trees can be expressed by the following identities:

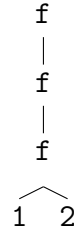
- Expression \leftrightarrow Tree
- Subexpression \leftrightarrow Subtree
- Function \leftrightarrow Node
- Arguments of function \leftrightarrow Children of node
- Function arity \leftrightarrow Node arity
- Constant/Variable \leftrightarrow Leaf
- Outermost symbol \leftrightarrow Root

In order to illustrate the concept of trees used for representing expressions, here are some examples:

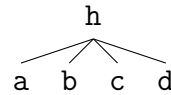
- $f[a, b, g[c, d]]$



- $f[f[f[1, 2]]]$



- $h[a, b, c, d]$



- 0

0

In the subsequent sections, both expression- and tree terminology will be used to describe exactly the same objects, namely expressions. However, whenever the focus lies on representation and, say, the word “function” is used, it should be clear that actually “node” is meant.

For sake of completeness it also has to be mentioned that there is a huge variety of different ways to represent expressions/terms in an efficient way; Trees are only one of them. For instance, one could relax the definition of trees a little bit by not requiring that each vertex has at most one parent. The result are then just arbitrary directed acyclic graphs, which might also be considered as trees where nodes “share” some of their children. One of the systems that pursues exactly this approach is again the previously mentioned Maude system (Section 5 in [8]).

2.8 Mappings

Later in this thesis, *mappings* will play an important role. More precisely, *admissible* mappings from the LHS of a rule to the RHS of the same rule. So, what are those admissible mappings? First it should be made clear what a mapping is in the context of this thesis. It is basically a function, depending on two expressions, whose domain is the set of objects (functions, constants, variables (\rightarrow vertices)) of the first expression, and whose codomain is the set of objects of the second expression, plus an additional element \emptyset . Hence, let me again emphasize that a mapping *always* depends on two expressions. However, in most cases these two expressions are clear from the context, and will therefore be omitted.

2.8.1 Type- and Arity Preserving Mappings

Not all mappings between two expressions are needed to be considered. In fact, there are only two different kinds: *Type preserving* and *arity preserving* mappings, whose definitions follow below. Their meanings will become clear in Chapters 5 and 6, but let me give a brief hint already now: The purpose of type preserving mappings is, basically, to map functions to functions (without considering their respective arities), constants to constants, and occurrences of some variable to occurrences of the same variable, i.e. they preserve the *type* of objects (hence the name). Arity preserving mappings, on the other hand, map functions to functions *with the same arity*, occurrences of some variable to occurrences of the same variable, and completely ignore constants (map them all to \emptyset), i.e. they preserve the *arity* of functions (hence the name). Both kinds of mappings do so in an “as bijective as possible” manner ... what this really means is explained in their definitions below (note that the additional element \emptyset in the mapping’s codomain is in fact only needed to ensure injectivity, in some sense).

Examples of mappings can be found a bit later in this section, in Subsection 2.8.2.

Definition 14. A mapping μ is called *type preserving* iff

1. Every function is mapped either to a function or to \emptyset
2. Every occurrence of a variable v is mapped either to an occurrence of the same variable v or to \emptyset
3. Every constant is mapped either to a constant or to \emptyset
4. Let D denote the domain of μ and $N := \{x \in D \mid \mu(x) = \emptyset\}$. Then $\mu|_{D \setminus N}$ is injective. In other words, \emptyset is the only element of the codomain that may be reached by more than one element of the domain (injectivity)
5. If there is a function that is mapped to \emptyset , then every function of the codomain is in the range of the mapping (surjectivity)
6. If there is a variable v that is mapped to \emptyset , then every such variable v of the codomain is in the range the mapping (surjectivity)
7. If there is a constant that is mapped to \emptyset , then every constant of the codomain is in the range of the mapping (surjectivity)

Definition 15. A mapping μ is called *arity preserving* iff

1. Every function is mapped either to a function *of the same arity*, or to \emptyset

2. Every occurrence of a variable v is mapped either to an occurrence of the same variable v or to \emptyset
3. Every constant is mapped to \emptyset
4. Let D denote the domain of μ and $N := \{x \in D \mid \mu(x) = \emptyset\}$. Then $\mu|_{D \setminus N}$ is injective. In other words, \emptyset is the only element of the codomain that may be reached by more than one element of the domain (injectivity)
5. If there is a function that is mapped to \emptyset , then every function of the same arity of the codomain is in the range of the mapping (surjectivity)
6. If there is a variable v that is mapped to \emptyset , then every such variable v of the codomain is in the range the mapping (surjectivity)

There are only two differences between type- and arity preserving mappings: In arity preserving mappings, the arity of a function has to be preserved by the mapping (hence the name), and constants are always mapped to \emptyset .

Definition 16. A mapping is said to be *admissible* iff it is type preserving or arity preserving (or both).

For type/arity preserving mappings also an inverse mapping can be defined, which itself is type/arity preserving as well. For that purpose, let again be μ such a mapping and e_1, e_2 the two expressions where it is defined on, D its domain and C its codomain. Then μ^{-1} is defined in the following way:

1. For any $x \in D$: If $\mu(x) = y$ and $y \neq \emptyset$, then obviously y is in the domain of μ^{-1} and $\mu^{-1}(y) = x$.
2. For any other element w in the domain of μ^{-1} which is not dealt with in the first case, define $\mu^{-1}(w) = \emptyset$.

By construction of the inverse, the following theorem holds:

Theorem 17. Let μ be a mapping with domain D and codomain C . Then μ^{-1} is a mapping with domain $\overline{D} = C \setminus \{\emptyset\}$ and codomain $\overline{C} = D \cup \{\emptyset\}$ such that

1. For all $x \in D$: Either $\mu(x) = \emptyset$ or $\mu^{-1}(\mu(x)) = x$
2. For all $y \in \overline{D}$: Either $\mu^{-1}(y) = \emptyset$ or $\mu(\mu^{-1}(y)) = y$

2.8.2 The Notation of Mappings

In the following, the position of an object x in the expression $expr$ is denoted by $\varphi_{expr}(x)$ (see Section 2.2.2 for the definition of “position”). Let μ be a mapping between expressions a and b . Then this mapping is denoted by $\{\varphi_a(x_1) \rightarrow \varphi_b(\mu(x_1)), \dots, \varphi_a(x_n) \rightarrow \varphi_b(\mu(x_n))\}$, where x_1, \dots, x_n are all the objects of a , and additionally, for any expression $expr$, $\varphi_{expr}(\emptyset)$ is set to -1. Please note that sometimes all the terms $n \rightarrow -1$, for any n , are omitted in order to make the notation shorter.

The following are some examples of pairs of expressions plus all of their admissible mappings (m and n denote variables, all other non-function symbols are constants):

- $+ [m, S[n]], S[+[m, n]]$
 1. $\{0 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow 1, 3 \rightarrow 3\}$ (type)
 2. $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0, 3 \rightarrow 3\}$ (both)
- $* [m, S[n]], +[*[m, n], m]$
 1. $\{0 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow 1, 3 \rightarrow 3\}$ (type)
 2. $\{0 \rightarrow 0, 1 \rightarrow 4, 2 \rightarrow 1, 3 \rightarrow 3\}$ (type)
 3. $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0, 3 \rightarrow 3\}$ (type)
 4. $\{0 \rightarrow 1, 1 \rightarrow 4, 2 \rightarrow 0, 3 \rightarrow 3\}$ (type)
 5. $\{0 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow -1, 3 \rightarrow 3\}$ (arity)
 6. $\{0 \rightarrow 0, 1 \rightarrow 4, 2 \rightarrow -1, 3 \rightarrow 3\}$ (arity)
 7. $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow -1, 3 \rightarrow 3\}$ (arity)
 8. $\{0 \rightarrow 1, 1 \rightarrow 4, 2 \rightarrow -1, 3 \rightarrow 3\}$ (arity)
- $f[m, 0], g[f[m, 0], 1]$
 1. $\{0 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow 3\}$ (type)
 2. $\{0 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow 4\}$ (type)
 3. $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3\}$ (type)
 4. $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 4\}$ (type)
 5. $\{0 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow -1\}$ (arity)
 6. $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow -1\}$ (arity)

2.8.3 The Number of Mappings

As it will turn out, some of the programs have to compute *all* of the type/arity preserving mappings between two expressions. The reason behind this effort is explained in Chapters 5 and 6. However, in any case, it would be good to know *how many* of those mappings there are. For that purpose, let a and b in the following paragraphs be two expressions, let $\mathcal{V} := \{x_1, x_2, \dots, x_v\}$ be the union of the sets of variables of a and b , and let d be the maximum arity of functions occuring in a or b . Let furthermore denote

- $v_{a,i}, v_{b,i}$ the number of occurrences of the variable x_i in a and b , resp., for all $1 \leq i \leq v$
- $f_{a,i}, f_{b,i}$ the number of functions of arity i in a and b , resp., for all $0 \leq i \leq d$
- c_a, c_b the total number of constants in a and b , resp.
- f_a, f_b the total number of functions in a and b , resp., i. e. $f_a = f_{a,0} + \dots + f_{a,d}$ (analogous for f_b)
- $v_{m,i} := \min\{v_{a,i}, v_{b,i}\}, v_{M,i} := \max\{v_{a,i}, v_{b,i}\}$ for all $1 \leq i \leq v$
- $f_{m,i} := \min\{f_{a,i}, f_{b,i}\}, f_{M,i} := \max\{f_{a,i}, f_{b,i}\}$ for all $0 \leq i \leq d$
- $c_m := \min\{c_a, c_b\}, c_M := \max\{c_a, c_b\}$
- $f_m := \min\{f_a, f_b\}, f_M := \max\{f_a, f_b\}$

Before actually computing the number of mappings, another definition is needed:

Definition 18. For $m \geq n \in \mathbb{N}$: m^n is called *falling factorial* and defined as $m^n := \frac{m!}{(m-n)!}$

Let us first consider type preserving mappings. After applying basic combinatorics, one gets that the total number of type preserving mappings between a and b is equal to

$$n_{a,b,type} = \prod_{i=1}^v v_{M,i}^{v_{m,i}} \cdot f_M^{f_m} \cdot c_M^{c_m} \quad (2.1)$$

For arity preserving mappings the formula looks similar:

$$n_{a,b,arity} = \prod_{i=1}^v v_{M,i}^{v_{m,i}} \cdot \prod_{j=0}^d f_{M,j}^{f_{m,j}} \quad (2.2)$$

Remark 19. Please note that the total number of admissible mappings between a and b is not just the sum of $n_{a,b,type}$ and $n_{a,b,arity}$, because some mappings might be both type- and arity preserving (like in the first mappings-example). Of course, one could also think about calculating this number, but actually it is not necessary, because always *either* type- *or* arity preserving mappings are considered, never both.

Chapter 3

General Implementation Documentation

This chapter describes some very general implementation concepts that are shared by all three different programs that have been implemented in the frame of this thesis. First, it is described how (function-/constant-) symbols are stored and shared among different objects. Then it is explained how expressions, or more precisely trees, and rules are implemented. Afterwards, the parsing algorithm of expressions and rules is described in detail, and finally the rewriting strategy, i. e. which rules are applied to which subexpressions, is explained.

3.1 Symbols

“Symbol” means in the context of this thesis of course “name of a function/constant”. Since expressions are given by strings of characters, those names are also strings of characters, and therefore the C++ standard-library class `string` (search [1] for more information) suits perfectly for representing them. But, rather than equipping every function-/constant object with such a string, all symbols are stored in, depending on the concrete version, either one or two globally accessible *list(s) of symbols*. These are simply linked lists of type `string` containing all symbols that either appear in a rule or in an expression that has been typed in by the user in order to be rewritten. If only one list is used, then really every symbol is included there; If two lists are used, then one of them is for the symbols appearing *only* in expressions typed in at run-time, and the other one is for symbols appearing in at least one rule. In any case, every symbol is included only once in those lists, even

though it might appear several times in the previously mentioned sets. Objects that implement functions/constants then only *point* to the corresponding list element; See Section 3.2 for more detail on that.

The reasons behind this concept of globally storing symbols are rather obvious: First, multiple occurrences of one and the same symbol are stored only once, which saves of course memory. Second, changing the name of functions/constants can be done by changing only one single pointer, and not by possibly allocating additional memory for storing the new string; This will be of special interest especially in connection with the reusing-paradigm. The only disadvantage is that always when a symbol is read, the whole list has to be searched in order to check whether this symbol is already included or not. However, compared to the two huge advantages, this seems to be negligible.

3.2 Implementation of Trees

Recall from Section 2.7 that *all* expressions, as defined in Section 2.2, are isomorphic to trees, and that it is exactly those trees that are really dealt with in the programs. So, how are they implemented?

Since the implementation of the programs was done in C++, all the features of an object-oriented programming language, such as inheritance and type polymorphism, as well as pointers, were available and also heavily used. Let me refer to, for example, [14, 1] for more information about those programming concepts. In particular, expressions/trees are implemented in the following way:

- Abstract base class **Expression**
- Derived:
 - Class **Constant**
 - Class **Function**
 - Class **Variable**

Although there are several differences between the various versions, the hierarchy of the classes from above is always the same. The motivation for this concept is apparent: Every tree consists of a root vertex where (0 or more) subtrees originate from. The roots of these subtrees may be nodes (functions) or leaves (constants or variables), and therefore it is good practice to make use of the object-oriented features of C++ and create an abstract superclass of functions/constants/variables that can be made the type of a node's children. This superclass, or base class, is of course class **Expression**.

3.2.1 Class Expression

This abstract class is the base of every tree element. It contains various functions, e.g. for copying and comparing trees, creating trees from strings of characters and vice versa, and many more; Most of them are *virtual* and overridden (i.e. implemented) in the derived classes. Note that class **Expression** also contains the list(s) of symbols mentioned in Section 3.1 as *static* members.

3.2.2 Class Constant

This class implements constants. It contains the name of the constant in terms of a **string**-pointer pointing to the corresponding element in the list of symbols, see Section 3.1. In “Interpreter” and “Compiler” (Versions 1 and 2), there is only one instance of class **Constant** for every constant, no matter how often it appears. All of these instances are stored in, again either one or two, *list(s) of constants*, working exactly as the list(s) of symbols. These lists of constants are contained in class **Constant** as static members.

3.2.3 Class Function

This class implements functions/nodes. Like class **Constant** it contains the name of the function in terms of a **string**-pointer, but apart from that it also contains its arguments/children. Since function arguments can be anything (functions, constants, variables), they are stored as members of type **Expression**. In “Interpreter” and “Compiler” (Versions 1 and 2), they are stored in an array, in “Compiler/SV” (Version 3) as a linked list. Consult the implementation documentation of the various versions for more information, in particular Section 4.1, Section 5.2 and Section 6.2.

3.2.4 Class Variable

This class implements variables in the LHS and RHS of rules. Although each variable has a *name* when writing down the rules in terms of Section 2.4, those names do not really matter; The only important thing is whether two variables are *equal* (i.e. have the same name) or not (have different names). However, this can also be modelled with numbers, or more precisely Natural numbers, so-called

indices, meaning that two variables have the same index iff they are equal. So, instead of doing the same thing with pointers to strings as in classes **Constant** and **Function**, every variable simply contains a member of type **int**, which is exactly that corresponding index. Note that, although **int** contains negative numbers as well, variable indices are always non-negative.

3.3 Implementation of Rules

First of all, in the compiler-versions 2 and 3, there are huge differences between the implementation of rules in the compiler and the compiled programs. This section only deals with their implementation in the respective compiler, because it is very similar to their implementation in the interpreter (Version 1). For information about the compiled programs, have a look at the chapters about the various versions.

The implementation of rules is done with the class **Rule**. Unlike before, when talking about expressions, this is no base class for other classes, but the one and only class needed for representing rules. It contains, among other things, two pointers to **Expression** objects, namely to the expressions defining the LHS and the RHS of the rule. Other important members are the number of variables of the rule (of type **int**) and the list of the names of all rule sets that have been loaded into the program (static member; must be known since every rule set must be loaded at most once).

The most important functions of this class are the two functions for matching an expression and rewriting an expressions. But, similar to expressions, there are also functions for transforming a string of characters, which defines a rule according to Section 2.4, into an instance of class **Rule**, and vice versa.

For every rule a new instance of class **Rule** is created, with its members set accordingly, and all of those instances are stored in a *list of rules* in the main file of the program implementation; See the chapters dealing with the implementations of the various versions for more information.

3.4 Parsing Expressions and Rules

Parsing an expression is straight-forward: The string of characters is scanned from left to right until the first opening square bracket is reached. If no such square bracket is reached at all (because the string ends or a comma is reached), the

expression is only a constant; In this case, first of all the name of that constant is searched in the list(s) of symbols, and if it is not found, a new list entry is created. Then, the corresponding instance of class **Constant** is either searched in the list(s) of constants (Versions 1 and 2), or a new instance of class **Constant** is created immediately (Version 3).

Otherwise, the outermost object is a function; Again, the name of that function (i.e. everything *before* the square bracket) is searched in the list(s) of symbols and added if not found. Then, a new instance of class **Function** is created and the position of its closing square bracket is searched. Everything between the two brackets are arguments of the function, and the parsing algorithm is applied recursively on it. Note that in every version where function arguments are stored in an array, first the *number* of arguments is counted in order to allocate that array.

The only minor difficulties come from the fact that, after a comma, the first white-space character has to be ignored, as imposed by the definition of expressions in Section 2.2, and from the alternative representation of Integers and Natural numbers: Whenever the parsing algorithm comes across a constant consisting only of digits (and possibly a “-” in front) and the user requested to transform numbers automatically into **Int**- or **succ** objects, then it of course doesn’t simply create the constant as it is, but creates its alternative representation as an **Int**- or **succ** object, respectively.

Concerning rules, the parsing is done almost completely in the same way as for expressions. This is due to the fact that rules are given just as expressions, but in an even more restrictive format (name of outermost function needs to be “Rule”, ..., see Section 2.4). The only difficulty arises from the presence of variables: First of all, the arguments of the **VarList**-function need to be pairwise different symbols, which can be checked very easily. Secondly, all those symbols have to be stored temporarily, because whenever there is a non-function symbol in the LHS or RHS whose name coincides with one of the variable-symbols, then it is no constant, but a variable. Again, this can be done in a straight-forward manner, by simply passing the list of variable-symbols as an additional argument to the parsing algorithm.

The inverse of the parsing function is the function that, given a concrete expression or rule, creates its representation as a string of characters. This, however, is rather trivial. The only minor difficulties come again from **Int**- and **succ** objects that are requested to be transformed into numbers.

3.5 Rule-Application Strategy

Assume there is an expression that has to be rewritten using a given set of rewrite-rules. It is quite obvious, that, depending of course on the expression and the rules, there are possibly many (more than one) subexpressions that might be rewritten using possibly many (more than one) rules. So, there are in general many different ways how to actually rewrite that expression, and thus the programs need to choose on of them. But how is this done?

Let's have a look at the rules first. Assume there is an expression a that (as a whole) is matched by the LHS of any of the rules r_1, r_2, \dots, r_n , for $n \geq 2$, and for all $i < j$ the rule r_i really comes before the rule r_j in the rule set (the order of the rules is preserved). Then there are two possible behaviors of any of the programs: Either they just choose the *first* rule r_1 , or they choose one of the rules *randomly*. This behavior can be controlled by the user; Let me refer to Section A.6 for more information on that. Please note that if the first rule is chosen, as soon as the first rule is found, the searching for further rules is stopped and the rule is immediately applied. If a random rule is applied, then *all* rules are tried and the ones that really match the expression are stored temporarily in a list. Hence, always choosing the first rule works a lot faster in general.

Now that it should be clear which *rule* is chosen to be applied, if there are more than one possibilities, it would be interesting to know which *subexpression* is chosen to be rewritten, if there are more than one possibilities. Again, there is a certain strategy that is followed by all of the programs, but unlike before, this strategy cannot be changed by any means by the user: Always the *leftmost innermost* possible subexpression is rewritten. "Leftmost innermost" can be defined in the following recursive way, exploiting the fact that expressions are isomorphic to trees:

Definition 20. Let v be the root of a tree. Then the *leftmost innermost* vertex of that tree is denoted by $\text{first}(v)$ and defined in the following way:

1. If v is a leaf, then $\text{first}(v) := v$
2. Otherwise, if v is a node and c_1, \dots, c_n are its children, then $\text{first}(v) := \text{first}(c_1)$

However, since not only the leftmost innermost vertex is needed, but the leftmost innermost *possible subtree* (where "possible" means here: "matched by the LHS of at least one rule"), another definition has to be given:

Definition 21. Let v be the root of a tree and let P be an arbitrary unary predicate defined on trees. Then the root of the *leftmost innermost possible* subtree of $[v]$ w.r.t. P is denoted as $\text{first}_P(v)$ and defined in the following way:

1. If v is a leaf, then

$$\text{first}_P(v) := \begin{cases} v & : P([v]) \\ \emptyset & : \text{otherwise} \end{cases}$$

2. Otherwise, if v is a node and c_1, \dots, c_n are its children, then

$$\text{first}_P(v) := \begin{cases} \text{first}_P(c_i) & : \text{first}_P(c_i) \neq \emptyset \wedge \forall_{j < i} : \text{first}_P(c_j) = \emptyset \\ v & : P([v]) \wedge \forall_j : \text{first}_P(c_j) = \emptyset \\ \emptyset & : \text{otherwise} \end{cases}$$

If $P(t)$ is defined as “ t is matched by the LHS of at least one rule”, then the subexpression of expression a that is chosen to be rewritten corresponds to the subtree $[\text{first}_P(v)]$, where v is the root of the tree corresponding to a , if such a subtree exists.

Some examples are given to illustrate the concept of “leftmost innermost possible”. For that purpose, let $\text{Rule}[\text{VarList}[x, y], f[x, y], x]$ be the only rule in the rule set.

- $f[0, 0]$: The only possible subexpression is the expression itself
- $f[f[0, 0], f[1, 1]]$: Three possible subexpressions; The one starting at position 1 (i.e. $f[0, 0]$) is leftmost innermost
- $f[g[0, 0], f[1, 2]]$: Two possible subexpressions; The one starting at position 4 (i.e. $f[1, 2]$) is leftmost innermost

The implementation of the concept described above is done in the following way, with the help of a *stack*:

1. Set v to the root of the considered tree
2. Search $v_1 := \text{first}(v)$ and push all nodes between v and v_1 on the stack
3. Compute $P([v_1])$. In case of success, apply the corresponding rule on $[v_1]$, set v to the root of the resulting tree and go back to step 2.
Otherwise, distinguish three cases:

- (a) If p is the parent of v_1 (p is then top of stack) and v_1 has a right neighbor v_2 in the children of p , set $v := v_2$ and go back to step 2
- (b) If p is the parent of v_1 , but v_1 does not have a right neighbor in the children of p , pop p from the stack, set $v_1 := p$ and go back to step 3
- (c) If v_1 does not have a parent, we are done, because there is no possible subtree at all

Now that the overall rewriting strategy of the programs is made clear, some words about its consequences: Of course, given a concrete expression and a concrete rule set, the result of the rewriting process might be different if a different strategy was pursued. That means that indeed all the results *may* depend on the strategy. But, however, there are some rule sets that have the so-called *Church-Rosser* property: No matter *how* the rules are applied to a given expression, after sufficiently many rewriting steps the result will always be the same. More information about the Church-Rosser property can be found, for instance, in [15, 4].

Chapter 4

Version 1: The Interpreter

The first version that has been created, in the following simply called “Interpreter”, works as follows: It reads a given rule set, creates and stores an internal representation of the rules, and is afterwards immediately ready to solve rewriting tasks using the previously stored rules. That is why it is called “Interpreter”: Instead of creating separate programs for each rule set (compiling them), it interprets the rules just as they are without going the long way round.

The outline of this chapter is the following:

First, the main algorithms and their implementation are presented, as well as the most important classes and their hierarchy in the implementation.

Afterwards, a short description of how the program is used is provided.

Finally, the chapter ends with a simple example that illustrates how the application of one rewrite rule to a given expression is carried out internally.

4.1 Algorithms and Implementation

The description of the implementation of “Interpreter” can be structured into four parts:

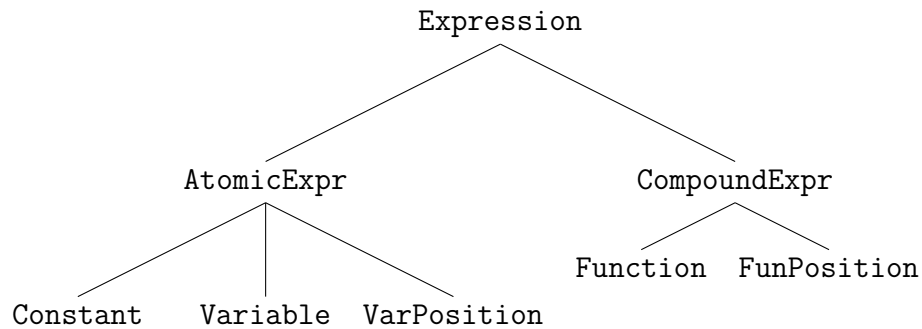
1. First of all, an overview of some implementation details is given.
2. When starting the program, the user has to specify the rule set that is used. Hence, the methods for loading a rule and creating its internal representation are described in the second part.

3. Afterwards, the user may type in expressions which then are rewritten. For that purpose, the program has to check whether a rule matches a given expression. The third part describes this matching algorithm.
4. Last but not least, if an expression a and a rule r matching a are known, a needs to be rewritten by applying r . The fourth part describes this rewriting algorithm.

There is a reason for separating the matching- and the rewriting algorithm. In principle, one could put everything together into one algorithm, which first checks whether the expression is matched, and in case of success immediately rewrites it. However, this is not possible in connection with the “use random rule” strategy (Section 3.5): First *all* rules are tried, and afterwards one of those that really match the expression is chosen to be applied. Therefore, the matching- and the rewriting algorithms have to be separated from each other.

4.1.1 Implementation Details

- As was already mentioned briefly in Section 3.2.3, the arguments of a function are stored as an array (of type `Expression**`) in the `Function`-class. This means that the arity of such an object is fixed, i. e. cannot be changed after the array was created. An additional member of type `int`, named `argCount`, contains the length of the array.
- Again, as was previously mentioned in Section 3.2.2, there is only one instance of class `Constant` for every constant, no matter how often it appears. All of those instances are stored in the static member `Constants` of type `List<Constant>` in class `Constant`.
- In addition to the three classes `Constant`, `Function` and `Variable`, there are four more classes that are derived from `Expression`: `AtomicExpr`, `CompoundExpr`, `FunPosition` and `VarPosition`. However, the first two are only auxiliary abstract classes providing functionality that is shared among some of the other classes, and the latter two are *only* used in the RHS of rules. The hierarchy is the following:



4.1.1.1 Class `FunPosition`

Like class `Function`, this class also consists of a name and an argument-array (which are therefore members of the common base class `CompoundExpr`). But in addition to that, it has a member of type `int`, named `position`, as well. This `position` refers to a position in the LHS of the corresponding rule; To which one is explained when describing the algorithm that transforms the `rhs`-object into a position tree, in Section 4.1.2.

4.1.1.2 Class `VarPosition`

Class `VarPosition` is related to class `Variable` similar as class `FunPosition` is related to class `Function`: Instead of the `index`-object it contains another member of type `int`, which is again called `position` and which again refers to a position in the LHS of the corresponding rule.

4.1.2 Loading and Storing a Rule

Assume that a string of characters describing a rule according to Section 2.4 is given. The main questions, that are answered in this subsection, are: How is the internal representation of that rule created? How is that rule stored in the program?

First of all, the parsing of rules is described in Section 3.4. Therefore, assume from now on that there is a new instance of class `Rule` where the two members `lhs` and `rhs` point to the LHS- and RHS expression of that rule, respectively, and the member `varCount` is set to the number of variables occurring in the rule. The next thing that happens is that several auxiliary members of the `Rule`-object are

set, which is done in the function `setVectorsAndArray`. The following is a list of all that objects, their meanings and their initial values (set by the previously mentioned function).

int arrayLength This object simply contains the size of the LHS of the rule, and it is set immediately in the function `setVectorsAndArray`. The size of a tree is defined quite naturally as the total number of its vertices.

vector<bool> *isVar This object is a bit-vector whose length is exactly `arrayLength`, i.e. the size of the LHS. For any index i , the i -th element of this vector is 1 (or TRUE) if and only if the object at position i in the LHS is a variable, otherwise it is 0 (or FALSE).

vector<bool> *isUsed This object is again a bit-vector with the same length as the `isVar` vector, i.e. the size of the LHS. At the beginnig, every element of it is set to 0 (FALSE). It is an auxiliary object that is needed when making the `rhs` expression a so-called *position tree* (see below).

Expression **varInst This object is an array of length `varCount`, where each element points to an object of type `Expression` (or NULL). It is, however, not set in the function `setVectorsAndArray`, but right in the constructor of class `Rule`, where each element is set to point to NULL. Later on, in the matching- and rewriting algorithm, it will contain the instances of the various variables.

There is one thing left that is done in the very last part of the `setVectorsAndArray`-function: Transforming the `rhs`-object into a so-called *position tree*. That is, replacing some of its `Function`- and `Variable` vertices by the previously mentioned `FunPosition`- and `VarPosition` vertices, respectively. This is done in the following way (by the function `toPositionExpr` of class `Expression`):

1. Go through the entire `rhs`-tree
2. Whenever you reach a `Function`-vertex f , go through the entire `lhs`-tree until you reach a `Function`-vertex g with the same arity at a position where the `isUsed`-vector is still 0. If such a vertex exists, set the element of the `isUsed`-vector that corresponds to g to 1 and replace f by a `FunPosition`-vertex with

- its name set to the name of f
 - its arguments set to those of f
 - its `position`-member set to the position of g in the `lhs`-tree
3. Whenever you reach a `Variable`-vertex v , go through the entire `lhs`-tree until you reach a `Variable`-vertex u with the same index at a position where the `isUsed`-vector is still 0. If such a vertex exists, set the element of the `isUsed`-vector that corresponds to u to 1 and replace v by a `VarPosition`-vertex with its `position` member set to the position of u in the `lhs`-tree

The resulting position tree defines an arity preserving mapping μ from the LHS of the rule to the RHS of the rule (see Section 2.8): If, at position i in the `rhs`-tree, there is a `FunPosition`-vertex with the value of `position` equal to j , then μ contains the element $j \rightarrow i$. Same for `VarPosition`-vertices.

Now there is still one question to be answered: Since there are, in general, lots of arity preserving mappings (see Section 2.8.3), why aren't all of them taken into account? The reason is that in Version 1 it simply does not matter which of them is eventually chosen: Any of them is as good as any other, so the algorithm just chooses "the first" one. In the other versions, however, it indeed does matter, and as it turns out, "the best" one will be chosen there.

The very last thing that happens before the user can eventually rewrite expressions is that an auxiliary array, named `exprArray` of type `Expression**`, is allocated. This object is a static member of class `Rule` and will be needed in the rewriting algorithm; Its length is the maximum of the values of the `arrayLength`-members of all the rules loaded into the program.

4.1.3 The Matching Algorithm

Assume in this subsection that there is an expression a and a rule r , and the task is to check whether r matches a , and if so, find out what the corresponding substitution σ is. This can be done in a rather straight-forward way: The function `matchExpression` of class `Rule` first sets all elements of `varInst` to `NULL` and then calls the `match`-function of class `Expression` on `lhs(r)` and a . The algorithm itself works as follows:

1. Let x denote the root of the first input argument (coming from the LHS of the rule) and y the root of the second input argument (coming from the expression)

2. Now distinguish several cases:

- (a) If x and y are both functions with the same arity and the same name, call the function recursively on all pairs of their arguments, one after the other. If the result of one of those recursive calls is FALSE, return FALSE; Otherwise, return TRUE
- (b) If x and y are both constants with the same name (since in Version 1 there is only one instance for constants with the same name, x and y are even the same object), return TRUE
- (c) If x is a variable and the corresponding element of the `varInst`-array still points to NULL, make that element point to y and return TRUE
- (d) If x is a variable and the corresponding element of the `varInst`-array points to the root of a tree b , compare b and $[y]$ (the subtree with root y). If they are equal, return TRUE; Otherwise, return FALSE
- (e) In any other case, return FALSE

If the result of this algorithm is TRUE, then the rule r matches the expression a and the substitution σ is given by the `varInst`-array: Assuming that the variables are x_0, x_1, \dots, x_n , if the i -th element of `varInst` points to the root of an expression e_i , then σ contains the element $x_i \rightarrow e_i$.

Two short remarks: Of course, a variable can appear more than once on the LHS of r . In that case, it is not quite clear which of the roots of the corresponding instances of a the element of `varInst` points to. However, since they have to be equal anyway, this does not matter at all.

It is in principle also possible that even after the execution of the above algorithm still some elements of `varInst` point to NULL. This is the case if the corresponding variable does not occur on the LHS at all, but then, due to Definition 5, it must neither occur on the RHS, which means that the variable is redundant.

4.1.4 The Rewriting Algorithm

Assume in this subsection that there is an expression a and a rule r , which is already known to match a with substitution σ stored in the `varInst`-array, and the task is to apply r to a , of course taking into account the reusing-paradigm. The rewriting algorithm consists of three steps:

1. Create references to (almost) all vertices of a

2. Copy the **rhs**-tree of r
3. Delete all unused vertices of a

Those three steps will be explained in detail in the next three subsections.

4.1.4.1 Step 1: Creating References

Creating references means: Fill the auxiliary **exprArray**-array with pointers to the vertices of the tree corresponding to a . However, there is still one more definition required:

Definition 22. Every vertex v of $\text{lhs}(r)$ is *associated* to one vertex of a in the following way:

- If v is a function, then, since r matches a , there must be also a function g with the same name and the same arity at the same place in a . v is associated to g .
- If v is a constant, then, since r matches a , there must be also a constant c with the same name at the same place in a . v is associated to c .
- If v is a variable, then, at the same place in a , there might be any subtree. v is associated to the root of that subtree.

The **exprArray**-array is filled in the following way (let n denote the size of the LHS of r):

For all $0 \leq i < n$, the i -th element of **exprArray** points to the associated vertex of the vertex at position i in $\text{lhs}(r)$.

But there is one more thing that is done, too, when filling the **exprArray**-array: The argument-pointers of every function-object, where really a function rather than a variable of the rule is associated to, are set to NULL. This simply means that all those functions somehow “forget” their arguments, although they still exist; The only way how they can be accessed now is via **exprArray**.

4.1.4.2 Step 2: Copying the rhs-Tree

Copying a tree is done by copying its vertices and setting the edges accordingly. Since there are five types of vertices in the **rhs**-tree and most of them are treated differently, here is a detailed description how each of them is copied:

- **Function**: Copied by creating a new instance of class **Function** with the same name as the original function. The arguments of the new object are set to the copies of the arguments of the original function.
- **Constant**: Not copied at all, because there is only one instance of class **Constant** for every constant. One could say that copying an object of type **Constant** just returns the original object.
- **Variable**: Copied by copying its instance, which can be accessed via the `varInst`-array.
- **FunPosition**: Not copied, but taken from *a*: If the value of the `position`-member of the **FunPosition**-object is *i*, then copying returns the object which is pointed to by the *i*-th element of `exprArray`; That pointer is afterwards immediately set to NULL. The name of the “new” function is set to the name of the **FunPosition**-object, and its arguments are set to the copies of the arguments of the **FunPosition**-object.
- **VarPosition**: Similar to **FunPosition**-objects, but of course no name and argument-array has to be set.

The last two items illustrate the concept of reusing existing objects: Instead of copying **FunPosition**- and **VarPosition**-objects (by really creating new instances of some classes), the existing objects from *a* are reused.

4.1.4.3 Step 3: Deleting Unused Vertices

The last step is the easiest one: Just go through the `exprArray`-array and delete all non-NULL pointers which do not point to instances of class **Constant**. Note that deleting an instance of class **Function** - in this version - also automatically deletes all of its arguments. This is the reason why in step 1 they need to forget them: Because some arguments might indeed be reused, although their parent is not. Those arguments must of course not be deleted.

4.2 How To Use “Interpreter”

This section contains a short description of how “Interpreter” can actually be used when run on a computer. First some general overview is given and then some words are spent on *input files*.

4.2.1 General Overview

After starting the program, it first asks the user to provide a rule set. Since rule sets are given by text files (see Section 2.5), it is enough to just enter the name (including the file extension, for example “.txt”), and possibly also the path to it. If the program finds the file, it opens it, reads the rules, and stores them. Otherwise it returns an error message, and the user is asked to provide a different file. However, instead of typing in the name of a file, it is also possible to just type in the special command `exit`, which will immediately close the program. Alternatively, the file containing the rule set can be passed as a command line argument, too. In this case, the previously mentioned dialog is skipped.

From now on assume that the rule set has been successfully specified. This means that the user may now type in arbitrary expressions, in the format explained in Section 2.2, which, after hitting ENTER, are then rewritten using the given rules. To see how this is done internally, please consult the sections on implementation, in particular Section 3.5. The resulting expression is of course printed on the screen. So, this is what one normally expects from a rewriting program, but still there are lots of other things that can be done. For more information, consult the section on special commands in the appendix (Section A).

Please note that there is also the possibility to call the program with at most two command line arguments. If only one is provided, then it is the file containing the rule set, as already described above. If both are provided, then the first one is the rule set, and the second one has to be an *input file* (see next subsection). The program terminates immediately as soon as all the expressions and commands in the input file have been processed.

4.2.2 Input Files

Sometimes it is quite useful to rewrite whole sets of expressions at once, for example when presenting a demo, performing tests and timing experiments, and so on. For this purpose there are so-called *input files*, ordinary text files, just like rule sets, containing line-by-line the expressions that have to be rewritten, among special commands (Appendix A). Empty lines may be inserted as well in order to give the document more structure; They are simply ignored when reading the file. The results, however, are not printed on the screen, but rather written to another text file (again line-by-line) located in the same directory as the input file, and having the same name as the input file with the suffix “_out” appended (existing files with such a name are overwritten). Please note that such an output file is also

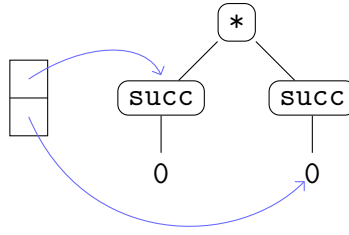


Figure 4.1: The `varInst`-array (left) pointing to the roots of the instances of the variables m (top) and n (bottom)

equipped with a time stamp in its first line.

Input files may be given either as a command line argument (see information about the individual versions), or by a special command (see Section A.7).

4.3 An Example

This example illustrates the *rewriting* process (not matching) on the basis of rewriting the expression

$$*[succ[0], succ[0]]$$

to the expression

$$+[*[succ[0], 0], succ[0]]$$

by applying the rule

$$\text{Rule}[\text{VarList}[m, n], *[m, succ[n]], +[*[m, n], m]]$$

once. Therefore, assume that the `varInst`-array (of length 2) has already been filled in the matching process, and Figure 4.1 shows how it looks like.

The `rhs`-tree of the rule, after transforming it into a position tree, is shown in Figure 4.2.

The subsequent figures all consist of five parts: On the very left, the `varInst`-array is drawn; Right to it, the old expression is drawn as it currently is; In the center,

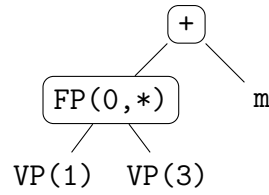


Figure 4.2: The **rhs**-tree of the rule $(FP(n, \chi))$ means: **FunPosition** with name χ and value of **position** equal to n ; Similar for $VP(n)$



Figure 4.3: After performing step 1

the new expression is drawn; On the right-hand-side, the **rhs**-tree is drawn; At the bottom, the **exprArray**-array is drawn. Blue arrows indicate pointers from the **varInst**- and **exprArray**-array to tree-vertices.

4.3.1 Step 1: Creating References

Creating references means filling **exprArray** (of length 4) accordingly. The result is given in Figure 4.3.

Please note that most of the edges in the tree are missing, because the functions “forgot” their arguments. The only exception is the edge between the first **succ** and 0 vertices: This is due to the fact that this subtree is an instance of a variable (**m**), and in such instances the arguments are still accessible by their parent functions. For the same reason, there is no element in **exprArray** pointing to the first 0.

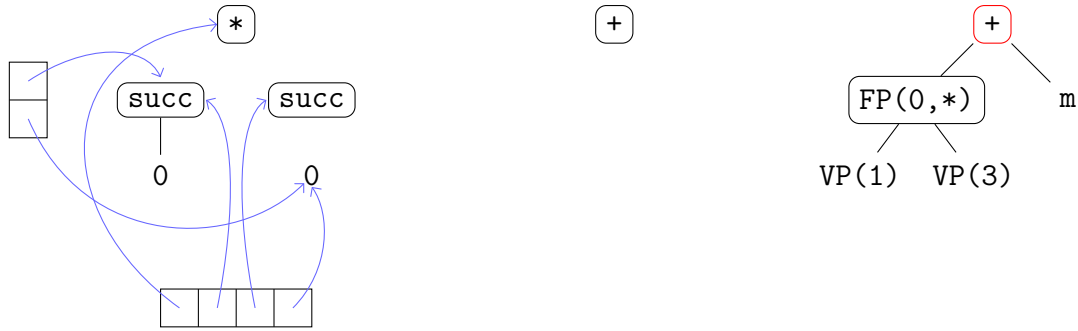


Figure 4.4: Copying the root by creating a new instance of the `Function`-class, which becomes the root of the new expression

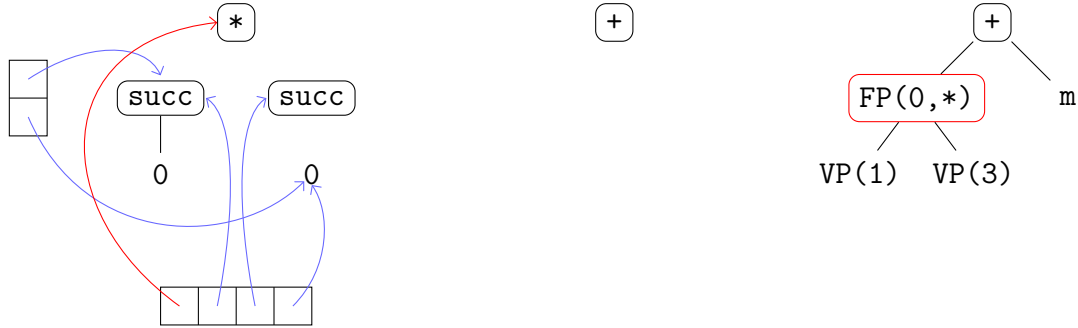


Figure 4.5: Copying the second vertex by taking an object from the old expression
...

4.3.2 Step 2: Copying the rhs-Tree

In this step, the `rhs`-tree is copied. This is done by successively copying its vertices. In the following figures, the vertex that is currently copied is marked with a red frame, and red arrows indicate how an object (and of course *which* object) is accessed.

One very important remark: In the new expression, the first argument of function `*` is exactly the same as in the old expression (Not only are they equal subtrees, but even the same objects). This means that, if the function `*` would not have forgot its first argument, the entire process of taking that argument from the old expression and moving it to its new place (Figures 4.7 and 4.8) could have been omitted. And this, of course, would make the rewriting even faster, because fewer operations had to be carried out.

This idea is taken into account in Versions 2 and 3.

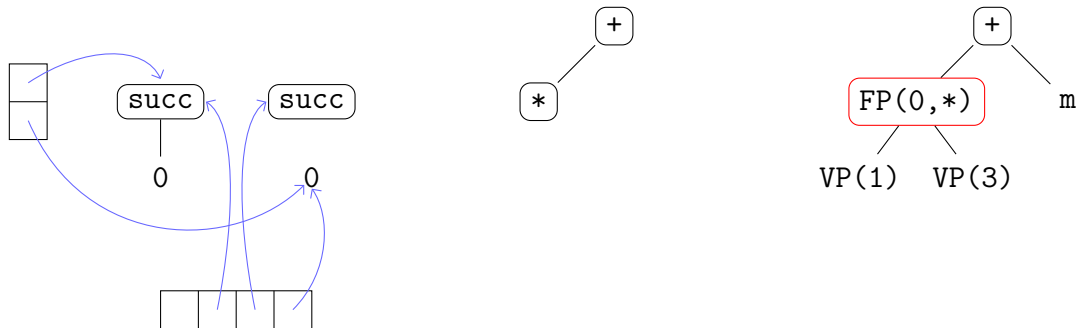


Figure 4.6: ... and moving it to its new place and setting its name accordingly (the pointer in `exprArray` is set to NULL)

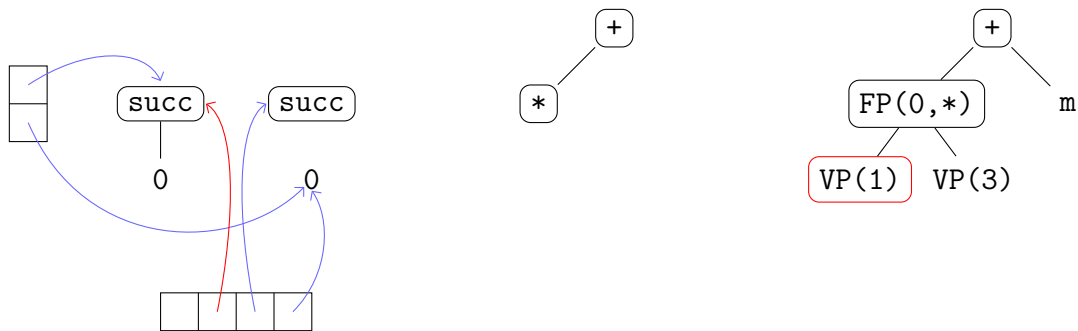


Figure 4.7: Copying the third vertex by taking an object from the old expression ...

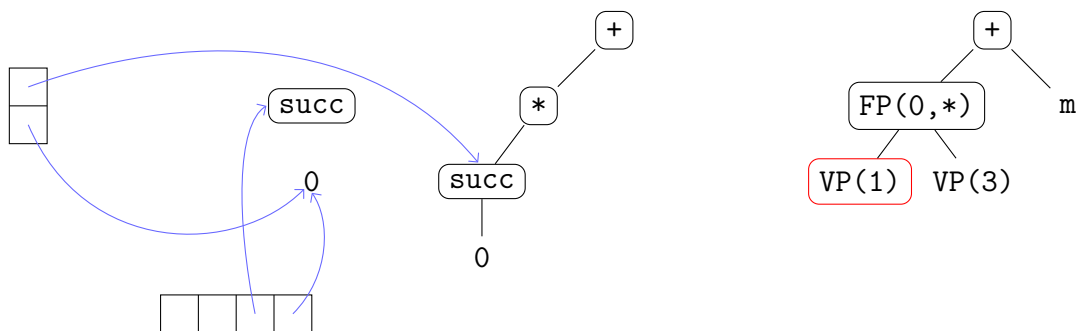


Figure 4.8: ... and moving it to its new place (its argument is *not* changed!)

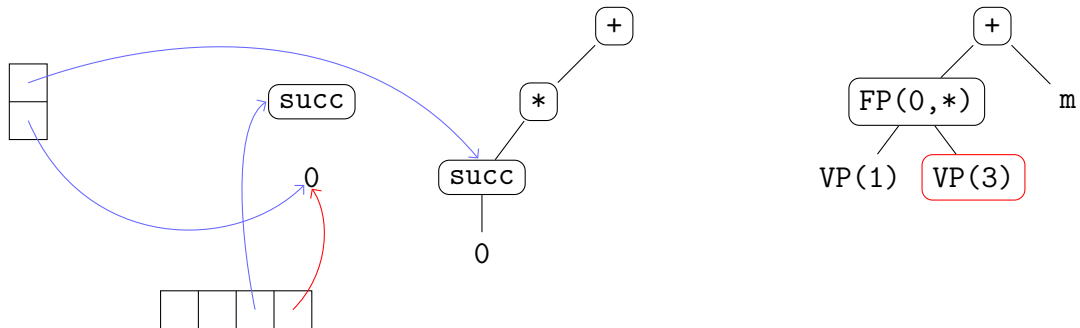


Figure 4.9: Copying the fourth vertex by taking an object from the old expression
...

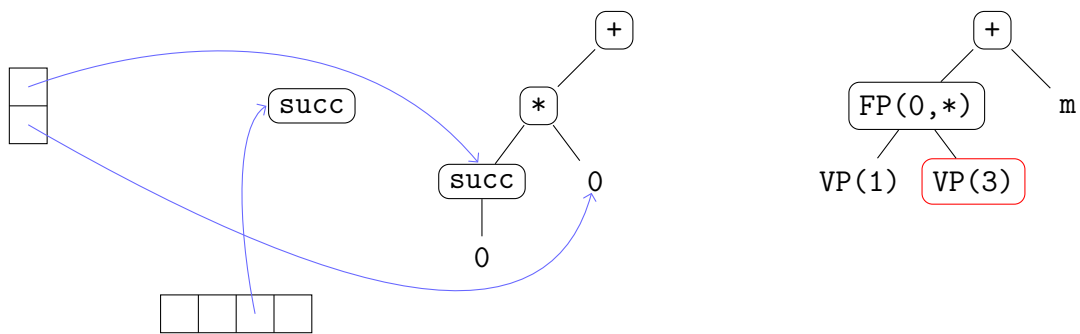


Figure 4.10: ... and moving it to its new place

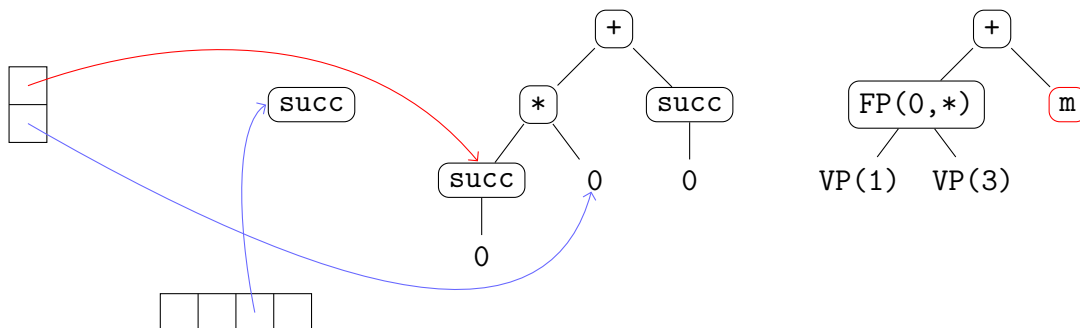


Figure 4.11: Copying the last vertex by copying the instance of variable m

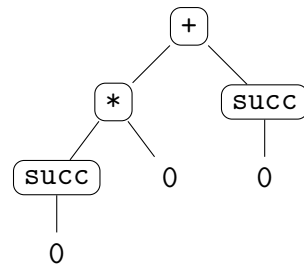


Figure 4.12: The final result after deleting unused objects

4.3.3 Step 3: Deleting Unused Vertices

In this last step, all the unused vertices of the old expression are deleted. In this concrete example, there is only one such object, namely the `succ`-object. Recall that it is exactly those objects which are still pointed to by elements of `exprArray` that have to be deleted. Figure 4.12 shows the final result, which is exactly what it is supposed to be.

Chapter 5

Version 2: The Compiler

The next version that has been created, in the following simply called “Compiler”, is very different to the previous one, the interpreter. As its name already suggests, it is a *compiler*, i.e. it transforms rule sets, given as explained in Section 2.5, into executable programs. It does so by creating C++ source- and header files containing the code that implements the rules, so to say, that have to be compiled into executable programs by an external C++ compiler afterwards. The reason for all this is obvious: It is a well-known fact that compiling is in general more efficient than interpreting, and since this thesis is all about making things more efficient, it is only natural to also try out this approach (Of course, the main idea of this thesis of reusing objects is taken into account as well).

The outline of this chapter is the following:

First, the main algorithms and their implementation are presented, as well as the most important classes and their hierarchy in the implementation. However, when talking about “implementation”, it is important to distinguish between the implementation of the *compiler* itself and the implementation of the *compiled programs*. The latter one is dealt with first, followed by a detailed description of all the compiler’s main algorithms and their implementation.

Afterwards, a short description of how the compiler and the compiled programs are used is provided.

The chapter ends with a simple example that illustrates what exactly the output of the compiler is and how a given expression is rewritten by it (after being transformed into an executable program, of course).

5.1 Implementation of the Compiled Programs

In principle, the implementation of a compiled program, in the sequel denoted as P , is very similar to that of Version 1, with one big difference: Since P already depends on a rule set \mathcal{R} , the implementation of the rules is completely different. Although there is still a class `Rule`, this one only serves as an abstract base class for all the classes implementing the rules in \mathcal{R} . Namely, if r is the i -th rule in \mathcal{R} (starting from 0), then there is a derived class of class `Rule`, called `Rulei`, which implements r (and only r) by overriding the two virtual functions `match` and `rewrite` of the base class. The meanings of those two functions are rather obvious: The first one takes an expression and checks whether it is matched by r ; The second one takes an expression that is already known to be matched by r and returns the result of rewriting it w.r.t. r .

Apart from the two functions `match` and `rewrite` there are also some other members of the base class `Rule` that are *not* overridden:

Expression `exprArray`** This static member is intended as a replacement for local variables of type `Expression*` in the `match`- and `rewrite`-functions. This means that, whenever a local variable would be needed, instead a free element of `exprArray` is taken. The reason behind this concept is that now, when calling e.g. `rewrite`, no local variables (of type `Expression*`) have to be pushed to the program stack, which makes the execution a bit faster.

Since `exprArray` is shared by all rules (that means, the functions of every derived class may use it), its length is the maximum of the numbers of the auxiliary variables of type `Expression*` needed in all of the functions `match` and `rewrite`. It is allocated and deleted only once.

Please also note that unlike in Version 1, the elements of `exprArray` have no deeper meaning any more (like “element 0 always points to the root of the expression that is rewritten”).

Function `deleteArray` This function is called before the program terminates. Its only task is to delete `exprArray`.

Function `makeSymbols` This function fills the `RuleSymbols`-array of class `Expression` with all the symbols occurring in \mathcal{R} (see Section 3.1). In this version, unlike in Version 1, two lists of symbols are used: One for the symbols appearing in the rule set and one for the symbols appearing only in the expressions typed in at run-time.

Function makeConstants This function fills the `RuleConstants`-array of class `Constant` with all the constants occurring in the rules, because, just as in Version 1, there is only one instance of class `Constant` for every constant (see Section 3.2.2). In this version, unlike in Version 1, two lists of constants are used: One for the constants appearing in the rule set and one for the constants appearing only in the expressions typed in at run-time.

Function makeRules This function basically returns an array of underlying type `Rule*` whose length is exactly the number of rules in \mathcal{R} . For any i , the i -th element points to an instance of the derived class `Rulei`. It is exactly this array that is iterated through when an expression is attempted to be rewritten, by calling first the `match`- and, in case of success, the `rewrite`-function of its elements.

The rules are implemented in the two files “Rules.h” and “Rules.cpp” (note the plural!) which are both *automatically* generated by the compiler. The only things of interest are, of course, the definitions of the two functions `match` and `rewrite`, and that is exactly what the next subsection deals with.

Please also note that there are no `FunPosition`- and `VarPosition`-classes in the `Expression`-hierarchy any more, since they are simply not needed: The mapping from the LHS to the RHS of a rule is defined by the compiler and, so to say, hard-coded in the definition of the `rewrite`-function. The same applies for the `varInst`-array: In Version 1 it is used for storing the instances of the variables when matching an expression; In this version, it is not needed any more.

5.2 Algorithms and Implementation of the Compiler

The compiler does two things: It first reads the rules of a rule set \mathcal{R} , creates their internal representation and stores them, and then creates the two files “Rules.h” and “Rules.cpp” containing the implementation of \mathcal{R} .

The first part is done quite analogously to Version 1: There is again a class `Rule` containing, among others, two pointers to the expressions representing the LHS and RHS of that rule, and so on. The only difference is that in this version the `rhs`-tree is *not* transformed into a position tree, but left as it is. Hence, in the implementation of the compiler there are no `FunPosition`- and `VarPosition`-classes.

The interesting part is of course the second part, because that is where mappings

are considered and the eventual C++ source code is created. This is also where the focus of the current section lies, however, the most important things are the definitions of the `match`- and `rewrite`-functions, or more precisely, *how* they are created.

5.2.1 Class Rule

Some words about class `Rule` and its most important members: As in Version 1, this class represents rules. It has two pointers of type `Expression*`, called `lhs` and `rhs`, pointing to the LHS and RHS of the rule, respectively. The parsing is done in completely the same way as in “Interpreter” with the static function `fromString`. However, there are no `match`- and `rewrite`-functions any more, because instances of class `Rule` do not have to match and rewrite expressions, but rather to create C++ source code that takes care of that. Therefore, there are the two functions `toHeader` and `toSource`: The first one returns, as a string, the C++ class declaration for the rule; The second one returns, again as a string, the C++ function definitions for the `match`- and `rewrite`-functions.

The most important function of class `Rule` is the function `makeCode`. This function really creates the command-sequence (see Section 5.2.3) that implements the `rewrite`-function. It depends on two important auxiliary functions, which are members of class `Rule` as well, namely `findBestMapping` and `findBestPaths`. A detailed information about what those functions do can be found in Section 5.2.4.

Class `Rule` also consists of lots of private variables; Mentioning all of them would lead to far, since most of them are only auxiliary variables that are only needed in some very special parts of function `makeCode`. However, some are indeed of interest:

int *exprArrayIndex This array, whose length is the size of the LHS, is used for storing meta-information about the vertices of the `lhs`-tree. The meaning of single elements varies depending on which part of the main function `makeCode` is currently executed. More information can be found in Section 5.2.4.4.

Expression **bestInvMap This array, whose length is the size of the RHS, is used for storing an arity preserving mapping from the RHS of the rule to its LHS (That’s why the name “Inv”: Normally, mappings from the LHS to the RHS are considered). It is filled in function `makeCode`: If the i -th element points to the vertex at position j , then $i \rightarrow j$ is an element of the mapping; If it points to

NULL, then $i \rightarrow -1$ is an element. Originally, arity preserving mappings from the *LHS* to the *RHS* are considered, but since all of them also have an arity preserving inverse (Section 2.8) and the original mappings are not needed, only the inverse mappings are stored (more precisely: *one* of them). How this mapping looks like is explained in Section 5.2.4.

CommandList rewriting This variable is used for storing a sequence of instances of class `Command` (see Section 5.2.3), namely the commands that eventually really define the `rewrite`-function. Again, it is filled in the main function `makeCode`.

5.2.2 Constructing Function `match`

In this version, the definition of function `match` is solely created in function `toSource` (unlike in “Compiler/SV”). In fact, matching is again rather straightforward (assume that a is the expression that is attempted to be matched):

1. First, the overall *shape* of a is checked: If there is an n -ary function in the LHS of the rule, is there also an n -ary function with the same name at the same position in a ? If there is a constant in the LHS, is the same constant at the same position in a ?
2. Afterwards it is checked whether multiple occurrences of the same variable are instantiated with equal subexpressions of a .

Now to the implementation of the algorithm that creates the C++ function definition of function `match` (in function `toSource`):

The first part (shape) is implemented in a recursive manner by the function `createMatchingCode` of class `Expression`, which is called on `lhs`. This function requires several arguments, one of them is a list of available `exprArray`-indices: Since the `exprArray`-array replaces local variables in both the `match`- and `rewrite`-function, it is very important to know, at any time, which indices are *available* and which not. An index i is available iff the i -th element of `exprArray` can be set without overwriting a pointer that is still needed.

The second part (variable instances) is implemented by just iterating through all variables that occur more than once and comparing the subexpressions of a at their respective positions. If they are all equal, a is matched by the rule.

However, there is one thing missing that is not done in function `toSource`, but in the main function `makeCode`, namely computing the number of “local” variables of type `Expression*` needed for matching. Since the array `exprArray` is used for

representing those variables, it has to be made clear what *size* this array should have. Function `countMatchingSize` of class `Expression`, called on `lhs`, does the job. Of course the size does not only depend on the definition of function `match` but also on the definition of function `rewrite`.

The following example shows the definitions of the function `match` of all the rules of a given rule set \mathcal{R} :

$\mathcal{R} =$
`Rule[VarList[m, n], *[m, succ[n]], +[*[m, n], m]]`
`Rule[VarList[x], f[x, x], g[x]]`
`Rule[VarList[x, y], f[x, 0, y, g[x]], 0]`

Part of file “Rules.cpp” (automatically generated by the compiler):

```

1  bool Rule0::match(Expression *expr){
2      if (expr->isFunction(Expression::RuleSymbols + 0, 2) &&
3          expr->getArgument(1)->isFunction(Expression::RuleSymbols + 1, 1)){
4          return true;
5      }
6      return false;
7  }
8
9  bool Rule1::match(Expression *expr){
10     if (expr->isFunction(Expression::RuleSymbols + 3, 2)){
11         if (expr->getArgument(0)->equals(expr->getArgument(1))) return true;
12     }
13     return false;
14 }
15
16 bool Rule2::match(Expression *expr){
17     if (expr->isFunction(Expression::RuleSymbols + 3, 4) &&
18         expr->getArgument(1) == Constant::RuleConstants + 0 &&
19         (exprArray[0]=expr->getArgument(3))->isFunction(Expression::RuleSymbols+4,1)){
20         if (expr->getArgument(0)->equals(exprArray[0]->getArgument(0))) return true;
21     }
22     return false;
23 }
```

Note that the content of `Expression::RuleSymbols` is `[* , succ , + , f , g , 0]` and that the only element of `Constant::RuleConstants` is constant 0.

5.2.3 Commands

In C++, the definition of a function is given by a sequence of commands, like assignments, conditions, loops, function calls, etc. Hence, for any rule r , the definition of its `rewrite`-function is also given by such a sequence. Now there are at least two possibilities: Either the commands are immediately written, as C++ code, to the file, or they are first represented as a *list of comands* and then this

list is written to the file as C++ code. In principle, the first approach is not that bad, but the implementation of “Compiler” (and also “Compiler/SV”) uses the second approach, for the following reason: After creating the function definition in terms of a list of commands, this list can be easily manipulated later on. This is especially important if it should be additionally optimized after creation in order to make the function implementation even more efficient. If the commands were immediately written to the file, that task would be rather cumbersome.

In any case, before going into detail and describing *how* such a list of commands is created in Section 5.2.4, it has to be made clear *what* commands there are:

- Setting a variable of type **Expression*** (either an element of **exprArray** or the root of the old expression)
- Setting the name of functions
- Deleting expressions
- Setting arguments of functions
- Creating new functions
- Copying expressions

Obviously there are neither conditions nor loops, but that does not mean that the complexity of **rewrite**-functions is always constant. For example, copying an expression requires every vertex of the corresponding tree to be copied, and thus the complexity clearly depends on the size of that tree. Have a look at Section 8.1 for more information.

The next thing that needs to be explained is *how* those commands are implemented: Once again, there is an abstract base class **Command** all the derived classes inherit from; For each of the previously mentioned commands there is such a derived class. In fact, there are even a bit more, listed below. The motivation for structuring the command-classes in such a way is quite apparent: The commands will be stored in a list, but since the elements of the list are most likely *different* commands, and therefore also have different types, one supertype (**Command**) is needed to put them all into one single list. This is a commonly-used strategy, called type-polymorphism (see [14] for more information).

Note that there is a class **ExprArrayIndex**. This class is very simple and only implemented for convenience. It only consists of the member **i** of type **int**, and its only purpose is the following: Whenever an instance of that class is written to a C++ source file, it is written as **exprArray[i]** if *i* is non-negative, and as **expr**

otherwise. In the `rewrite`-function `expr` denotes the root of the expression that is to be rewritten.

In the following list of commands, whenever a variable, say `index`, is displayed in italic letters, this simply means “value of `index`”. The value of an object of type `ExprArrayIndex` is always either `exprArray[i]` or `expr` (see above).

CAssignment This class consists of two indices `lhs` and `rhs` of type `ExprArrayIndex` (one for the LHS of the assignment, one for the RHS) and a list of non-negative argument-indices `arg` of type `int`. The C++ code corresponding to an instance of this class is the following:

```
lhs = rhs->getArgument(arg[0])...->getArgument(arg[Length(arg) - 1])
```

The effect of executing this command is that afterwards `lhs` points to the expression on the RHS of the assignment.

CSetName This class only consists of one index `index`, again of type `ExprArrayIndex`, and the index of a symbol `nameIndex` of type `int`. The C++ code corresponding to an instance of this class is the following:

```
index->setName(Expression::RuleSymbols + nameIndex)
```

The effect of executing this command is that, if `index` points to a function `f`, the name of `f` is set to the `nameIndex`-th symbol in the list of symbols.

CDelete This class basically consists of one index `lhs` of type `ExprArrayIndex` and one argument-index `arg` of type `int`. The C++ code corresponding to an instance of this class is the following:

```
index->getArgument(arg)->deleteObject()
```

If `arg` is negative, the `getArgument`-part is omitted. The effect of executing this command is simply that the object is deleted.

CNewFunction This class consists of one index `lhs` of type `ExprArrayIndex`, the index of a symbol `nameIndex` of type `int`, and the arity of the new function `arity` of type `int`. The C++ code corresponding to an instance of this class is the following:

```
lhs = new Function(Expression::RuleSymbols + nameIndex, arity)
```

This command creates a new function with arity *arity*, whose name is the *nameIndex*-th symbol in the list of symbols, and makes *lhs* point to it.

CReuseObject This class consists of two indices `lhs` and `index` of type `ExprArrayIndex` and one argument-index `arg` of type `int`. The C++ code corresponding to an instance of this class is the following:

```
lhs->setArgument(index, arg)
```

Executing this command simply sets the *arg*-th argument of the function pointed to by *lhs* to the object pointed to by *index*.

CCopyObject This class consists of the same members as class `CReuseObject`. The C++ code corresponding to an instance of this class is the following:

```
lhs->setArgument(index->copy(), arg)
```

Executing this command basically does the same as the command before, except that the new argument is not the object pointed to by *index* itself, but a copy of it.

CSetConstant This class consists of the same members as class `CReuseObject`. The C++ code corresponding to an instance of this class is the following:

```
lhs->setArgument(Constant::RuleConstants + index, arg)
```

Executing this command sets the *arg*-th argument of the function pointed to by *lhs* to the *index*-th constant in the list of constants.

Now that it should be clear what commands there are and how they are implemented, let's move on and see how command sequences are created.

5.2.4 Constructing Function rewrite

In principle, one could do basically the same as in Version 1: Generate a mapping from the LHS of the rule to its RHS and then, instead of *executing* the three steps of the algorithm (Section 4.1.4), create the command sequence that does the same job when executed. Surely such an approach would work, and indeed this is in principle the approach that is pursued, but still there are two additional improvements.

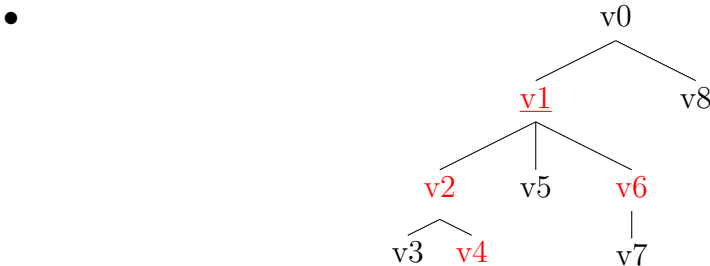
Recall the remark at the end of the second step in the example showing how the algorithm of Version 1 works. There it was mentioned that, given a concrete mapping μ , some parts of the LHS are preserved by μ , which means that each vertex in such a part on the LHS is mapped to exactly the same place in the corresponding part on the RHS. Note that “part” does not have the same meaning as “subtree”: Again, every part has a root, but in contrast to subtrees, not all children of the root have to be in the part as well; It is even possible that no children are in the part. A more formal definition is the following:

Definition 23. Let T be a tree and \mathcal{V} its set of vertices. The set $A \subseteq \mathcal{V}$ forms a *part* of T iff there is a vertex $r \in A$ such that for any other vertex $v \in A$:

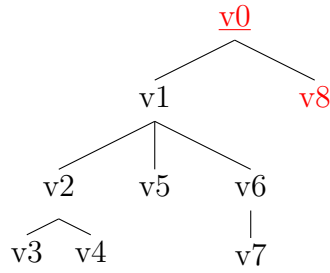
1. v is a successor of r
2. Every vertex on the unique path from r to v is also an element of A

The (uniquely defined) vertex r is called *root* of the part.

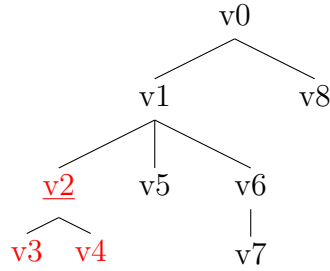
Every subtree is a part, but not vice versa; The following are some examples of parts of one and the same tree (vertices belonging to the part are printed in **red** letters; the root of the part is underlined):



•



•



Before it can be defined what it means that a mapping preserves a part, the notion of *path tuples* has to be defined.

Definition 24. Let T be a tree, v, u vertices such that u is a successor of v or $u = v$. Then there is one unique path P from v to u . This path P has an associated *path tuple*, a tuple of Natural numbers denoted as $\text{pt}(P)$, which is given as follows:

1. If $v = u$ then $\text{pt}(P) = \langle \rangle$
2. Otherwise v has to be a node and u has to be either a successor of its i -th child or the i -th child itself, for some $i \geq 0$. Let P_i denote, in any case, the path from the i -th child to u . Then $\text{pt}(P) = \langle i \rangle \smile \text{pt}(P_i)$, where “ \smile ” stands for concatenation.

Definition 25. Let T_1, T_2 be trees, A a part of T_1 with root r , and μ an arity preserving mapping from T_1 to T_2 . A is *preserved* by μ iff

1. No vertex in A is mapped to \emptyset
2. The images of the elements in A form a part in T_2 with root $\mu(r)$
3. For any $v \in A$ let P_1 be the path (in T_1) from r to v and P_2 the path (in T_2) from $\mu(r)$ to $\mu(v)$. Then $\text{pt}(P_1) = \text{pt}(P_2)$

If a part is preserved by a mapping μ this only means that the image of that part is again a part with the same shape. A vertex v being in such a part simply means that $\mu(v) \neq \emptyset$, or in other words, v is reused.

Obviously, whenever there are parts of the LHS that are preserved by a concrete mapping, it would suffice to only move the root to its new place - The rest of the part would then be automatically at the correct place as well, if the edges between the vertices were not forgotten as in “Interpreter” (edges originating from a node correspond to the pointers of a function that point to its arguments). Exploiting this fact saves operations when actually transforming the tree, which of course makes the rewriting process faster.

Improvement 1 Given the mapping μ that describes which objects are reused in which way, detect parts of the LHS that are preserved by μ and do not perform (more precisely: create) redundant operations/commands.

Improvement 1 saves operations and thus decreases the execution time. Hence, the more often it can be applied (the more vertices there are that belong to a part of the LHS with cardinality of at least 2 that is preserved by μ), the better. However, *how* often it can be applied depends of course on μ . This observation immediately leads to the second improvement.

Improvement 2 From all arity preserving mappings from the LHS of a rule r to its RHS, choose the one where Improvement 1 can be applied most often, i. e. where the *total number of operations* required for applying r is minimal when Improvement 1 is taken into account.

Those operations/commands that are mentioned several times above are of course the ones defined in Section 5.2.3.

5.2.4.1 Implementation of Improvement 2

Improvement 2 is implemented in a brute-force way: Simply try out *all* arity preserving mappings from the LHS to the RHS of the rule, for each mapping compute the total number of operations required for rewriting an expression following that mapping and taking into account Improvement 1, and eventually choose the mapping where the number of operations is minimal. This algorithm is implemented by the function `findBestMapping`, which recursively goes through all the arity preserving mappings and computes the number of operations by calling the two functions `countCopyingOps` and `countMissingOps` of class `Expression` and then the function `findBestPaths` of class `Rule` (Section 5.2.4.3). Please note that the operations themselves are *not* created (in terms of a command sequence), but only their *number* is computed!

Now it is also easy to see why it is the arity preserving mappings that are considered: Recall from Section 2.8.1 that arity preserving mappings “ignore” constants (\rightarrow There is only one instance of class `Constant` for each constant) and map functions to functions with the same arity (\rightarrow Number of arguments of `Function`-instances are fixed in the implementation of “Compiler”).

5.2.4.2 Implementation of Improvement 1

Improvement 1 is implemented in a straight-forward way: Given the `lhs`- and `rhs`-tree of a rule together with a mapping it is rather easy to find out which arguments of a concrete function f are mapped to arguments at the same place of the image of f . For all such arguments the algorithm is applied recursively until there are no more. The union of all those vertices surely forms a part that is preserved. Improvement 1 is implemented several times:

First, it is implemented in functions `countCopyingOps` and `countDeletingOps` of class `Expression` that call each other recursively and compute (one part of) the number of required operations. Depending on where a vertex is located in the LHS (root of a part that is preserved, root of a part that is not preserved, not root of a part that is preserved, not root of a part that is not preserved), different values for that number are returned.

Second, it is implemented in function `fillEAI` of class `Expression`. This function modifies the meta-information of the `lhs`-tree stored in the `exprArrayIndex`-array of class `Rule`; Consult Section 5.2.4.4 for more information about `exprArrayIndex`.

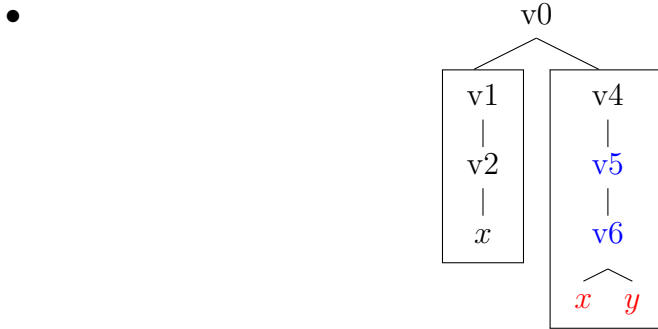
Third, it is implemented in functions `createCopyingCommands` and `createCreatingCommands` of class `Rule` that eventually really generate the command sequence and hence also need to take into account Improvement 1.

5.2.4.3 Function `findBestPaths`

This function covers one minor but nonetheless important detail of computing the number of required operations, given a mapping μ . Assume that in the LHS of the rule there is a variable x that is not the root of any part that is preserved by μ , and x occurs more often on the RHS of the rule than on its LHS. Thus, x , or more precisely its instance, needs to be copied at least once when applying the rule and creating the new expression. However, since x is not the root of any preserved part, its instance will possibly never be accessed (stored somewhere in `exprArray`) in the function `rewrite`, since this is exactly the purpose of Improvement 1! So, if x ’s instance (more precisely: the root of it) is not stored in `exprArray`, how can

it be copied then? Surely, the positions of its instances are known (there might be more than one instance, which are of course all equal), so whenever x needs to be copied the algorithm could simply traverse the tree until it reaches such an instance and copy it. Still, this is rather inefficient, especially if x needs to be copied more than once. So, why not just store one of its instances in `exprArray`? Indeed, this is the way how it is done. But now immediately the next question arises: *Which* instance, if there are more than one? The answer seems to be obvious: Choose the one that is nearest to the root of the preserved part it is included in, since it can be reached in shortest time then. However, the actual answer is not that easy: Assume that there is not only one such variable, but several. The best way of choosing the instances that are to be stored in `exprArray` is then the following: Minimize the total length of the paths leading from the roots of the respective preserved parts to those instances, where all parts of these paths that are shared by multiple paths are counted only once.

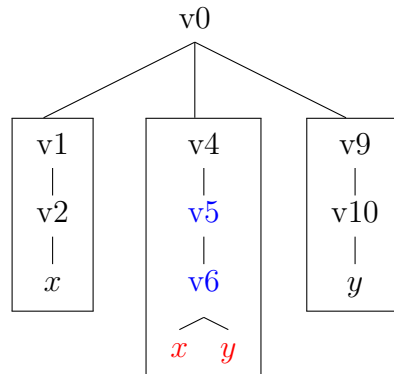
The following examples illustrate the idea from above. The trees always represent the LHS of a rule, and the two variables x and y always occur more often on the RHS of the corresponding rule and are never the root of a preserved part; Preserved parts have a frame around them. The occurrences that are eventually chosen for being stored are printed in **red**; All vertices on the path to such an occurrence that are not automatically stored in `exprArray` are printed in **blue**.



Total length = number of blue vertices = **2**

If the first instance of x was chosen, the total length would be 3, although this instance occurs somewhere nearer to the root.

•



Total length = 2

In all other cases, the total length would at least be 2 as well, although there are occurrences of both x and y that are nearer to the root.

In function `findBestPaths` again a brute-force strategy is pursued: For every variable try all occurrences and count the number of vertices that have to be additionally accessed. In the end, that number is added to the number of operations.

5.2.4.4 The `exprArrayIndex-Array`

As already mentioned before, the meaning of an element of the `exprArrayIndex-array` varies depending on its position in the `makeCode`-method. Basically, there are two different meanings:

Right after computing the best mapping and before calling `fillEAI` on `lhs`, the meaning is the following (recall that every element of `exprArrayIndex` is of type `int` and corresponds to a vertex v in the `lhs`-tree):

- 0: v is a constant; Constants can be ignored in the whole process, since there is only one instance of class `Constant` for every constant, which means that they do not have to be considered when finding the best mapping (arity preserving mappings map constants to \emptyset).
- -1: v does not have to be stored in `exprArray`, because it is in a preserved part (and not the root of that part)
- -2: v needs to be stored permanently in `exprArray`, because it is the root of a preserved part
- -3: v needs to be accessed and thus stored temporarily, because although it is in a preserved part (and not root of that part), some successor of it needs to be accessed

- -4: v is a node that has to be deleted (not in a preserved part) and thus needs to be stored temporarily, because in this case surely all of its children need to be accessed
- -5: v is no node and deleted and thus does not have to be stored
- -6: v is the root of a preserved part and mapped to the root of the RHS
- -7: v is a variable that needs to be stored because of the idea explained in Section 5.2.4.3
- -8: v needs to be stored permanently *only* because some successor of it is a variable where `exprArrayIndex` is -7
- -9: v is a function that needs to be stored permanently, because its name has to be changed

There is a difference between an object that is stored *permanently* and an object that is stored only *temporarily*: As in Version 1, the rewriting-function can be structured into steps: In the first step, pointers are created that point to some objects, and some other objects are deleted (subsumes the first and the third step of Version 1), and in the second step the new expression is created. If a vertex is stored permanently this means that it needs to be accessed in the second step; If it is only stored temporarily this means that some successors have to be accessed in the first step, but afterwards it is not needed any longer. That difference is particularly important, because whenever a temporarily stored object does not have to be accessed any more, the corresponding element in `exprArray` can be used for something else.

When the function `fillEAI` is called on `lhs`, with `exprArrayIndex` as its argument, the meaning of an element of the array changes. Still, for any index i let v be the corresponding vertex in the `lhs`-tree:

- $i = 0$: v does not have to be stored in `exprArray` at all
- $i = -1$: v is mapped to the root of the RHS
- $i > 0$: v is the root of a preserved part and has to be stored in `exprArray` at index $i - 1$
- $i < -10$: v is not the root of a preserved part (yet it *is* in a preserved part) and has to be stored in `exprArray` at index $-i - 11$

As can be seen, `exprArrayIndex` now does not only contain information whether a vertex has to be stored or not, but if so also *where*. The fact that temporarily stored objects do not have to be pointed to all the time (can be “forgotten” by `exprArray`) is taken into account: As soon as that happens, the corresponding index of the `exprArray`-array is immediately made available for the next object that needs to be stored, which of course saves some memory when executing the compiled program.

5.2.4.5 Generating the Command Sequence

As already mentioned above, the `rewrite`-function always consists of two steps. In the first step, the `exprArray`-array is filled with pointers to some objects and all objects that are not reused are deleted. In the second step, the new expression is eventually created.

Functions `createCopyingCommands` and `createDeletingCommands` of class `Expression` generate the commands that implement the first step. They do so by going through the `lhs`-tree and, for every vertex, mainly depending on the value of the corresponding element of `exprArrayIndex`, adding one instance of one of the two command-classes `CAssignment` (for storing in `exprArray`) and `CDelete` (for deleting) to the command sequence.

Function `createCreatingCommands`, again of class `Expression`, generates the commands that implement the second step. It does so by going through the `rhs`-tree and, for every vertex, mainly depending on the value of the corresponding element of `bestInvMap`, adding one instance of some of the other derived classes of class `Command` to the command sequence.

5.2.4.6 Optimizing the Command Sequence

Optimizing the command sequence is one of the very last things that are done in function `makeCode`. There are two things that are optimized a little bit:

- Chains of assignments are put together into one assignment, which means that if the command sequence contains, for example, the subsequence (in C++ code)

```
exprArray[0] = expr->getArgument(0);  
exprArray[0] = exprArray[0]->getArgument(1);  
exprArray[0] = exprArray[0]->getArgument(2)
```

that does nothing else than making the first element of `exprArray` point to the third child of the second child of the first child of the root of the old expression, then this sequence is replaced by the one and only command

```
exprArray[0] =  
expr->getArgument(0)->getArgument(1)->getArgument(2);
```

that does exactly the same. Obviously, the time complexity of the function is *not* reduced by this optimization.

- Index-holes of `exprArray` are filled, which means that if, say, the third element of `exprArray` is not used within the entire definition of function `rewrite`, but the fourth element is, then instead of using the fourth element the third one is used. Thus, in every occurrence of `ExprArrayIndex` with the value of its member `i` equal to 3 (the fourth element) in the command sequence, the value of `i` is replaced by 2 (the third element). Obviously, doing this eventually saves memory, because the size of `exprArray` is reduced. Note that the requested size of `exprArray` for function `rewrite` is computed right when performing this optimization.

5.3 How To Use “Compiler”

The name of this version already indicates what it is expected to do. It is expected to work as a *compiler*, which means it should produce programs with certain properties, and indeed, this is exactly what it does. Given a rule set, it automatically generates C++ source- and header files that implement just those rules. If the files are then compiled with an external C++ compiler, the result is a program *P* working quite similar as “Interpreter” (Version 1), apart from the fact that, of course, no rule set needs to be specified any more after starting it, because its implementation already depends on such a rule set. The reason behind this approach is to make the rewriting process even faster - successfully, as can be seen in Chapter 7.

The compiler itself works quite simple. It has to be executed with at least one command line argument, which, just like in the previous version, has to be the name and path of the file containing the rule set that shall be processed. For this rule set the two files “rules.h” and “rules.cpp” are then generated and stored in the directory “source”, located in the same directory where the compiler is located, too. Together with the other files in that directory they form the C++ source of the program *P* described above; However, all other files are left unchanged,

because they are independent of the concrete rule set.

As already mentioned above, the compiler might be called with even more than only one command line argument. In fact, it is possible to provide the C++ compiler together with some arguments (no source- and target files) to the rewriting compiler, in order to immediately compile the C++ files and get the desired program P , whose name is then the same as the one of the rule set, and which is stored in the same directory as the rule set. But of course it is also possible to first generate the files and then call the C++ compiler separately.

The generated programs work very similar to “Interpreter” (see Section 4.2). The only differences are:

- After starting them, no rule set needs to be specified any more
- Some of the special commands do not work here; They are marked with “*” in the appendix (Appendix A)
- They may be called with at most one command line argument; In this case, the argument has to be an input file (Section 4.2.2)

5.4 An Example

This example illustrates both the behavior of the compiler and the behavior of a compiled program. Therefore let the rule set \mathcal{R} consist of the one and only rule

`Rule[VarList[x,y], f[g[g[x]],f[g[y],x]], f[f[x,f[0,x]],h[g[x]]]`

The tree representation of the LHS and RHS of the rule is shown in Figure 5.1.

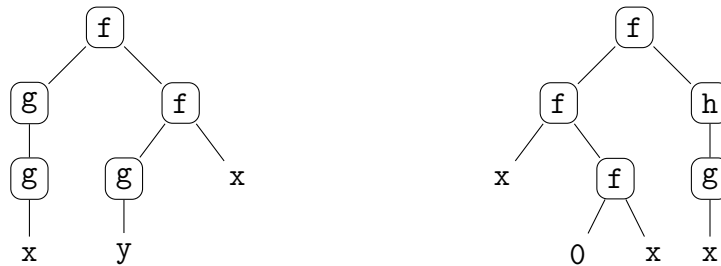


Figure 5.1: The LHS and RHS of the rule, represented as trees

There are in total 216 arity preserving mappings from the LHS to the RHS of the rule (3 for the first binary function, 2 for the second binary function, 3 for

the first unary function, 2 for the second unary function, 1 for the third unary function, 3 for the first occurrence of x , 2 for the second occurrence of x , 1 for y ; see Section 2.8.3). The best of those mappings w.r.t. the number of operations, as computed automatically by the compiler, is given by

$$\{0 \rightarrow 0, 1 \rightarrow 6, 2 \rightarrow 7, 3 \rightarrow 8, 4 \rightarrow 3, 7 \rightarrow 5\}$$

A graphical representation of the mapping is shown in Figure 5.2.

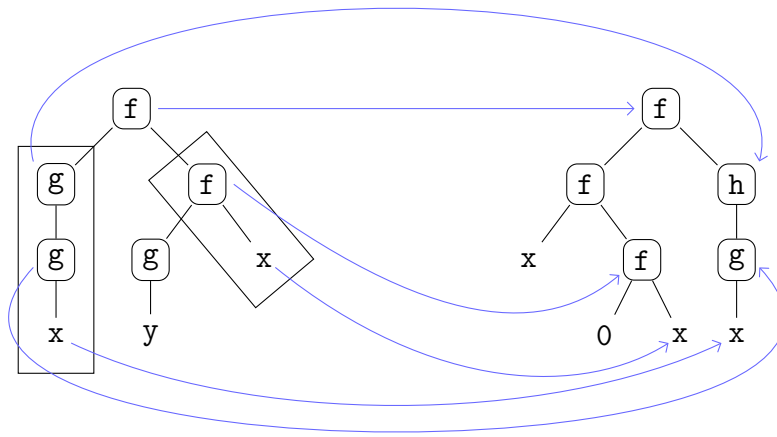


Figure 5.2: The best mapping, indicated by blue arrows - Preserved parts with at least two elements are framed

The content of `exprArrayIndex`, both *before* and *after* calling function `fillEAI`, is shown in Figure 5.3. Instead of presenting the flat array, the complete tree representation of the LHS is shown where the names of the vertices are replaced by the value of the corresponding element of `exprArrayIndex`. Please note that there are two vertices with `exprArrayIndex` equal to -13; This is due to the fact that the first one only needs to be stored temporarily, hence the index is again available afterwards.

The length of the resulting command sequence, as computed *before* the commands themselves have been generated, is 13:

- Concerning the first step in function `rewrite` (setting pointers, deleting objects): 4 commands for storing the 4 vertices whose `exprArrayIndex` after executing `fillEAI` is either > 0 or < -10 , and 2 commands for deleting the objects that are not needed any more (`exprArrayIndex` before executing `fillEAI` is equal to -4 or -5)



Figure 5.3: `exprArrayIndex` right after finding the best mapping (left) and after executing function `fillEAI` (right)

- Concerning the second step (creating the new expression): 2 commands for creating the binary function that is not available in the old expression and moving it to the right place, 1 command for copying the instance of variable `x` and moving it to the right place, 2 commands for moving the vertices with positive `exprArrayIndex` (again after executing `fillEAI`) to the correct place, 1 command for setting the constant 0, and 1 command for changing the name of one of the unary functions into `h`

The following is the resulting C++ source code, where each line in the definition of the function (apart from the last one) stands for exactly one command of the command sequence:

```

1 Expression *Rule0::rewrite(Expression *expr){
2     exprArray[0] = expr->getArgument(0);
3     exprArray[1] = expr->getArgument(1);
4     exprArray[2] = exprArray[1]->getArgument(0);
5     exprArray[2]->getArgument(0)->deleteObject(false);
6     exprArray[2]->deleteObject(true);
7     exprArray[2] = exprArray[1]->getArgument(1);
8     exprArray[3] = new Function(Expression::RuleSymbols + 0, 2);
9     expr->setArgument(exprArray[3], 0);
10    exprArray[3]->setArgument(exprArray[2]->copy(), 0);
11    exprArray[3]->setArgument(exprArray[1], 1);
12    exprArray[1]->setArgument(Constant::RuleConstants + 0, 0);
13    exprArray[0]->setName(Expression::RuleSymbols + 3);
14    expr->setArgument(exprArray[0], 1);
15    return expr;
16 }

```

Without taking into account the very last instruction that only returns the new expression, the definition indeed contains exactly 13 commands. Please note that the content of array `Expression::RuleSymbols` is `[f, g, 0, h]`, the only element of `Constant::RuleConstants` is constant 0, and that `expr` points to the root of the tree representing the expression that is to be rewritten. The size of `exprArray` obviously is 4 (unless function `match` needs even more elements, which is not the case in this example). In lines 5 and 6 there are two different ways of deleting objects. In line 5, where the argument of the function `deleteObject`

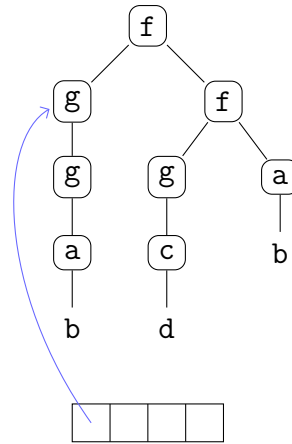


Figure 5.4: After executing line 2: `exprArray[0] = expr->getArgument(0)`

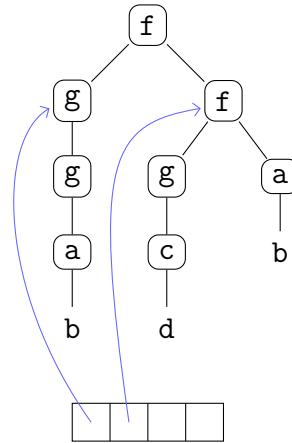


Figure 5.5: After executing line 3: `exprArray[1] = expr->getArgument(1)`

is the boolean value **false**, not only the object itself, but also the entire subtree originating at the object is deleted. In line 6, only the object is deleted.

Figures 5.4 to 5.16 illustrate the execution of the function line-by-line when rewriting the expression `f[g[g[a[b]]],f[g[c[d]],a[b]]]` (that is obviously matched by the rule), where the upper part of each figure is dedicated to the expression *as it is in the respective execution step*, and the lower part shows the `exprArray`-array and where its elements point to.

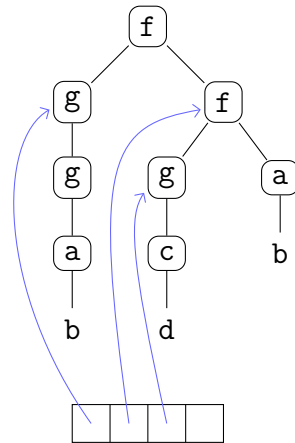


Figure 5.6: After executing line 4:
`exprArray[2] = exprArray[1]->getArgument(0)`

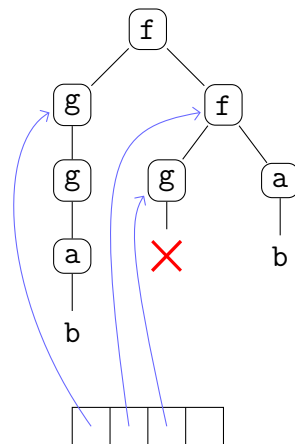


Figure 5.7: After executing line 5:
`exprArray[2]->getArgument(0)->deleteObject(false)`

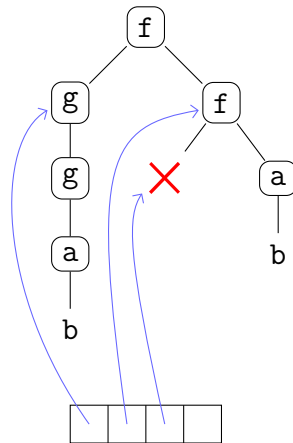


Figure 5.8: After executing line 6: `exprArray[2] -> deleteObject(true)`

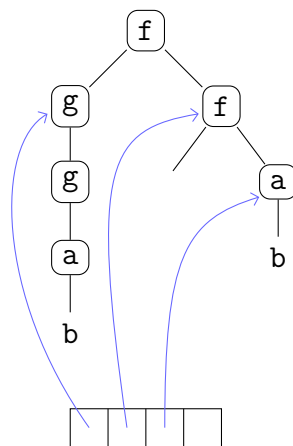


Figure 5.9: After executing line 7:
`exprArray[2] = exprArray[1] -> getArgument(1)`

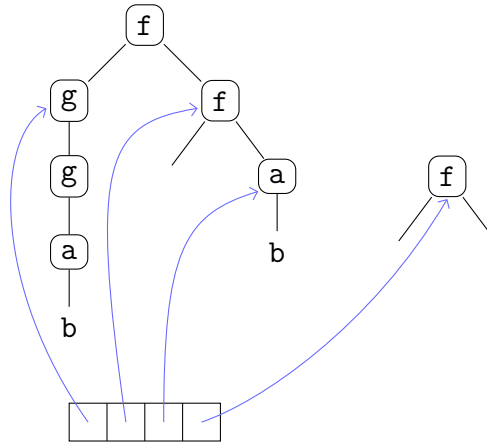
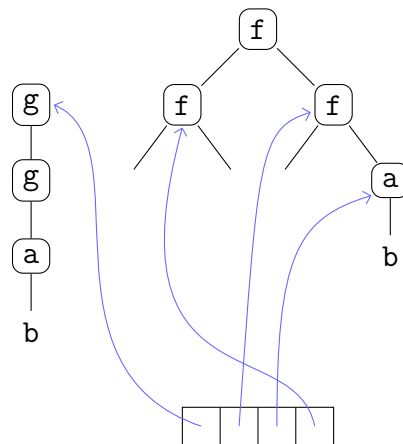


Figure 5.10: After executing line 8:

```
exprArray[3] = new Function(Expression::RuleSymbols + 0, 2)
```

Figure 5.11: After executing line 9: `expr->setArgument(exprArray[3], 0)`

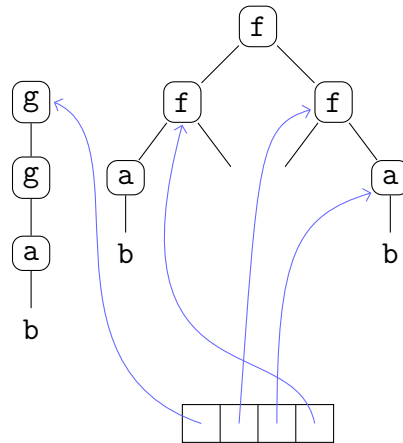


Figure 5.12: After executing line 10:
`exprArray[3] -> setArgument(exprArray[2] -> copy(), 0)`

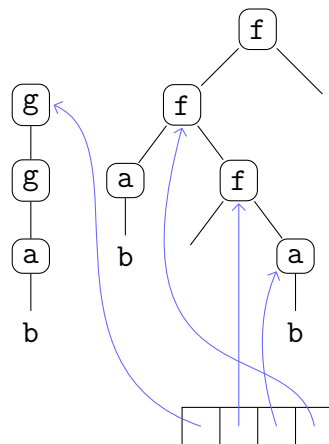


Figure 5.13: After executing line 11:
`exprArray[3] -> setArgument(exprArray[1], 1)`

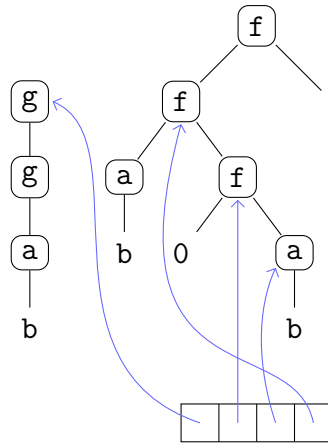


Figure 5.14: After executing line 12:

```
exprArray[1] -> setArgument(Constant::RuleConstants + 0, 0)
```

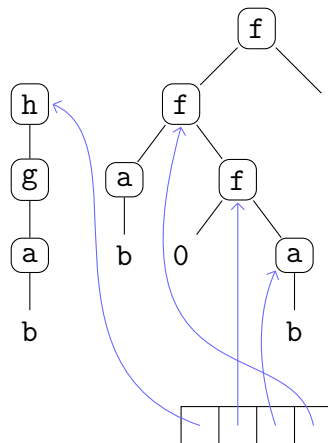
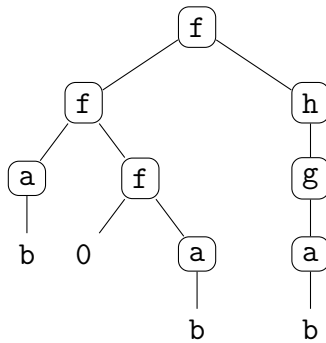


Figure 5.15: After executing line 13:

```
exprArray[0] -> setName(Expression::RuleSymbols + 3)
```

Figure 5.16: After executing line 14: `expr->setArgument(exprArray[0], 1)`

Chapter 6

Version 3: The Compiler with Sequence Variables

The very last version that has been created, in the following called “Compiler/SV”, is very similar to the one before, “Compiler”, with one little difference: It is also capable of dealing with sequence variables in the rewrite-rules. This little difference, however, has huge impact on the algorithms and therefore also on the implementation of the programs (both compiler and compiled programs), as will be seen in the subsequent sections.

The outline of this chapter is the following:

First, the main algorithms and their implementation are presented, as well as the most important classes and their hierarchy in the implementation. Just like in Version 2, “Compiler”, also in this version it is important to distinguish between the implementation of the compiled programs, which is dealt with first, and that of the compiler. There, not only the algorithms for *rewriting*, but also for *matching* are presented in detail, since they are far more complicated than those in the other two versions.

Afterwards, a short description of how the compiler and the compiled programs are used is provided.

The chapter ends with an example that illustrates what exactly the output of the compiler is and how a given expression is rewritten by it (after being transformed into an executable program, of course).

6.1 Implementation of the Compiled Programs

In principle, the implementation of the compiled programs in this version is quite the same as in “Compiler”. There is one abstract base class `Rule` and, for every rule, a derived class that provides definitions for the two virtual functions `match` and `rewrite`. The only difference here is that function `rewrite` is now of type `void` rather than of type `Expression*`; The argument (`expr`) is passed as a reference and points to the root of the tree representing the new expression after the execution of the function.

However, class `Rule` also contains three additional members, listed below together with some short information about their meaning.

int *seqVarLen This is a *non-static* array of underlying type `int` whose length varies from rule to rule and is exactly the number of sequence variables occurring in the rule. It is filled in function `match` and, in case that an expression is matched by the rule, the value of the *i*-th element of `seqVarLen` describes *how long* the sequence of instances of the *i*-th sequence variable has to be in order to match the expression. This information has to be stored, because it probably will be needed when eventually rewriting the expression.

int argNumber This static member is only needed as an auxiliary variable in some of the definitions of the `match`-functions.

Function setInt This static member is a very simple function that takes two arguments of type `int`, both of them as references, assigns the value of the second one to the first one and in the end always returns the boolean value `true`. It is only needed to make the definitions of the various `match`-functions a little bit shorter.

The most important difference between “Compiler” and “Compiler/SV” comes from the fact that some rule-variables may be instantiated with sequences of subexpressions. Therefore, additional flexibility is needed in terms of the arity of functions. It is better not to implement functions with a fixed arity, where the arguments are stored in an array, but rather with a flexible arity where the arguments are stored in a linked list, and where arguments can be added and removed at any time with very little effort (compared to the effort of allocating new memory when resizing an array). Indeed, class `Function` which implements functions does not have a list of expressions as its member, but rather only contains pointers to its first and last argument (might be the same; might be even `NULL`,

if the function has no arguments), called **first** and **last**. In addition to that, *every* object (functions and constants) contains a pointer, again of underlying type **Expression**, that points to the object's right neighbor in the argument list, or to NULL if there is no such right neighbor. This pointer is called **next** and, since every object has one, is a member of the base class **Expression**.

The fact that every object points to its right neighbor has one immediate consequence: Since one and the same constant might appear several times in an expression and therefore surely also has different right neighbors, it necessarily needs to be represented by different instances of class **Constant**, where the **next**-member of every instance points to the respective right neighbor. So, unlike in the two previous versions, there is no list of constants any more; All occurrences of a constant are represented by different objects, just like functions.

The example below (Figure 6.1) illustrates the new concept of storing arguments by showing the implementation of the tree representing the expression `f[a,g[b],g[c]]`:

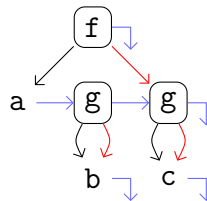


Figure 6.1: Internal representation of expression `f[a,g[b],g[c]]`: Black arrows indicate values of **first** pointers, red arrows indicate values of **last** pointers, blue arrows indicate values of **next** pointers; Arrows having a corner indicate NULL-pointers

Apart from the things mentioned above, everything remains the same compared to “Compiler”.

6.2 Algorithms and Implementation of the Compiler

The first thing that has to be said when talking about the implementation of the compiler is that just as in all other programs, except the compiled programs of this version, again function arguments are stored in an array rather than as a linked list. This is due to the fact that the compiler only needs to represent LHSs and

RHSs of rules, and although there might be sequence variables involved, they are still only individual symbols, which means that the arity of functions can be fixed. In fact, sequence variables affect the implementation of the compiler in only three ways:

1. Since the arity of functions is flexible and multiple occurrences of the same constant are represented by different objects (in the compiled programs), not arity preserving mappings (as with “Compiler”), but type preserving mappings have to be considered: They do not “ignore” constants, but they “ignore” a function’s arity. This in particular means that a function with arity, say, 3, may be reused as a function with arity, say, 2.
2. Multiple occurrences of one and the same constant are represented by different objects, just as in the compiled programs, because when dealing with type preserving mappings, constants have to be considered as well, i. e. it does matter where exactly an occurrence of a constant is mapped to (constants are not “ignored”, i. e. not mapped to \emptyset).
3. The matching-algorithm becomes much more complicated, since there might be not only one variable substitution that makes the LHS of the rule and the expression equal, but many. Thus, generating the code for the `match`-function is in general far more difficult.

Concerning the implementation of class rule, please consult Section 5.2.1, since it is almost completely the same in this version: There are functions `toHeader` and `toSource` that return the C++ source code of the class declaration and -definition, respectively, of the respective rule. Function `makeCode` is again the main function that really generates the command sequences for the `rewrite`- and, unlike in Version 2, also for the `match`-function. Functions `findBestMapping` and `findBestPaths` have the same meaning as before, apart from the fact that the best mapping obviously needs to be type preserving in this case.

6.2.1 Commands

As shortly mentioned above, also the definitions of the `match`-functions are represented as command sequences. Therefore, before explaining how those definitions (and of course the definitions of the `rewrite`-functions, too) are generated and how they look like, it has to be made clear what commands there are. Some of them are the same as or very similar to commands of Version 2, see Section 5.2.3 for more information. The implementation was done in completely the same way by providing an abstract base class `Command`, where all other classes inherit from.

However, there is one important thing to notice. Unlike in Version 2, function arguments are not stored in an array but somehow as a linked list. So, accessing them cannot be done in constant, but only in linear time. This means that one has to pay attention to the *order* in which arguments are eventually accessed. For example, if there is a function with two arguments a_1 and a_2 at positions i and j in the argument list, respectively, that still have to be dealt with (in whatever way), and it does not matter which of them has to be processed first, then always the *first* one in the argument list should be processed first, say this is a_1 ($\Rightarrow i < j$). The reason is that accessing a_1 requires $\mathcal{O}(i)$ steps for traversing the argument list from the beginning, but then for accessing a_2 one does not have to start at the beginning of the list but at a_1 , which then only takes $\mathcal{O}(j - i)$ steps compared to $\mathcal{O}(j)$ steps.

There are commands whose only purpose is to move a pointer p through argument lists, namely `CMoveRight` and `CMoveAdvanced`. They do so by successively setting $p = p \rightarrow \text{next}$ (recall member `next` of class `Expression`). Also, whenever there is talk of something like “sequence of objects starting at obj ”, where obj is of type `Expression`, this simply means all the objects that can be accessed via $obj \rightarrow \text{next} \rightarrow \dots \rightarrow \text{next}$.

Please note that there is one important additional member of the base class, compared to the previous version: Every command now has a type, returned by the function `commandType`, which is one of the following:

- Normal command: This command cannot be used inside a condition, but only in a sequence of commands
- Normal/conditional command: This command can be used both inside a condition and as a normal command in a sequence of commands
- Conditional command: This command can be used only inside a condition (and hence returns a boolean value)
- Loop command: This command represents a `for`-loop

This short list already indicates that in Version 3 there are also conditions and loops. All derived classes are listed below, together with a short explanation. The naming and typesetting conventions are the same as in Section 5.2.3, with three additional shortcuts:

Sometimes some function arguments have to be accessed. This is as usual done by providing the indices of the respective arguments, but what if those indices are not known at compile-time? It could be the case that there are some sequence variables *before* the requested arguments, and the length of their sequences of instances is

obviously not known. The solution is somehow simple: Although those lengths are not known at compile-time, they are known at runtime (array `seqVarLen`). Hence, let in the following n be the total number of sequence variables in the rule (might of course be 0). If a is an `int`-array of length $n + 1$ and a member of one of the commands, and in the given C++ source code $\langle a \rangle$ appears, then this is a shortcut for

$$a[0]*seqVarLen[0]+a[1]*seqVarLen[1]+\dots+a[n-1]*seqVarLen[n-1]+a[n]$$

if $a[n]$ is non-negative, and

$$-(a[0]*seqVarLen[0]+a[1]*seqVarLen[1]+\dots+a[n-1]*seqVarLen[n-1]-a[n])$$

otherwise. Note that arguments can be accessed with negative indices as well, which simply means that accessing them is done by starting from the last argument (index -1 refers to the last argument in the argument list).

The second additional shortcut is closely related to the first one. Sometimes instances of sequence variables have to be copied or removed, inserted, etc., to an argument list. Therefore, the length of those instances again needs to be known. So, if m is a member of type `int` of one of the command-classes, and in the given C++ code somewhere $[m]$ appears, then this is a shortcut for

$$m$$

if m is positive, and

$$seqVarLen[-m]$$

otherwise.

Finally, the third additional shortcut is the following: Recall the static `argNumber`-member of class `Rule` in the implementation of the compiled programs (Section 6.1). Whenever there is an `int`-member m of one of the command-classes and in the given C++ source code somewhere $\{m\}$ appears, then this is a shortcut for

$$m$$

if m is non-negative, and

$$argNumber$$

otherwise.

CAssignment This class consists of two indices `lhs` and `rhs` of type `ExprArrayIndex` (one for the LHS of the assignment, one for the RHS) and one non-negative argument-index `arg` of type `int`. The C++ code corresponding to an instance of this class is the following:

$$lhs = rhs \rightarrow \text{getArgument}(arg)$$

The effect of executing this command is that afterwards `lhs` points to the expression on the RHS of the assignment. However, if it is used inside a condition, it does the same but afterwards always returns `true`.

CSetToZero This class can be viewed as a simplification of **CAssignment**. It only consists of one non-negative index called `index`, and the C++ code is the following:

$$index = 0$$

Obviously, the effect of executing this command is that afterwards `index` is a NULL-pointer.

CSetName Completely the same as in the previous version, see Section 5.2.3.

CDelete This class only consists of one index called `index` of type `ExprArrayIndex`, and the corresponding C++ code is the following:

$$\text{delete } index$$

CRemove_Delete This class consists of one index called `parent` of type `ExprArrayIndex`, two numbers called `posIndex` and `length` of type `int`, and a flag called `remove` of type `bool` that determines whether the objects that are considered have to be deleted or just removed from an argument list. The C++ code reads as

$$parent \rightarrow \text{deleteArguments}(posIndex, [length])$$

if *remove* is **false**, and

```
parent->removeArguments(posIndex, [length])
```

otherwise. The effect of executing this command is that, starting *after posIndex*, [*length*] arguments are deleted/removed from the argument list of *parent*.

CNewFunction_Constant This class consists of one index called **lhs** of type **ExprArrayIndex**, a non-negative name-index **nameIndex** of type **int**, and a flag **isConstant** of type **bool**. The corresponding C++ code reads as

```
lhs = new Function(Expression::RuleSymbols + nameIndex)
```

or

```
lhs = new Constant(Expression::RuleSymbols + nameIndex)
```

respectively, and its execution simply creates a new function/constant with the name defined by *nameIndex*. Since the arity of functions is flexible, it does not have to be given when a function is created (it is zero at the beginning).

CCopy This class consists of two indices **lhsIndex** and **rhsIndex** of type **ExprArrayIndex** and one member of type **int** called **len**. The C++ code representation is the following:

```
Expression::Copy(rhsIndex, [len], lhsIndex)
```

The effect of executing this command is that afterwards *lhsIndex* is the first object of the copy of [*len*] objects, starting from *rhsIndex*, i.e. *rhsIndex* has a **next**-object, which itself also has a **next**-object, and so on, and the first [*len*] of them are copied such that the copies again point to each other, and *lhsIndex* is the first of them.

CInsert This class consists of two indices of type `ExprArrayIndex`, called `parent` and `arg`, and two members of type `int` called `pos` and `len`. The corresponding C++ code is

```
parent->insertArguments(pos, arg, [len])
```

The effect of this command is that the first $[len]$ objects starting with *arg* are inserted to the argument list of function *parent* right after argument *pos*, if this pointer is non-NULL, or at the very front otherwise.

CSwap_Move This class consists of one `ExprArrayIndex` called `parent`, three members of type `int` called `pos` (non-negative), `len1` and `len2`, one `int`-array of length $n + 1$ called `dist` and one flag `swap` of type `bool`. The corresponding C++ code is

```
parent->swapArguments(exprArray[pos], [len1], <dist>, [len2])
```

if *swap* is `true`, and

```
parent->moveArguments(exprArray[pos], <dist>, [len2])
```

otherwise. In the first case, the $[len1]$ arguments after (exclusive) the one `exprArray[pos]` points to are swapped with the $[len2]$ arguments after (inclusive) the one that comes $\langle dist \rangle + [len1]$ positions after `exprArray[pos]`. In the second case, the $[len2]$ arguments after (inclusive) the one that comes $\langle dist \rangle$ positions after `exprArray[pos]` are moved to the front such that afterwards they come immediately after `exprArray[pos]`. If `exprArray[pos]` is a NULL-pointer then those arguments are moved to the very front. In both cases, `exprArray[pos]` is moved $[len2]$ positions to the right in the end.

CIisFunctionOrConstant This class consists of one `ExprArrayIndex` called `obj`, one non-negative `int`-member `name` and one flag `isF` of type `bool`. The corresponding C++ code is

```
obj->isFunction(Expression::RuleSymbols + name)
```

if *isF* is **true**, and

```
obj->isConstant(Expression::RuleSymbols + name)
```

otherwise. This command is a conditional command (returns a boolean value); It checks whether *obj* is a function/constant whose name is the *name*-th one in the list of symbols.

CHasArity This class consists of one **ExprArrayIndex** called *obj*, one **int**-member **arity** and one flag **exact** of type **bool**. The corresponding C++ code is

```
obj->argumentCount() == {arity}
```

if *exact* is **true**, and

```
obj->argumentCount() >= {arity}
```

otherwise. This command is a conditional command; It checks whether *obj* is a function whose arity is either equal to {*arity*} or at least {*arity*}, respectively.

CDividesArity This class consists of one **ExprArrayIndex** called *obj* and one positive **int**-member **modulus**. The corresponding C++ code is

```
!((obj->argumentCount() - argNumber) % modulus)
```

This command is again a conditional command; It checks whether the division of the arity of function *obj* by *modulus* has remainder *argNumber*.

CAssignSeqVarLen This command consists of one **ExprArrayIndex** called *obj* and two **int**-members called **index** and **divisor**. The corresponding C++ code is

```
seqVarLen[index] = (obj->argumentCount() - argNumber) / divisor
```

if *divisor* is greater than 0, and

$$\text{seqVarLen}[\text{index}] = 0$$

otherwise. The effect of executing this command is that afterwards the value of the *index*-th element of the `seqVarLen`-array is set to the value on the RHS of the assignment.

CAssignArgNumber This class only consists of an `int`-array of length $n+1$, called `argNumber`. Its C++ code representation is the following:

$$\text{setInt}(\text{argNumber}, \langle \text{argNumber} \rangle)$$

if it occurs inside a condition, and

$$\text{argNumber} = \langle \text{argNumber} \rangle$$

otherwise. Obviously, the effect of executing it is that afterwards the value of the static `argNumber`-member of class `Rule` is set to the number on the RHS. Recall function `setInt` of class `Rule` (Section 6.1): This function always returns `true`.

CGetArgument This class somehow extends command `CAssignment`. It consists of one `ExprArrayIndex` called `obj`, one non-negative `int`-member `index` and a list of `int`-arrays, each of length $n+1$, called `argNumber`, with m items. Its C++ code representation is the following:

$$\begin{aligned} & \text{Expression::assignExpression}(\text{exprArray}[\text{index}], \\ & \text{obj} \rightarrow \text{getArgument}(\langle \text{argNumber}[0] \rangle) \rightarrow \dots \rightarrow \text{getArgument}(\langle \text{argNumber}[m] \rangle)) \end{aligned}$$

if it occurs inside a condition, and

$$\begin{aligned} & \text{exprArray}[\text{index}] = \\ & \text{obj} \rightarrow \text{getArgument}(\langle \text{argNumber}[0] \rangle) \rightarrow \dots \rightarrow \text{getArgument}(\langle \text{argNumber}[m] \rangle) \end{aligned}$$

otherwise. The upper version does completely the same as the lower one, but returns the boolean constant `true`.

CMoveRight This class consists of one `ExprArrayIndex` called `parent` and two members of type `int` called `index` and `stepSize`, respectively. Its C++ code representation is the following:

```
parent->move(exprArray[index] , [stepSize])
```

if the value of member `i` of `parent` is greater than or equal to -1, or

```
Expression::moveRight(exprArray[index] , [stepSize])
```

otherwise. The effect of executing the upper version is that afterwards `exprArray[index]` points to the argument that comes [*stepSize*] arguments after the one it previously pointed to in *parent*'s argument list; If `exprArray[index]` is a NULL-pointer, it is set to the first argument of *parent* and [*stepSize*] is decreased by 1 before setting it. The effect of executing the latter version is almost the same, except that `exprArray[index]` must not be a NULL-pointer, and it always returns `true`.

CMoveAdvanced This class somehow extends the previous command. It again consists of one `ExprArrayIndex` called `parent` and one non-negative `int`-member `index`, but in contrast to `CMoveRight` also one `int`-array of length $n + 1$ called `argNumber`. Its C++ code representation is the following:

```
Expression::moveRight(exprArray[index] , <argNumber>)
```

The effect of executing it is basically the same as described above, but now it is possible to express the step-size in terms of lengths of instances of arbitrary sequence variables.

CGetFirst This very simple class only consists of one `ExprArrayIndex` called `index` and one non-negative `int`-member `lhs`. Its C++ code representation is

```
exprArray[lhs] = index->getFirst()
```

and executing it only makes `exprArray[lhs]` point to the first argument of the function pointed to by *index*.

CIsEqual This class consists of two members of type `ExprArrayIndex`, called `lhs` and `rhs`, respectively, and one member of type `int` called `index`. Its C++ code representation is the following:

```
Expression::equal(lhs, rhs, seqVarLen[index])
```

if *index* is non-negative, or

```
lhs->equals(rhs)
```

otherwise. This command is a conditional command and it checks whether the subtree-sequences starting at *lhs* and *rhs*, respectively, having the same length as the instance of the *index*-th sequence variable, are equal (not only the roots of those subtrees are checked, but the entire subtrees). In the latter version, the length of the sequences that are checked is just 1.

CForLoop This class represents the only loop-command. It consists of one `ExprArrayIndex` called `parent`, one `int`-member `seqVarIndex`, and one `int`-array of length $n + 1$ called `argNumber`. The corresponding C++ code is

```
for (seqVarLen[seqVarIndex] = 0; <argNumber> <= parent->argumentCount();  
    ++seqVarLen[seqVarIndex])
```

The effect of executing this command is that it iterates through all values of `seqVarLen[seqVarIndex]`, starting at 0, until the loop condition fails to evaluate to `true`, which eventually will be the case if `argNumber[seqVarIndex]` is non-zero.

Since there might be loops and conditions in the command sequence, the question is: When do the scopes of those loops/conditions end? The answer is quite simple: They last until the very end of the command sequence, i. e. all other commands coming after them are inside the loops/conditions. Hence, there are no additional commands needed that “close” conditions/loops, so to say.

6.2.2 Constructing Function match

Matching is a rather complicated task when sequence variables are involved. This is due to the fact that the lengths of the sequences of instances of each sequence

variable are not known in advance, neither at compile-time nor at run-time at the beginning of the matching process. A second problem is that, in general, if an expression e matches another expression h , there are lots of possible substitutions for the sequence variables in e that are suitable. This immediately leads to the question which of them is eventually chosen, and the answer is not that simple. The algorithm that was implemented in the frame of this thesis somehow “sorts” the sequence variables and tries to find a substitution where the length of the sequence of instances of the first sequence variable w.r.t. that ordering is minimal, similar for the other ones. The algorithm itself, and how it is constructed, is described in detail below, in Section 6.2.2.5.

Let from now on in this section always be e the expression describing the pattern, let h denote the expression that is tried to be matched, and let T and U denote their corresponding trees. Also let, for any sequence variable $a_$ in e , $|a_|$ denote the length of its instantiation when matching expression h . If more than one such instantiations are possible, $|a_|$ denotes the length of one of them. Please also note that only the *length* of the instantiation, i.e. the number of expressions that instantiate $a_$, and not the instantiation itself is of interest.

6.2.2.1 A Brute-Force Approach

The problem of matching with sequence variables mainly arises from the fact that the lengths of suitable sequences of instances of the variables (i.e. sequences of instances that make the expressions equal) are not known. Therefore one could just try out, for each sequence variable, all lengths (at least to an upper bound) and then check whether such a setting of lengths is fine or not. Indeed, this brute-force approach would really work, but is very inefficient. Namely, in many cases there are sequence variables whose instance-length is uniquely determined, and thus no loop or recursion is needed. For example, take the very simple expression $e = f[a_]$, where $a_$ stands for a sequence variable. No matter how the expression that is tried to be matched looks like, if it is matched by e its outermost function must have name f , and the only suitable substitution replaces $a_$ by the entire argument list of that function. So, $|a_|$ is uniquely determined. Trying all lengths from 0 to some upper bound would be really inefficient.

6.2.2.2 A More Efficient Approach

The considerations that were made in the last section immediately lead to a more efficient approach:

1. Find all sequence variables whose instance-lengths are uniquely determined.
2. In this process, also check all the other objects in the tree that are known until now, like names of functions/constants, and check whether different occurrences of the same variable (individual or sequence) are instantiated with equal objects.
3. Apply the brute-force approach to all other sequence variables, but only one after the other: Fixing a possible length for one variable may cause the length of another variable to be determined uniquely; Therefore, go back to 1.

The following example illustrates the new approach. Again, every symbol with a “_” at the end of its name stands for a sequence variable. The expression e that serves as the pattern is defined as

$$f[a_, b_, g[a_, a_]]$$

Although there are two different sequence variables involved, for every expression h that is matched by e there is only one suitable variable substitution, which means that the lengths of the sequences of instances of both sequence variables are uniquely determined. The reason is the following:

1. First of all, the name of the outermost function of h must be f .
2. No matter how many arguments the outermost function of h has, as long as it has at least one, the last one must be a function with name g . This can also be checked immediately.
3. The number of arguments of that function (g) can be arbitrary, as long as it is even. If it is even, $|a_|$ is given by the half of the number of function g 's arguments.
4. Now, it can be checked immediately whether the two occurrences of $a_$ in function g are instantiated by equal sequences of objects.
5. Since $|a_|$ is known, it can be checked whether the outermost function (f) has enough arguments, namely at least $|a_| + 1$. If so, also $|b_|$ is determined uniquely.

In the above steps no single iteration over possible lengths was needed to check whether e matches h .

This is exactly the strategy that is pursued by the matching-algorithm that was implemented for Version 3, see Section 6.2.2.5.

6.2.2.3 The Determinate Part of a Tree

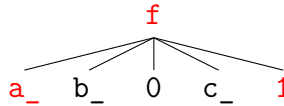
Assume that e indeed matches h and that in the process of matching e against h the lengths of the sequences of objects that instantiate some of the sequence variables are already known. Then some part (compare Definition 23) of T is said to be *determinate*; It is formally defined in the following way:

Definition 26. Let T , e and h be as above. Then the *determinate* part of T is the set of vertices of T consisting of

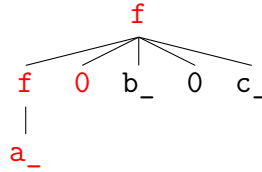
1. The root of T (this is also the root of the determinate part), and
2. Whenever there is a node in the determinate part, also of the first of its children (from left to right) until the first child that is a sequence variable with still indeterminate length, and of its children (from left to right) starting after the last child that is a sequence variable with still indeterminate length.

The following examples illustrate the concept of determinate parts of trees. Therefore, let a_* be a sequence variable where $|a_*|$ is already known, and b_* and c_* sequence variables such that $|b_*|$ and $|c_*|$ are not known at the moment. All other leaves are either individual variables or constants - That does not make any difference. Vertices belonging to the determinate part are printed in **red** letters:

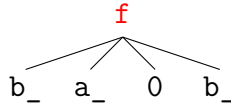
•



•



•



The reason for introducing the concept of the determinate part of a tree is the following: The places of all vertices in U (i.e. the tree corresponding to h) that correspond to vertices in the determinate part of T , and only of those vertices, are known. This means that they can be accessed without needing to know the lengths of the instances of the other, still indeterminate sequence variables. Take,

for example, the 1 in the first of the above examples. Although there are sequence variables `b_` and `c_`, whose instances still have indeterminate lengths, in the tree, and even with the same parent as 1, the place of the vertex in U corresponding to 1 is uniquely determined: It is the *last* child of the root node. Hence, 1 belongs to the determinate part. On the other hand, 0 is not an element of the determinate part. The only information one can derive is that *some* child of the root of U must be 0, but it is not yet known *where exactly* this vertex has to be, because in T both before it and after it there are sequence variables whose instances still have indeterminate lengths.

6.2.2.4 Class `SplitNode`

The implementation of the concept of determinate parts is done with the new class `SplitNode`. This class inherits from class `Expression` and basically models nodes in trees, just like class `Function`: It has a name and children/arguments. However, unlike ordinary functions, its arguments are not just stored in one single array, but in three separate lists, called `left`, `center` and `right`, respectively. Please note that class `SplitNode` is part of the implementation of the *compiler* and hence `left`, `center` and `right` are really *lists* and not just pointers to single objects which themselves then point to their right neighbors, and so on, as in the implementation of the *compiled programs* (see Section 6.1). The meaning of those three lists is quite obvious: `left` contains all the arguments that belong to the determinate part of the tree and come before the first sequence variable with unknown length, `right` contains all the arguments that belong to the determinate part of the tree and come after the last sequence variable with unknown length, and `center` contains the rest. Please note two things: First, during the construction of the matching-function the content of the three lists changes: A sequence variable whose length is unknown at the beginning may (and, in fact, certainly will) at some point become known (That's why `left`, `center`, `right` are implemented as dynamic *lists* rather than arrays). Second, note that it is possible that some of the lists are empty; In particular, this is the case when there is no sequence variable with indeterminate length in the argument list of the `SplitNode`-object. Then, everything is contained in `left`, and `center` and `right` are empty.

Before the matching-algorithm is constructed, a copy of the LHS of the rule is created where `Function`-objects are replaced by `SplitNode`-objects where *all* sequence variables are regarded as having indeterminate length. All this happens in function `makeCode` of class `Rule`.

6.2.2.5 Constructing the Algorithm

The algorithm for matching, just as the definition of the `rewrite`-function, is constructed in terms of a sequence of commands in function `makeCode` of class `Rule`. It follows the strategy presented in Section 6.2.2.2, but before it can be described in detail, there is one very important aspect to be noticed: Obviously, the algorithm is constructed at compile-time. At this time, no concrete expression h is known, so how can one say “The length of the instance of sequence variable $a_$ is determinate now”? Surely, the lengths of the instances of sequence variables depend on the expression that is tried to be matched. The answer is the following: Although the lengths are not known, it is possible to find out at which point they will be known later when eventually really executing the matching-algorithm on concrete expressions, and therefore the method that creates the matching-algorithm somehow “pretends” to know them. Just recall the example in Section 6.2.2.2: Although only the pattern of the rule and no concrete expression is given, from the third element in the enumeration on it is clear that $|a_|$ will be known when the algorithm is executed on a concrete expression.

The following is the outline of the method that constructs the definition of function `match` in terms of a command sequence. Its input is the tree T corresponding to the LHS of the rule, where instead of `Function`-nodes there are `SplitNode`-nodes. However, the commands that are created don’t work on T , but on the tree U corresponding to the expression that is tried to matched:

1. Go through the determinate part of T until you reach a node N where `center` contains occurrences of only *one* sequence variable with indeterminate length (let $a_$ denote that sequence variable). If no such node exists, go to 6.
2. Let N' denote the node in U corresponding to N . Create commands for checking whether all the predecessor nodes of N' in U have the right name and sufficiently many arguments (the exact number might not be known until now).
3. Create a command that assigns $|a_|$ to the element of array `seqVarLen` corresponding to $a_$. This value depends on the arity of N' , the number of occurrences of $a_$ in the arguments of N' , and on the number of the other arguments of N' (their lengths are already determined), and it is

$$|a_| = \frac{\text{arity}(N') - \sum_{i \in \text{arguments}(N') \setminus \{a_ \}} \text{length}(i) \cdot \#\text{occurences}(i)}{\#\text{occurences}(a_)}$$

Of course, first of all a command needs to be created that checks whether this value is a Natural number. If so, from now on $|a_|$ can be assumed to be known.

4. Update T : There might be other nodes in its determinate part with $a_$ as the first/last element in **center**, which is not allowed any more since $|a_|$ is determinate now and the first/last element of **center** must be indeterminate.
5. Go back to 1.
6. Create commands that check all the nodes in U corresponding to nodes in the determinate part of T for having the right names and arity; Create commands that check all the constants in U corresponding to constants in the determinate part of T for having the right names; Create commands that check whether all the different occurrences of one and the same variable (individual or sequence) in the determinate part of T are instantiated with equal objects in U , for all variables.
7. If there is no more sequence variable with unknown length, create a command that simply returns **true**, and we are done. Otherwise, continue with the next step.
8. Search for a node N in the determinate part of T where the number of different sequence variables with unknown length occurring in **center** is minimal (it is at least 2). Let $a_$ denote the first element of **center** and N' again denote the corresponding node in U . Create a for-loop-command that iterates through all the possible values for $|a_|$, from 0 to the arity of N' minus the sum of the lengths of all the arguments of N' with known length.
9. Since from now on $|a_|$ can be assumed to be known (in the definition of function **match** we are now inside the body of the loop that iterates through the possibilities for $|a_|$), go back to 4.

The last thing that happens when the command sequence is created is a little optimization, just as with the command sequence of the **rewrite**-function. The optimization consists of three steps:

1. Remove redundant loops: There might be for-loop-commands where afterwards no single condition appears, but only assignments and the **return true** command created in step 7. Thus, the first iteration will always yield success, and therefore no loop is needed; Instead, the length of the corresponding sequence variable can immediately be set to 0.
2. Remove redundant assignments: A redundant assignment is an assignment to a variable (more precisely: element of **exprArray**) that is never used later on.
3. Fill index holes: Index-holes of **exprArray** are filled. Please consult Section 5.2.4.6 for more information.

6.2.2.6 An Example

Let us consider now an example in order to present the ideas proposed in the previous sections in a more intuitive way. For that reason, assume that in the following the pattern (i.e. LHS of the rule) is given by the expression

$$f[a_, b_, c_, g[b_, b_]]$$

where all the non-function symbols $a_$, $b_$ and $c_$ are sequence variables. The following C++ source code was generated fully automatically by the compiler of Version 3 by applying the algorithm showed in the previous section, and it represents the definition of function `match`:

```

1  bool Rule0::match(Expression *expr){
2      argNumber = 1;
3
4      if (expr->isFunction(Expression::RuleSymbols + 0) &&
5          expr->argumentCount() >= argNumber &&
6          (Expression::assignExpression(exprArray[1], expr->getArgument(-(1)))) &&
7          (setInt(argNumber, 0)) &&
8          exprArray[1]->isFunction(Expression::RuleSymbols + 1) &&
9          exprArray[1]->argumentCount() >= argNumber &&
10         !((exprArray[1]->argumentCount() - argNumber) % 2)){
11
12         seqVarLen[1] = (exprArray[1]->argumentCount() - argNumber) / 2;
13         exprArray[0] = exprArray[1]->getArgument(0);
14         exprArray[1] = expr->getFirst();
15         Expression::moveRight(exprArray[1], expr->argumentCount() - (1));
16         exprArray[2] = exprArray[1]->getFirst();
17         Expression::moveRight(exprArray[2], seqVarLen[1]);
18         exprArray[3] = exprArray[0];
19
20         if (Expression::equal(exprArray[2], exprArray[3], seqVarLen[1])){
21
22             for (seqVarLen[0] = 0;
23                 seqVarLen[0] + seqVarLen[1] + 1 <= expr->argumentCount();
24                 ++seqVarLen[0]){
25
26                 argNumber = seqVarLen[0] + seqVarLen[1] + 1;
27                 seqVarLen[2] = expr->argumentCount() - argNumber;
28                 exprArray[1] = expr->getFirst();
29                 Expression::moveRight(exprArray[1], seqVarLen[0]);
30                 exprArray[2] = exprArray[0];
31
32                 if (Expression::equal(exprArray[1], exprArray[2], seqVarLen[1]))
33                     return true;
34             }
35         }
36     }
37     return false;
38 }
```

Please note that the content of the `Expression::RuleSymbols`-array is `[f, g]` and that `seqVarLen` is an int-array of length 3 of class `Rule0`. `argNumber` and

`exprArray` both are static members of the base class `Rule`, and therefore also of class `Rule0` which inherits from it, where the first one is of type `int` and the latter one is an `int`-array of length 4.

Line 2 of the above code simply sets `argNumber` to 1. The next thing is a big `if`-block, starting at **line 4**. The condition of the `if`-block is a big conjunction, ranging from line 4 to line 10. Please note that in C++, as soon as the first element of a conjunction evaluates to `false`, the other elements of the conjunction are ignored and `false` is returned immediately. **Line 4** checks whether `expr` is not a single constant but a function whose name is `f`. In **line 5** it is checked whether this function has at least `argNumber` arguments, where the value of `argNumber` is 1. This is due to the fact that no matter what the various variables will be instantiated with, there *must* be one last argument (namely `g`). The next line, **line 6**, makes the second element of `exprArray` point to exactly that last argument (please note that function `assignExpression` always returns `true`). In **line 7**, `argNumber` is set to 0 (this function always returns `true`, too). In **lines 8 and 9** it is checked whether the last argument of the outermost function is itself a function with name `g` and at least `argNumber` (`=0`) arguments. The last thing that is checked in **line 10** is whether the number of arguments of that functions is even.

In case of success, the overall “shape” of the input is right. Now come the instances of the sequence variables. **Line 12** sets the length of the instance of variable `b_` to half of the number of arguments of function `g`, since `argNumber` is still 0. **Lines 13 to 18** do nothing else than making the third and fourth element, respectively, of `exprArray` point to the two first subexpressions in the sequences that instantiate the two occurrences of `b_` in the argument list of function `g`. The way this happens is indeed a bit cumbersome, taking into account that the same thing could be achieved with fewer commands, and thus could be subject to improvement in future versions.

The next part of the program is again an `if`-block, this time consisting of only one condition (and no conjunction): **Line 20** simply checks whether the two sequences that instantiate the two occurrences of `b_` in the argument list of `g` are equal.

Now comes in **lines 22 to 24** a `for`-loop which iterates over the possible lengths of the instance of the first sequence variable `a_`, since this cannot be determined directly (at least when following the algorithm that was described in Section 6.2.2.5). The least possible length is of course 0, the upper bound is given by the fact that $|a_| + |b_| + 1$ must be less than or equal to the arity of the outermost function.

From now on, in the body of the loop, $|a_|$ is fixed. In **line 26** the value of `argNumber` is set to the number of arguments of the outermost function that are known until now; By the loop condition, this number is guaranteed not to be

greater than the arity of that function. Since $|a_|$ was fixed, also $|c_|$ can be fixed now, and this happens in **line 27**. The next three lines, **lines 28 to 30**, again do nothing else than making the second element of `exprArray` point to the first subexpression in the sequence that instantiates the occurrence of $b_$ in the argument list of f (depends on $|a_|$), and the third element of `exprArray` point to the first subexpression in the sequence that instantiates the first occurrence of $b_$ in the argument list of g .

The last thing happens in **lines 32 and 33**: If the two sequences are equal, then the input expression is matched and `true` is returned. Otherwise, one more loop iteration is carried out, until no more is possible. In this case, `false` is returned in **line 37**.

6.2.3 Constructing Function `rewrite`

The construction of the definition of function `rewrite` is done entirely in function `makeCode` of class `Rule`. Indeed, the same approach is pursued as in Version 2: Consider mappings from the LHS of the rule to its RHS (this time they have to be type preserving), choose the one where the least number of operations is required to transform the LHS into the RHS by using the commands listed in Section 6.2.1, and eventually generate those commands. However, there is one thing that is different than before: Only mappings of *functions* are considered at the beginning! This is due to the fact that, in general, there are far more type preserving mappings than arity preserving mappings, which means that trying all of them could take too much time. Therefore, only the different mappings w.r.t. functions are generated and the other objects (i.e. variables and constants) are ignored at the beginning. Then, for every such mapping, the argument lists of all the functions of the LHS that are not mapped to \emptyset but to other functions are compared with the argument lists of their images. This is done in the following way: Assume there are five operations on argument lists:

- **insert**: This operation modifies the given argument list by inserting a new argument at an arbitrary position.
- **remove**: This operation modifies the given argument list by removing an arbitrary argument.
- **replace**: This operation modifies the given argument list by replacing an arbitrary argument by a new argument.
- **swap**: This operation modifies the given argument list by swapping two arguments in it.

- **move**: This operation modifies the given argument list by moving an argument leftwards for an arbitrary amount.

Then, comparing the argument lists of two functions f and g (not necessarily with the same arity) consists of computing the least number of operations needed for transforming f 's argument list into the one of g .

Before showing the algorithm that takes care of this comparison in detail, it is necessary to define a new relation on tree-vertices, namely *weak equality*.

Definition 27. Let v be a vertex of tree T , u any vertex, μ a type preserving mapping from T to somewhere else (indeed, only the part of μ that maps the functions is important). Then v is weakly equal to u w.r.t. μ , denoted as $v \simeq_\mu u$, iff

- Both v and u are constants, or
- v and u are the same variable, or
- Both v and u are functions and $\mu(v) = u$

The algorithm is implemented in function `compareSignatures` of class `Expression` and works in the following way: Assume you are given the two argument lists A and B and a type preserving mapping μ (again, only the restriction of μ to functions matters):

- If neither A nor B is empty, $A = [a, l_]$, $B = [b, r_]$ ($l_$ and $r_$ stand for sequences of arguments), and $a \simeq_\mu b$, then just continue with $[l_]$ and $[r_]$
- Otherwise, if again neither A nor B is empty and $A = [a, l_]$ and $B = [b, r_]$, let $a_l := \{x \in l_ : a \simeq_\mu x\}$, $a_r := \{x \in r_ : a \simeq_\mu x\}$, $b_l := \{x \in l_ : x \simeq_\mu b\}$ and $b_r := \{x \in r_ : x \simeq_\mu b\}$:
 - If $|b_r| \geq |b_l|$, then let x be an object with $x \simeq_\mu b$:
 - * If $|a_r| > |a_l|$, then *inserting* x in front of a will do the job; Continue with $[a, l_]$ and $[r_]$
 - * Otherwise, *replacing* a by x will do the job; Continue with $[l_]$ and $[r_]$
 - Otherwise

- * If $|a_r| > |a_l|$, then either *swapping* a with one of the elements in b_l or *moving* one of those elements in front of a will do the job; Try all of those operations Φ and call this algorithm recursively on $\Phi([a, l_])$ and $[r_]$. In the end, choose the operation Φ where the number of additional operations for transforming $\Phi([a, l_])$ into $[r_]$ is minimal.
 - * Otherwise, *removing* a will be a suitable operation; Continue with $[l_]$ and $[b, r_]$
- If A is empty, then *inserting* vertices that are weakly equal to the elements of B will do the job
 - If B is empty, then *removing* all the elements of A will do the job

Remark 28. If $A = [a, l_]$, $B = [b, r_]$, $a_l = \{x \in l_ : a \simeq_\mu x\}$, $a_r = \{x \in r_ : a \simeq_\mu x\}$ (like above) and a is a function, then $a_l = \emptyset$ and $|a_r| \leq 1$. Similar for b , b_r and b_l .

After executing the above algorithm, the two argument lists are weakly equal, i. e. they have the same length and the i -th arguments of both lists are weakly equal, for all i .

However, since the same idea is pursued as in Version 2, function `compareSignatures` does not really carry out the operations but only checks how many of them are needed. Moreover, the considered mapping μ , which is already defined for *all* the functions, is extended to *some* of the other objects (variables, constants): For example, take the very first case from above. Neither A nor B is empty and their first elements are weakly equal. The description of the algorithm in this case simply says “Continue with the rest of A and B ”, but this can be viewed as “Map the first element of A to the first element of B ”. Hence, μ is extended (except the first element of A is a function, then μ has already been defined on it before). But not all vertices of the LHS of the rule are mapped to something. For example, consider the case in the above algorithm where the object x appears. The description just says “Let x be an object with ...”, but it is not said where this object comes from (only that it surely not comes from A). So, it either comes from another argument list somewhere else in the tree, or it has to be created. Coming from another argument list means that it is not needed any longer there (it is *removed*).

As in the previous version, function `findBestMapping` of class `Rule` takes care of finding the best mapping. It does so by first considering all mappings of functions and then calling, for every such mapping μ , function `countCopyingOps` of class `Function`, which has the same meaning as in Version 2, and which

has `compareSignatures` as a helper function. The only thing that function `countCopyingOps` does when called on two trees and a mapping μ between those two trees (which is only defined on the functions) is extending the mapping μ to a certain amount, counting the number of *some* (not all!) operations required for transforming one tree into the other given the mapping μ , and filling some part of the `exprArrayIndex`-array, which has absolutely the same purpose as in Version 2: Storing additional information about the vertices of the tree representing the LHS of the rule. The part of the array that is filled is the one that corresponds to functions (nodes). The meaning of an element of `exprArrayIndex` is the following (f is the corresponding function):

- -1: f doesn't have to be stored in function `rewrite`
- -2: f needs to be stored, because it is reused and its parent changes in the rewriting process
- -3: f needs to be stored, because although it is reused and its parent remains the same, some successor of it needs to be accessed (e.g. some arguments have to be swapped/moved/inserted)
- -4: f needs to be stored, because it is not reused and therefore surely all of its successors have to be accessed

The last thing that is done in function `findBestMapping` is the same as in the previous version: Finding, for every variable that has to be copied at some point, an occurrence in the LHS of the rule such that the total length of the paths leading to those occurrences is minimal. Since this works really in completely the same way as in Version 2, let me refer to Section 5.2.4.3 for a detailed description. Again, some part of `exprArrayIndex` is filled as well. In addition to the four possible values of elements corresponding to functions, there are two more possible values (let o be the corresponding object; not necessarily a function):

- -6: o is a variable which is reused and whose parent remains the same in the rewriting process, but nonetheless needs to be stored (for copying)
- -7: o is a function that is reused and whose parent remains the same, and the `exprArrayIndex`-value of some successor of it is -6

Summarizing, function `findBestMapping` does the following:

- Constructing some part of a type preserving mapping (defined on all functions and some variables and constants) that gives rise to a transformation of

the LHS of the rule into its RHS where the number of required operations is minimal. Although the mapping is in general not defined on all vertices, it is defined on sufficiently many vertices to be sure that the number of operations is minimal. Note that also the inverse of that mapping is constructed.

- Filling some part of the `exprArrayIndex`-array. The meaning of the values of the elements is described above.

In order to be able to eventually really construct the definition of function `rewrite`, the mapping μ that is used for transformation needs to be defined on the entire LHS, which is not the case until now. There might be, say, an occurrence of the variable x where μ is not yet defined on the LHS and also an occurrence of the same variable on the RHS that does not yet have an inverse w.r.t. μ . This, however, means that μ is not even a type preserving mapping, according to Definition 14. The solution to this problem is rather simple: Just map the LHS-occurrence of x to the RHS-occurrence. But what if there are many such occurrences? Then there are also many possibilities of mapping ... Does that mean that once again all of them have to be considered and finally the best one is chosen? The answer is: No. Although there are in general lots of possibilities to map the remaining variables and constants on both sides of the rule to each other, they are all equally good in terms of minimal number of operations. No matter which object is mapped to which object, the eventual rewriting-algorithm surely has to traverse the tree twice: First to access the corresponding vertex and store it somewhere in `exprArray`, and second to move it to its *new* position. The emphasis is on *new*, since the parent of the considered vertex surely changes. If its parent remained the same, i.e. in the rule the image of the parent of the corresponding object was exactly the parent of its image, then the `compareSignatures`-function would have already detected that and thus automatically defined μ on it.

Now, since it doesn't matter how the mapping of the remaining objects looks like (as long it is type preserving), simply the "first" one is chosen. "First" means that the first object on the LHS where μ is not defined on is mapped to the first suitable object on the RHS that doesn't have an inverse w.r.t. μ , and so on. There is only one little detail that has to be taken into account: In principle, every constant can be mapped to every constant, because constants can be renamed. However, renaming, although only one pointer operation, still requires some effort. Hence, in the extension-process of μ it is attempted to map constants to constants with the same name in order to avoid renaming. Only if this is not possible any more, constants are mapped to any constants. There is no such thing with variables, since an occurrence of a variable can only be mapped to an occurrence of the same variable anyway.

In the next step, the definition of the `rewrite`-function is constructed in terms of a command sequence by calling the two functions (of class `Expression`) `createCopyingCommands` and `createCreatingCommands`. They work very similar to their counterparts in Version 2. The only thing that is worth mentioning is that in function `createCreatingCommands` once again some “trying of all possibilities” has to be done. Namely, if one argument of a function is mapped to an argument of the image of this function, but to a different place in the argument list, this can be achieved both by moving it to that place or by swapping it with another argument. Function `compareSignatures` already tried out both possibilities and found out which one is better (w.r.t. number of operations), but since `compareSignatures` only cares about the number of operations, this information is not stored and thus it has to be done again.

The very last step coincides with the last step of the previous version. It consists of performing some very little optimizations of the command sequence:

- Remove redundant assignments: A redundant assignment is an assignment to a variable (more precisely: element of `exprArray`) that is never used later on.
- Fill index holes: Index-holes of `exprArray` are filled. Please consult Section 5.2.4.6 for more information.

The following remark is of particular importance:

Remark 29. In the entire description of the construction of the `rewrite`-function, no distinction is made between individual variables and sequence variables. This is due to the fact that sequence variables behave just as individual variables when it comes to transforming a tree into another tree: Their instances can be deleted, created, inserted, ... just as any other objects, as soon as the length of their instances is known, which is the case in function `rewrite`.

6.3 How To Use “Compiler/SV”

In fact, there is nothing to say about how “Compiler/SV” can be used, because it behaves exactly as “Compiler”. Again, it is a compiler generating programs that implement specific rule sets. The only difference is that it can deal with sequence variables, but this affects only its implementation, and not how it is used. Therefore, let me refer to Section 5.3 for more information.

6.4 An Example

This example illustrates both the behavior of the compiler and the behavior of a compiled program. Therefore let the rule set \mathcal{R} consist of the one and only rule

```
Rule[VarList[a_, b], f[a_, g[a_,b], a_, 1, 2], f[1, g[a_,a_], 2]]
```

The tree representation of the LHS and RHS of the rule is shown in Figure 6.2. As usual, and as defined in Section 2.6, a variable with “_” at the end of its name denotes a sequence variable; In this case, there is only $a_$.

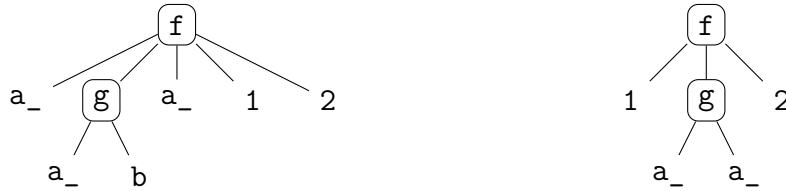


Figure 6.2: The LHS and RHS of the rule, represented as trees

There are in total 12 type preserving mappings from the LHS of the rule to the RHS (compare Section 2.8.3), which are not that many. In general, however, there are very many type preserving mappings from one expression to another, at least when the expressions grow huge, and there are more type preserving mappings than arity preserving mappings.

The best mapping w.r.t. the number of operations in the above rule is given by

$$\{0 \rightarrow 0, 1 \rightarrow 4, 2 \rightarrow 2, 3 \rightarrow 3, 6 \rightarrow 1, 7 \rightarrow 5\}$$

and was computed fully automatically by the compiler. A graphical representation of the mapping is shown in Figure 6.3.

If one compares Figure 6.3 to Figure 5.2 (which is the figure showing the best mapping in the example of Version 2), one realizes that no preserved parts are shown in this example. The reason is that it is hard to define a concept as “preserved parts” at all if sequence variables are involved. Take, for instance, function g in this example: Although it is the second argument of its parent and also its image is the second argument of its respective parent, that does not mean that the object corresponding to it in the concrete expression that is rewritten will be the second argument of its parent as well, since in the rule there comes a sequence variable before g .

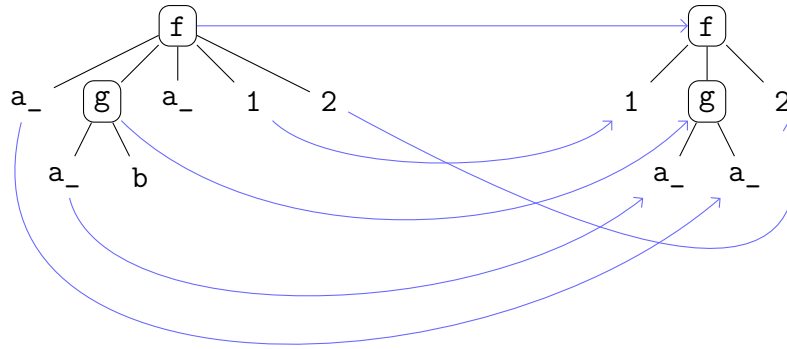


Figure 6.3: The best mapping, indicated by blue arrows

The following is the resulting C++ source code of the definition of function `rewrite`, where each line in the function body stands for exactly one command of the command sequence that was automatically generated by the compiler, following the strategy presented in Section 6.2.3:

```

1 void Rule0::rewrite(Expression* &expr){
2     exprArray[0] = 0;
3     exprArray[1] = exprArray[0];
4     expr->move(exprArray[1], 1);
5     expr->removeArguments(exprArray[0], seqVarLen[0]);
6     expr->move(exprArray[0], 1);
7     exprArray[2] = 0;
8     exprArray[0]->move(exprArray[2], seqVarLen[0]);
9     exprArray[0]->deleteArguments(exprArray[2], 1);
10    expr->deleteArguments(exprArray[0], seqVarLen[0]);
11    exprArray[0] = 0;
12    expr->swapArguments(exprArray[0], 1, 0, 1);
13    expr->move(exprArray[0], 1);
14    exprArray[2] = 0;
15    exprArray[0]->move(exprArray[2], seqVarLen[0]);
16    exprArray[0]->insertArguments(exprArray[2], exprArray[1], seqVarLen[0]);
17 }
```

Obviously, the definition consists of 15 instructions. However, some of them should better be grouped together to form only one single instruction, since from a logical point of view this makes more sense (like, e.g., the commands in lines 7 and 8: They simply make the third element of `exprArray` point to a certain object).

Ad line 4: Moving a NULL-pointer n positions to the right makes it point to the n -th argument in the argument list, for any $n > 0$.

Ad line 5: Removing a sequence of objects from an argument list does not delete the objects! The first parameter of the function defines the last object that is *not* removed (if this is a NULL-pointer, removing is started at the very beginning), the second parameter defines the length of the sequence that is removed.

Ad line 16: The `insertArguments`-command requires three parameters: The first one defines the argument where the new objects should be inserted after; The second one points to the first object in the sequence that should be inserted; The last one defines the number of objects that should be inserted.

The subsequent figure illustrate how the above code works when the function is called on the expression

`f[0, 1, 2, g[0,1,2,3], 0, 1, 2, 1, 2]`

that is obviously matched by the LHS of the rule if `a_` is instantiated by the sequence `[0, 1, 2]` and `b` is instantiated by `3`. Hence, `seqVarLen` can be assumed to be an array of length 1, and the value of its only element is the length of the instance of `a_`, which is 3. `exprArray`, on the other hand, is an array of length 3. The upper part of each of the following figures (Figures 6.4 to 6.12) is dedicated to the expression *as it currently is*, whereas the lower part is dedicated to `exprArray` and where its elements point to.

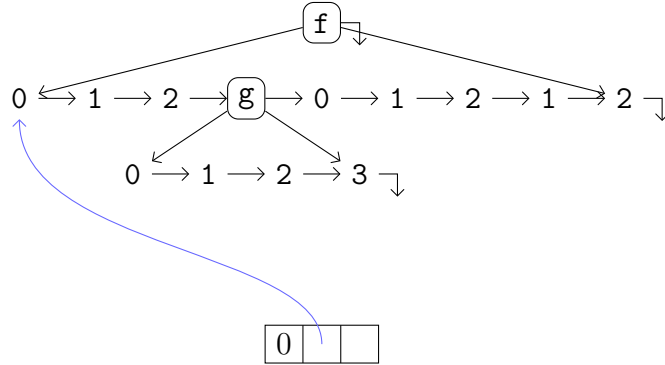


Figure 6.4: After executing lines 2, 3 and 4; Recall Section 6.1 about the implementation of expressions in the *compiled* programs

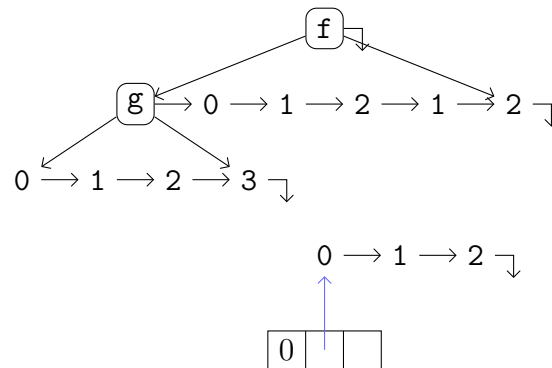


Figure 6.5: After executing line 5:

`expr->removeArguments(exprArray[0], seqVarLen[0])`

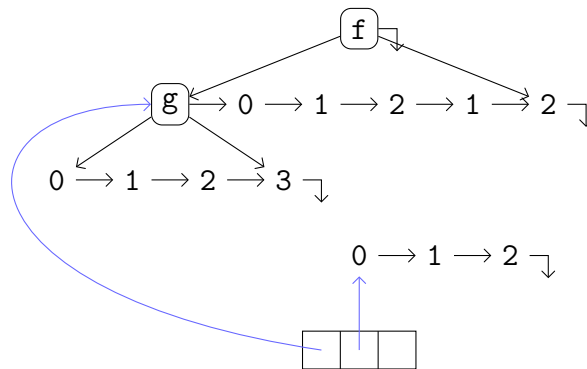
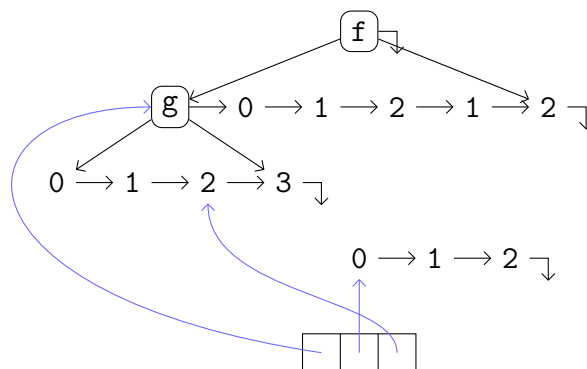
Figure 6.6: After executing line 6: `expr->move(exprArray[0], 1)`

Figure 6.7: After executing lines 7 and 8

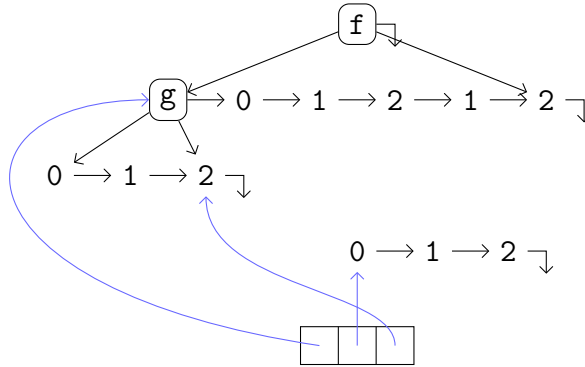


Figure 6.8: After executing line 9:
`exprArray[0]->deleteArguments(exprArray[2], 1)`

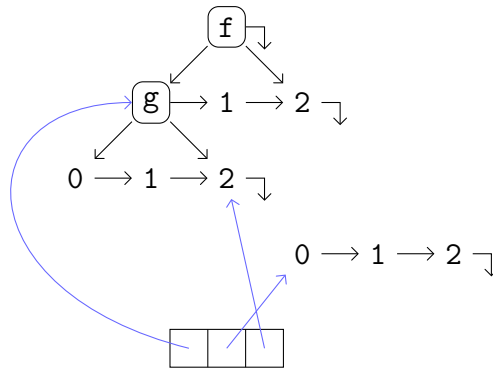


Figure 6.9: After executing line 10:
`expr->deleteArguments(exprArray[0], seqVarLen[0])`

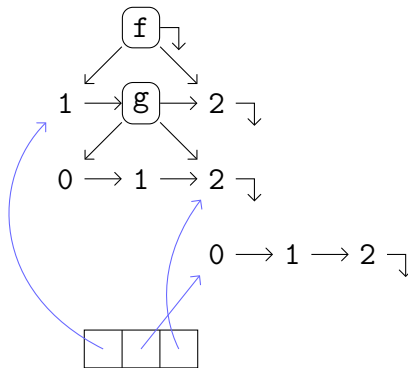


Figure 6.10: After executing lines 11 and 12

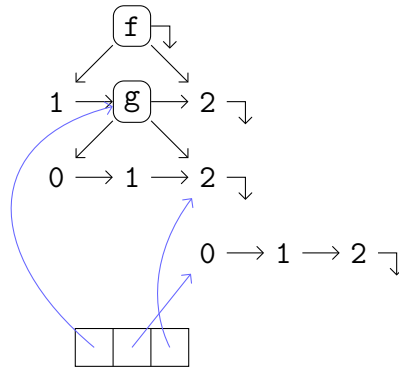


Figure 6.11: After executing lines 13, 14 and 15; The latter two would not be needed

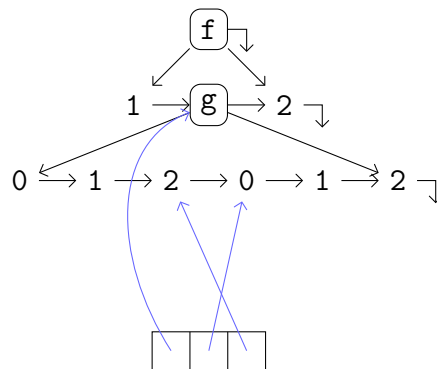


Figure 6.12: After executing line 16:
`exprArray[0]->insertArguments(exprArray[2], exprArray[1], seqVarLen[0])`

Chapter 7

Tests And Timing Experiments

This chapter is dedicated to some tests and timing experiments which have been performed in order to compare the three versions as well as the new, destructive concept to the old, constructive way of rewriting. Various (small to medium-size¹) rule sets have been used to rewrite various expressions again and again, some thousand times (this number is in the following denoted by N), and the gross time was recorded using the built-in C++ function `clock()`. Sometimes the net time was recorded, too. In some cases, both the time needed for finding a subexpression and a rule that matches that subexpression *and* the rewriting-process were considered, in other cases *only* the rewriting-process itself, since this is where the focus of this thesis lies. Of course, before a timing experiment is presented in the sequel, it is explicitly made clear what parameters were used, i.e. how often the expression was rewritten, and so on. It is clear that all tests were not performed only once, but several times to make sure there are no “freak values”. Please also note that in connection with the compiler-versions (Versions 2 and 3), only the execution times of the *compiled* programs were considered, and no compilation times.

As already pointed out, the main instrument for timing was the `clock()` function of C++. This function does *not* return the total time that elapsed since the start of the program, but only the time that elapsed while the program was really executed (i.e. the processor-time of the program), in milliseconds. Hence, if it is called twice in the program, one time *before* and one time *after* the computation that should be timed, and the difference of the two results is taken, then one gets exactly the processor-time of the computation. More information on `clock()` can be found, for instance, in [1].

¹The number of rules varies between 2 and 108

The difference between gross- and net time is the following: If an expression has to be rewritten a lot of times, then also a lot of bookkeeping, that has nothing to do with rewriting, needs to be done. At the very beginning, the expression must be copied into a temporary variable, because after rewriting it, it is lost (that is somehow exactly the purpose of the new concept). However, since afterwards nobody is interested in the result of the rewriting process, the resulting expression has to be deleted, and the original expression must be copied again from the temporary variable in order to start the whole process again, and so on. The gross time includes all those things, whereas the net time only contains the real rewriting-process.

Before going on with the tests and timing experiments, some technical details of the hardware/software that was used:

- **Operating System:** Windows 7 (64 Bit)
- **CPU:** Intel Pentium P6200
- **CPU Speed:** $2.13 \text{ GHz} \times 2$
- **RAM:** 3.68 GB

7.1 Comparing the Different Versions

In this section, some tests are presented that were performed in order to compare the three different versions. Of course, “Compiler/SV” somehow takes a special position since it is the only version capable of dealing with sequence variables. Thus, whenever there is a rule set containing at least one of those sequence variables, then there is no question about which version to use. As a consequence, no rule set used for the tests in this section contains sequence variables.

7.1.1 General Parameter Settings

- In all of the following tests *both* the time needed for finding a subexpression and a rule that matches this subexpression *and* the time for rewriting this subexpression are considered.
- Not only one rule, but a whole rule set is used, and rules are applied *as often as possible*, just as in the standard rewriting process.

7.1.2 Rule Set Addition

The first rule set that was tried is **Addition**, which can be found in the appendix, Section B.1. It contains rules that make it possible to add up Integers represented as **Int**-objects (see Section 2.3), if the outermost function has name `+` and two arguments. The expressions that are rewritten are

$$+[1039750682, -230147035]$$

and

$$+[31, 12]$$

Note that the numbers can be automatically transformed into **Int**-objects, if the program is told to do so (see Section A.4). If the rule set does its job right, then the result of rewriting the first expression should be `809603647`, which is indeed the case (more precisely, the result is `0809603647`, but that is the same value in terms of numbers), and the second result should be `43`, which is also the case.

The value for N , i. e. the total number of rewritings that have been performed, in this example is 1000 for the first expression, and 10000 for the second one. Table 7.1 shows the gross times, in seconds, which were needed by the three different versions using rule set **Addition.txt**.

Expression	N	Interpreter	Compiler	Compiler/SV
<code>+[1039750682, -230147035]</code>	1000	8.767	1.528	1.529
<code>+[31, 12]</code>	10000	6.630	1.358	1.342

Table 7.1: Timings for rule set **Addition**

7.1.3 Rule Set Multiplication

The next rule set is **Multiplication**, which can also be found in the appendix, Section B.2. It contains rules that make it possible to multiply Integers again represented as **Int**-objects, if the outermost function has name `*` and two arguments. The two expressions that are rewritten are

$$*[46978, 563140]$$

and

`*[14, 3]`

If the rule set does its job right, then the result of rewriting should be 26455190920 for the first expression, which is indeed the case if the maximum number of rewriting steps is set to somewhere around 300, or higher (see Section A.2), and 42 for the second expression, which is also the case.

The value for N for rewriting the first expression is 1000, for the second expression it is 10000. Table 7.2 shows the gross times, in seconds, which were needed by the three different versions to rewrite the given expressions into the correct results using rule set **Multiplication**.

Expression	N	Interpreter	Compiler	Compiler/SV
<code>*[46978, 563140]</code>	1000	59.904	11.045	10.296
<code>*[14, 3]</code>	10000	20.202	4.321	3.791

Table 7.2: Timings for rule set **Multiplication**

7.1.4 Rule Set **NatAddition**

The next rule set is **NatAddition**, which, just as all the other rule sets, can be found in the appendix, Section B.3. It contains rules that make it possible to add up Natural numbers represented as **succ**-objects (let me again refer to Section 2.3), if the outermost function has name `+` and two arguments. The expression that is rewritten is

`+[31, 12]`

The two numbers can be automatically transformed into **succ**-objects, if the program is told to do so. If the rule set does its job right, then the result of rewriting should be of course 43, which is indeed the case.

The value for N in this example is 10000. Table 7.3 shows the gross times, in seconds, which were needed by the three different versions to rewrite the given expression into the correct result using rule set **NatAddition**.

Expression	N	Interpreter	Compiler	Compiler/SV
$+ [31, 12]$	10000	13.837	11.123	9.672

Table 7.3: Timings for rule set `NatAddition`

7.1.5 Rule Set `NatMultiplication`

The next rule set is `NatMultiplication`, which can be found in Section B.4 in the appendix. It contains rules that make it possible to multiply Natural numbers represented as `succ`-objects, if the outermost function has name `*` and two arguments. The expression that is rewritten is

$$* [14, 3]$$

If the rule set does its job right, then the result of rewriting should be of course 42, which is indeed the case.

The value for N in this example is again 10000. Table 7.4 shows the gross times, in seconds, which were needed by the three different versions to rewrite the given expression into the correct result using rule set `NatMultiplication`.

Expression	N	Interpreter	Compiler	Compiler/SV
$* [14, 3]$	10000	33.368	22.05	20.124

Table 7.4: Timings for rule set `NatMultiplication`

7.1.6 Rule Set `NatSum`

The next rule set is `NatSum`, which can be found in Section B.5 in the appendix. It contains rules that make it possible to add up the first n Natural numbers, for any Natural number n , which has to be given in terms of a `succ`-object. The expression that is rewritten is

$$\text{Sum}[5]$$

If the rule set does its job right, then the result of rewriting should be of course 15, which is indeed the case.

The value for N in this example is again 10000. Table 7.5 shows the gross times, in seconds, which were needed by the three different versions to rewrite the given expression into the correct result using rule set **NatSum**.

Expression	N	Interpreter	Compiler	Compiler/SV
Sum[5]	10000	8.611	5.476	5.007

Table 7.5: Timings for rule set **NatSum**

7.1.7 Rule Set **PolyAddition**

The last rule set is **PolyAddition**, which can be found in Section B.6 in the appendix. It contains rules that make it possible to add up two polynomials, where each is represented as

$$\text{Poly}[c_0, \text{Poly}[c_1, \dots \text{Poly}[c_d, 0] \dots]]$$

c_i corresponds to the coefficient of x^i , for any i . The expression that is rewritten is

$$+[\text{Poly}[1, \text{Poly}[2, 0]], \text{Poly}[-2, \text{Poly}[-1, 0]]]$$

which means that the sum of $1 + 2x$ and $-2 - x$ is computed. If the rule set does its job right, then the result of rewriting should be of course **Poly**[-1, **Poly**[1, 0]], corresponding to $-1 + x$, which is indeed the case. Note that the coefficients have to be given in terms of **Int**-objects, since this rule sets builds on rule set **Addition**.

The value for N in this example is again 10000. Table 7.6 shows the gross times, in seconds, which were needed by the three different versions to rewrite the given expression into the correct result using rule set **PolyAddition**.

Expression	N	Interpreter	Compiler	Compiler/SV
$+[\text{Poly}[1, \text{Poly}[2, 0]], \text{Poly}[-2, \text{Poly}[-1, 0]]]$	10000	15.397	3.448	2.995

Table 7.6: Timings for rule set **PolyAddition**

7.1.8 Rule Set **SequentCalculus**

This rule set, **SequentCalculus**, is used for presenting a rule set that contains sequence variables. As already pointed out earlier, such a rule set cannot be used for comparing the various versions, but only to present the absolute time Version 3 needed to rewrite an expression N times. Rule set **SequentCalculus** described in Section B.8 contains rules that make it possible to find out the truth values of propositional formulae, built from the logical connectives **not** (\neg), **and** (\wedge), **or** (\vee), **impl** (\Rightarrow) and **equiv** (\Leftrightarrow). The expression that is rewritten is

`prove[equiv[impl[A,B], not[and[A,not[B]]]]]`

which means that it is checked whether $A \Rightarrow B$ is logically equivalent to $\neg(A \wedge \neg B)$. If the rule set does its job right, then the result of rewriting should be **proved**, which is indeed the case.

The value for N in this example is again 10000, and the resulting gross time Version 3 needed for rewriting is 4.898 seconds.

7.1.9 Summary

All the tables from before are put together in Table 7.7, which makes it possible to conclude various things from the raw data:

- First of all, the fastest version seems to be “Compiler/SV” (Version 3), which is rather surprising. Although the implementation of expressions in its compiled programs involves lists rather than arrays, which means that arguments of functions cannot be accessed directly in constant time, it takes on average only 92% of the time “Compiler” (Version 2) takes for rewriting exactly the same expressions.
- Secondly, the slowest version seems to be the interpreter (Version 1), as expected. It takes on average almost four times as long as “Compiler/SV”, the fastest one.
- It is apparent that multiplication takes much longer than addition, using the given rule sets, both when representing numbers as **Int**- and as **succ**-objects. This, however, is not very surprising, since every multiplication involves lots of additions.

- It is also apparent that representing (Natural-) numbers as `Int`-objects rather than `succ`-objects makes the rewriting process much faster. If one compares the timings for `Addition + +[31,12]` and `NatAddition`, then the first one takes on average, taking into account all versions, only 25% of the time of the second one. Comparing `Multiplication + *[14,3]` to `NatMultiplication` yields a similar result: The first one takes only 33% of the time the second one takes.

Rule set	N	Interpreter	Compiler	Compiler/SV
Addition <small>(+[1039750682,-230147035])</small>	1000	8.767	1.528	1.529
Addition <small>(+[31,12])</small>	10000	6.630	1.358	1.342
Multiplication <small>(*[46978,563140])</small>	1000	59.904	11.045	10.296
Multiplication <small>(*[14,3])</small>	10000	20.202	4.321	3.791
NatAddition	10000	13.837	11.123	9.672
NatMultiplication	10000	33.368	22.058	20.124
NatSum	10000	8.611	5.476	5.007
PolyAddition	10000	15.397	3.448	2.995
SequentCalculus	10000			4.898

Table 7.7: Timings for all rule sets

7.2 Comparing Old (Constructive) and New (Destructive) Concept

This section presents some tests that have been performed in order to compare the old, constructive concept of rewriting to the new, destructive one this thesis is all about. Therefore, it was necessary to implement the old concept as well, at least in connection with Version 1. Section A.8 shows how the old-style approach rather than the new one can be used there. Concerning the compiler-versions (Versions 2 and 3), it was sufficient to just provide alternative definitions for the `rewrite`-functions of the rules that were tested. Those definitions were given manually, i. e. without modifying the compilers.

7.2.1 General Parameter Settings

- In all of the following tests *only* the time needed for rewriting an expression is considered, no subexpression- and rule-finding.

- For each expression only one single rule is used to rewrite it. This rule is already known in advance to match the expression.
- This one rule is applied only once, even if it was possible to apply it further. Afterwards, the original expression is rewritten again, and so on, in total N times.
- The value of N is the same throughout the whole section: It is 100000.
- Both the net- and the gross times were recorded.

All the rules are stored in one and only rule set. Since neither the rule set as a whole, nor the single rules make any sense in terms of applicability to other problems (like, say, addition of Integers), this rule set cannot be found in the appendix. Instead, it is shown here:

```

1 #MAKE_NOTHING
2 Rule[VarList[a, b], f[a, b, g[c, c]], g[c, f[a, b, c]]]
3 Rule[VarList[a, b], g[a, b], f[a, g[a, c], b]]
4 Rule[VarList[a, b], f[a, g[a, c], b], g[a, b]]
5 Rule[VarList[a, b], f[a, b, h[c]], g[a, h[a]]]
6 Rule[VarList[a], f[a, h[a], a], g[c, g[d, d]]]

```

It consists of five rules, where each rule has a specific purpose:

1. In the first one, there are exactly the same objects on the LHS and on the RHS. Hence, the difference between the old way of rewriting and the new one should be rather big.
2. In the second one, every object on the LHS also appears on the RHS, which means that everything can be reused. However, unlike the first rule, there are also objects on the RHS that have to be created.
3. In the third rule it is the other way round: Every object on the RHS also appears on the LHS, which means nothing has to be created, but some objects have to be deleted.
4. The fourth rule is a blend of the second and the third: Some objects can be reused, others have to be deleted, and some objects have to be created.
5. In the last rule, nothing can be reused and the new expression has to be created entirely.

7.2.2 Rule 1

The first test deals with the first rule, i. e.

```
Rule[VarList[a, b], f[a, b, g[c, c]], g[c, f[a, b, c]]]
```

and the expression it is applied on is

```
f[+[1,2], 0, g[c,c]]
```

Table 7.8 shows the timings of the three different versions, both when using the old, constructive approach and the new, destructive one.

Concept	net/gross	Interpreter	Compiler	Compiler/SV
Constructive	net	1.470	1.329	1.181
Constructive	gross	3.166	2.792	2.309
Destructive	net	0.281	0.092	0.077
Destructive	gross	1.919	1.544	1.232

Table 7.8: Timings for the first rule

7.2.3 Rule 2

The second test deals with the second rule, i. e.

```
Rule[VarList[a, b], g[a, b], f[a, g[a, c], b]]
```

and the expression it is applied on is

```
g[+[1,2], 0]
```

Table 7.9 shows the timings of the three different versions, both when using the old, constructive approach and the new, destructive one.

Concept	net/gross	Interpreter	Compiler	Compiler/SV
Constructive	net	1.701	1.563	1.203
Constructive	gross	3.026	2.793	2.122
Destructive	net	0.840	0.703	0.608
Destructive	gross	2.184	1.981	1.529

Table 7.9: Timings for the second rule

7.2.4 Rule 3

The third test deals with the third rule, i. e.

```
Rule[VarList[a, b], f[a, g[a, c], b], g[a, b]]
```

and the expression it is applied on is

```
f[+[1,2], g[+[1,2],c], 0]
```

Table 7.10 shows the timings of the three different versions, both when using the old, constructive approach and the new, destructive one.

Concept	net/gross	Interpreter	Compiler	Compiler/SV
Constructive	net	1.235	1.216	0.968
Constructive	gross	3.136	2.839	2.153
Destructive	net	0.516	0.345	0.329
Destructive	gross	2.340	1.966	1.498

Table 7.10: Timings for the third rule

7.2.5 Rule 4

The fourth test deals with the fourth rule, i. e.

```
Rule[VarList[a, b], f[a, b, h[c]], g[a, h[a]]]
```

and the expression it is applied on is

$$f[+[1,2], 0, h[c]]$$

Table 7.11 shows the timings of the three different versions, both when using the old, constructive approach and the new, destructive one.

Concept	net/gross	Interpreter	Compiler	Compiler/SV
Constructive	net	1.879	1.577	1.215
Constructive	gross	3.557	3.214	2.200
Destructive	net	1.015	0.935	0.421
Destructive	gross	2.777	2.403	1.497

Table 7.11: Timings for the fourth rule

7.2.6 Rule 5

The fifth test deals with the fifth rule, i. e.

$$\text{Rule}[\text{VarList}[a], f[a, h[a], a], g[c, g[d, d]]]$$

and the expression it is applied on is

$$f[+[1,2], h[+[1,2]], +[1,2]]$$

Table 7.12 shows the timings of the three different versions, both when using the old, constructive approach and the new, destructive one.

Concept	net/gross	Interpreter	Compiler	Compiler/SV
Constructive	net	1.435	1.112	1.155
Constructive	gross	3.604	3.213	2.293
Destructive	net	1.504	1.211	0.670
Destructive	gross	3.666	3.183	1.965

Table 7.12: Timings for the fifth rule

7.2.7 Summary

Since the net times are the important ones, in the following considerations only the net times are taken into account. Comparing the times from the constructive approach to those of the destructive approach suggest the following conclusions:

- The new, destructive concept that was presented in this thesis is really faster than the old, constructive one: Depending on the rule, it is up to seven times as fast. However, if the rule does not allow exploiting the new concept, as is the case in the fifth rule, then the new approach is not really faster, as expected. Altogether, the net times of all five rules of all versions using the destructive way of rewriting make up only about 50% of the times using the constructive way.
- The tests performed in this section confirm the results obtained in the previous section: “Compiler/SV” is on average a bit faster than “Compiler”, and the interpreter is the slowest version.

The following remark is important:

Remark 30. When comparing the two concepts, it does matter which expressions the variables in the rules are instantiated with. The bigger those expressions, the bigger the difference in the timings.

Therefore, a theoretical complexity analysis, which deals exactly with the differences between the two concepts, is presented in Section 8.1. This complexity analysis confirms the results obtained in this section.

7.3 Comparing “Compiler/SV” to *Mathematica*

“Compiler/SV” (Version 3) was not only compared to the other two versions, but also to the established *Mathematica* system ([3]). For that purpose, rule set **SequentCalculus** was implemented in a *Mathematica*-notebook as well, following the *Mathematica*-syntax for rewrite rules, but leaving the order and meaning of the rules unchanged. The expression that was rewritten is the same as in Section 7.1.8, namely

```
prove[equiv[impl[A,B], not[and[A,not[B]]]]]
```

which by the way is also a perfectly well-formed expression in *Mathematica*. The command that was used for timing is

```
Timing[For[i = 0, i < 10000, i++,
prove[equiv[impl[A, B], not[and[A, not[B]]]]]]
]]
```

Hence, N was set to 10000, just as in Section 7.1.8. The time consumed by the above computation is in the area of 1.685 seconds, compared to 4.898 seconds consumed by “Compiler/SV” (both gross times). Thus, *Mathematica* is a lot faster; This, however, *could* mostly be due to a better matching strategy pursued by *Mathematica*, similar to what is also suggested in Section 8.2.2.

Although a module containing the rules that define the propositional sequent calculus has also been created for the use in connection with Maude ([2], see Section B.9 in the appendix), it seems to be impossible to perform timing experiments like the ones above in Maude: Maude only reports the time needed for rewriting an expression *once* (which is, as expected, 0 in the given example), but it is not possible to rewrite an expression N times.

Chapter 8

Conclusion

Now that all the important aspects of the new, destructive concept have been explained, together with a detailed description of the three different versions that have been implemented, it is time to summarize everything a little bit and draw some conclusions.

The main goal of this thesis was to implement various rewrite-systems that make use of a new idea, proposed by my supervisor Prof. Jebelean: When applying a rewrite-rule to an expression, do not delete the entire old expression and construct the entire new expression, but rather *reuse* all objects that can be reused. Do so by simply moving them to their new places.

The results are now three different programs, called “Interpreter” (Version 1), “Compiler” (Version 2) and “Compiler/SV” (Version 3; compiler that is also capable of dealing with sequence variables, see 2.6), each implementing the new idea and capable of solving “real” problems, like adding Integers, if equipped with the right rule set. Every version is “special”, in the sense that it contains features the others do not have. The interpreter, for instance, does not need to compile rule sets before they can be used, but uses them immediately. The drawback, on the other hand, is that it is by far the slowest program, as can be seen in virtually every timing experiment of Chapter 7.

It is quite hard to say which version is the best one, since this always depends on what the user wants to do. But one thing is certain: “Compiler/SV” is not only the most powerful version (capable of treating sequence variables), but surprisingly also the fastest: As can be seen in Chapter 7, it is a bit faster than “Compiler”. In any case, none of the three versions can already compete with one of the established rewrite-systems like *Mathematica* ([3]) or *Maude* ([2]) by

any means (speed, functionality), but in my opinion they, and especially the new concept, provide solid foundations for future systems. This opinion is also based on the short complexity analysis of the next section.

8.1 Complexity Analysis

This section deals with a short theoretical analysis of the time-complexity of the new concept of rewriting presented in this thesis. More precisely, the complexities of the old-style approach and the new approach are compared and conclusions are drawn from that.

Before the analysis can be performed, some definitions have to be made. First of all, the complexity of rewriting an expression in one step, i. e. applying one rule to it, depends on *both* the rule *and* the expression. Therefore, let in the following denote r the rule and e the expression (which is matched by r), and

- Let $f_{L,n}$ and $f_{R,n}$ denote the number of function symbols with arity n on the LHS and RHS of r , respectively, for all n
- Let $v_{L,n}$ and $v_{R,n}$ denote the number of occurrences of the variable with index n on the LHS and RHS, respectively, of r , for all indices n
- Let c_L and c_R denote the number of constants on the LHS and RHS of r , respectively
- Let D denote the maximum arity of functions occurring on either side of r , and V the number of different variables
- Let ξ_n denote the number of function symbols in the instance of the variable with index n , for all indices n
- Let γ_n denote the number of constants in the instance of the variable with index n , for all indices n

Let further \mathcal{C} denote the number of elementary operations for creating/deleting one instance of a derived class of class **Expression**, i. e. a function or constant or variable.

Of course, if one wants to compare the new concept to the old one, at the beginning one needs to make clear how constants are treated in programs implementing the old concept: Are multiple occurrences of one and the same constant represented

by only *one* instance of class **Constant**, as is the case in Versions 1 and 2, or by *many* instances, like in Version 3? Since the first possibility makes more sense for programs that delete/create everything, i.e. follow the constructive concept, it is assumed that multiple occurrences are represented by one and the same objects. The number of elementary operations needed for applying r to e using the constructive old-style approach is given by

$$\mathcal{O} \left(\sum_{i=0}^D (f_{L,i} + f_{R,i}) + \sum_{i=0}^V (v_{L,i} + v_{R,i}) \xi_i \right) \cdot \mathcal{C} \quad (8.1)$$

The validity of the above formula is quite apparent: Everything in e which is not a constant has to be deleted, and everything in the RHS of r has to be created, unless it is a constant. Please note two things: Firstly, in the second sum ranging over the variables, each summand is multiplied with ξ_i . This is because not only the roots of the instances have to be deleted/created, but the entire instances (again except constants). Secondly, although the main part of the formula is enclosed in an \mathcal{O} , which means the number of operations is only of that *order*, it is multiplied with the constant \mathcal{C} . This is only done to indicate that each of those operations involves one deletion/creation.

8.1.1 Arity Preserving

Let us first consider the arity preserving case. This means, above all, that multiple occurrences of one and the same constant are represented by only one instance of class **Constant**, see Section 3.2.2, which happens in Versions 1 and 2. The number of elementary operations needed for applying r to e using the destructive, new approach is given by

$$A_1 + A_2 \quad (8.2)$$

where

$$A_1 = \mathcal{O} \left(\sum_{i=0}^D |f_{L,i} - f_{R,i}| + \sum_{i=0}^V |v_{L,i} - v_{R,i}| \xi_i \right) \cdot \mathcal{C} \quad (8.3)$$

and

$$A_2 = \mathcal{O} \left(\sum_{i=0}^D \min\{f_{L,i}, f_{R,i}\} + \sum_{i=0}^V \min\{v_{L,i}, v_{R,i}\} \right) \quad (8.4)$$

A_1 counts the number of operations needed for deleting/creating objects. Unlike in Formula 8.1, not *every* non-constant object has to be deleted/created, but only the ones that are too many on the LHS/RHS. A_2 counts the number of operations needed for putting all the other objects, i.e. the ones that are reused, to their correct positions. The most important thing to notice in A_2 is that the summands in the second sum, ranging over the variables, are *not* multiplied with the ξ_i 's.

This is because moving a subexpression (or more precisely: subtree) to a new position only requires moving the root. Please also note that in connection with Version 2, A_2 only defines an upper bound, since Version 2 always chooses the “best” mapping from the LHS of r to the RHS of r , which means that some of the operations can be avoided.

8.1.2 Type Preserving

In addition to the arity preserving case there is also the type preserving case, where multiple occurrences of constants are represented by different objects, and where the arity of functions can change. The number of elementary operations needed for applying r to e using the destructive, new approach is given by

$$T_1 + T_2 \tag{8.5}$$

where

$$T_1 = \mathcal{O} \left(\left| \sum_{i=0}^D f_{L,i} - \sum_{i=0}^D f_{R,i} \right| + \sum_{i=0}^V |v_{L,i} - v_{R,i}| \cdot (\xi_i + \gamma_i) + |c_L - c_R| \right) \cdot \mathcal{C} \tag{8.6}$$

and

$$T_2 = \mathcal{O} \left(\min \left\{ \sum_{i=0}^D f_{L,i}, \sum_{i=0}^D f_{R,i} \right\} + \sum_{i=0}^V \min \{v_{L,i}, v_{R,i}\} + \min \{c_L, c_R\} \right) \tag{8.7}$$

T_1 and T_2 are quite similar to A_1 and A_2 , respectively. The only differences between T_1 and A_1 are the additional summand $|c_L - c_R|$, the “ $\cdot(\xi_i + \gamma_i)$ ” in the sum over the variables, and the fact that no distinction is made between the arities of functions on either side. Similar for T_2 and A_2 .

8.1.3 Complexity: Conclusion

The conclusion that can be drawn from the previous considerations is the following: The efficiency of the new concept that was presented in this thesis not solely depends on the rules that are used, but also on the concrete expressions that are rewritten, or more precisely, on the size of the variable-instances. The bigger those instances, the better is the performance of the destructive approach compared to the constructive approach. This can be seen quite easily: ξ_i and γ_i only contribute to the operations treating the variables that are really deleted/created, but *not* to the others.

This observation immediately leads to the next, rather obvious conclusion: The more objects can be reused, the better. This does not only hold for variables, but for functions and constants as well, and of course only depends on the rule and not on the expression that is rewritten.

The last thing one can conclude neither depends on rules, nor on expressions, but only on the system where the programs run: Constant \mathcal{C} denotes the number of elementary operations needed for deleting/creating objects. Obviously, the more efficient these tasks are handled by the system, the less the advantage of the destructive approach over the constructive one.

8.2 Future Work

This section contains some hints for future work, separated into work that could be done for adding functionality to the programs, and work for improving their implementation.

8.2.1 Program Features

Although the various programs already carry a lot of functionality, there are still several things that could be done to enhance them. For instance, in order to support the end-user, one could think about adding an “intelligent parser” that is capable of transforming input in “natural” style into the format that is requested by the programs. A very short example would be the very simple formula $1 + 2$ that could be automatically transformed into $+ [1, 2]$. Of course, this idea has nothing to do with the topic this thesis is about, not even with rewriting at all, but still it would make the use of the systems much more convenient.

Another idea would be to put all versions together into one single program which then only calls the already existing programs if an expression is to be rewritten. For instance, the user first types in the expression and then, before it is actually rewritten, specifies the version he wants to use. Just like the first idea, this would of course have nothing to do with rewriting itself, but would make life a lot easier for the user.

One thing that could prove to be a good idea is the idea of using one unique *universal function*. If this universal function is called, say, F , then the user-defined expression $f[x]$ would not be represented as $f[x]$, but as $F[f, x]$. This would

allow two new concepts: Firstly, in rules, functions could also be variables, which means the user would have the possibility to construct 2nd order rules that can be applied to *any* function, independent of its name. Secondly, it would be possible to form well-formed expressions like $f[x][y, z]$, represented as $F[F[f, x], y, z]$.

The next idea of enhancement is maybe the most important. Almost every other rewriting system, including MAUDE ([2]) and *Mathematica* ([3]), allow the user to equip rewrite rules with additional conditions: Not only the *pattern* of the expression has to be right, but it has to fulfill a condition as well, as might be, for instance, that a variable must be instantiated with a number, and that this number has to be less than another number. Up to now, this is not possible at all in neither of the three versions developed in the frame of this thesis.

The last idea deals with the performance of the systems and is due to my supervisor Prof. Jebelean. Until now, everything the programs do depends on the rule set that has been specified before. For example, if one specifies an empty rule set and types in the expression $+ [1, 2]$, then nothing happens although it is quite clear what *should* happen: The program should compute the sum of the two Natural numbers 1 and 2. However, there is no rule that tells the program to do so! So, why not provide a small library of basic built-in functions, like addition and multiplication? They would not only make it possible to compute simple arithmetic expressions without falling back upon user-defined rewrite rules, but also make the rewriting process much more efficient at all: More than 50 rules are needed for defining addition for numbers represented as `Int`-objects! 50 rules for a rather simple arithmetic operation that can also be easily performed directly by the programs, even for arbitrarily big numbers.

8.2.2 Implementation

There is still room for improvement not only for the functionality of the programs, but also for their implementation. The first idea deals with the strategy for finding the best mapping between two expressions, as presented in Sections 5.2.4.1 and 6.2.3. Maybe there is a better approach than just trying out all possibilities. However, this only affects the efficiency of the *compilers*, but not the efficiency of the *compiled* programs!

The second idea for future work is about the matching-functions, especially the ones in Version 3. Although a better strategy than brute-force is already presented in this thesis, maybe this strategy could still be refined and made more efficient. This, in contrast to the first idea proposed in this section, would then really make the rewriting process faster! Also, one could think about arranging the rewrite

rules not in a “flat” list, but analyzing them when they are loaded and then arranging them in a more efficient way. This means, for instance, depending on the outermost symbol of the expression that is rewritten, only those rules that could really possibly match the expression are tried, and the others are ignored.

The next suggestion for future work has to do with the concept of code optimization, presented in Section 5.2.4.6. This concept can still be refined in order to handle more subtle redundancy in the command sequence.

The very last idea was raised by Dr. Wolfgang Windsteiger from the Research Institute for Symbolic Computation. It is basically built on the following consideration: Whenever an object has to be deleted because it is not needed any more, then why not only remove it and store it somewhere, but *not* delete it, since it might be needed at some point in the future when a *different* rule is applied to a *different* expression? This approach could further decrease the number of deletion/creation operations and thus improve the performance of the programs.

Appendix A

Special commands

The following is a complete list of all special commands that can be executed in “Interpreter” (Version 1) and the compiled programs of “Compiler” and “Compiler/SV” (Versions 2 and 3), together with a detailed explanation. Please be aware that some of them are only available in “Interpreter”; They are marked with “*”.

A.1 Closing the Program

The command `exit` simply closes the program. Although one could of course also just close the window by clicking on the small ✕, it is recommended to do it this way, because some dynamically allocated memory needs to be freed again.

Remark 31. The command `exit` in an input file does not close the program, but only stops processing the file.

A.2 Maximum Number of Rewriting Steps

In order to prevent the program from running forever, as might be the case when working with certain rule sets, only a maximum number of rewrite rules is applied (or less, if no rule can be applied any more). This limit, which by default is 100, can be set with the command `!n`, where n has to be a non-negative integer standing for the new value of the maximum. If n is 0, that value is set back to 100. In any case, the output of this command is exactly the maximum number of rewriting steps.

It is also possible to only retrieve, but not set, that number. Just type in `?steps`. Please note that whenever the maximum number of rewriting steps is really reached, this is indicated by the additional output (`step limit reached`).

A.3 Rewriting the previous Result

Sometimes it is quite useful to further rewrite the expression that has been rewritten before, because the maximum number of rewriting steps was reached, but the result is not yet satisfactory. In this case, just type in `%` and hit ENTER. Note that, if there is no previous result at all, the result of `%` will be `<NOTHING>`.

A.4 Integers and Natural Numbers

As promised in Section 2.3, it is possible to automatically transform Integers and Natural numbers into `succ-` and `Int` objects. The command `!makesucc` makes the program from now on apply the first transformation, the command `!makeint` the second one, and the command `!makenothing` makes the program not apply any transformation at all (this is the default setting). The output of any of those commands is a pretty-print of the respective command, and a `Succ`, `Int` or nothing in front of the “»” (input indicator) appears to indicate what the current setting is.

All of those instructions also have an inverse, so to say, which means it is possible to print `succ-` and `Int` objects in the resulting expression not as such objects, but again as normal numbers. The command `!printsucc` does the job for `succ` objects, the command `!printint` for `Int` objects, and the command `!printnothing` makes the program print everything as it is (this is the default setting). Note that the `make-` and the `print` versions are completely independent of each other, i.e. it is absolutely fine to have the setting `!makesucc` and `!printint`.

A.5 Tracing the Rewriting Process

In some cases, especially when fixing bugs in rule sets, it is necessary to know, given a concrete expression, which rule is applied in each step and what the intermediate result is. This is of course possible, namely with the command `!trace`. This command prints a “_” somewhere in front of the “»” and, whenever rewriting an

expression, not only shows the final result, but also all intermediate steps. This is done in the following way: Each step consists of three columns, where the first one describes the *number* of the respective step (starting from 1), the second one the index of the rule that is applied (indices start from 0), and the third one the resulting expression.

The command `!untrace` stops tracing.

A.6 Rewriting Strategy

Every program has two different strategies implemented that determine *which* rewrite rule is applied to the expression (or a subexpression). In some situations, more than only rule can be applied, and then, of course, the program needs to decide: Either it applies the *first* one (first w.r.t. the rule set), or it chooses a *random* one. This behavior can be controlled with the two commands `!f` and `!r`. The first one makes the program always choose the first applicable rule (default), the second one a random applicable rule; The outputs are `first` and `random`, respectively. Note that the way how the program searches the expression for suitable (rewritable) subexpressions (see Section 3.5) *cannot* be changed.

The command `?strategy` returns `first` if the current strategy always chooses the first rule, or `random` otherwise.

A.7 Rewriting an Input File

As explained in the section about input files (Section 4.2.2), it is possible to rewrite whole sets of expressions at once, if they are stored in such an input file. This works in the following way: The special command `file filename` searches the input file given by *filename* and processes it. The output is simply `Rewriting filename ... Done`; If the file does not exist, the output is `File not found`. Please note that, as usual, also the file extension has to be given.

A.8 Rewriting Algorithm*

In Version 1 it is possible to change the algorithm that is used to rewrite expressions from the new, destructive concept (reusing objects; what this thesis is all about) to the old, constructive one (creating and deleting everything). This is mainly due

to the fact that it should be possible to compare the two approaches by means of timing experiments.

The command `!reuse` chooses the first algorithm for all upcoming rewriting tasks (default), the command `!delete` the second one.

A.9 Performing Timing Experiments

There are several ways to perform timing experiments. Of course, the most important thing is to know how much time a computation takes; The C++ built-in command `clock()` is used for that purpose - Please let me refer to documentation about C++, for example [1] in order to see what exactly the behavior of the command `clock()` is (might differ on various platforms).

Sometimes it is not enough to only rewrite an expression once, because the duration of that process is too short to get a time different from just 0. So, there is also the possibility to rewrite one and the same expression N times and retrieve the time needed for the whole computation. The command `!tN` tells the program to perform such timing experiments (N has to be a non-negative integer). If N is 0, the timing experiments are aborted, otherwise, a `#` appears in front of the “»” to indicate that from now on timing experiments are performed on all subsequent expressions. The output of such an experiment consists of two lines: The first line shows the net time that was needed (only matching and rewriting, no book-keeping), the second one shows the gross time (the overall time from the start of the experiment to its end). Bookkeeping here mainly means storing the original expression once and copying it again and again, and deleting the results again and again.

The command `!t` is a shortcut: If currently no timing experiments are performed, then it has the same effect as `!t1000`; Otherwise, it has the same effect as `!t0`. The command `?t` returns the number of rewriting processes that are carried out for timing experiments, or 0, if no timing experiments are performed at all.

Since the previously mentioned experiments also include the time needed for finding a suitable rule that matches a suitable subexpression, there is a way how to really only compute the duration of the rewriting algorithm itself. For that purpose, a single rule from the rule set can be chosen and successively applied to the expression. The number of those applications is controlled in the same way as above, by the command `!tN`, and the index of the rule can be set with the command `!ri`. i has to be an integer between 0, inclusive, and the total number of rules in the rule set, exclusive, specifying the index of the rule (any other number aborts that kind of timing experiments). The output has the same structure and meaning as described above. However, if the rule does not match the expression,

the output is `Expression` is not matched by rule *i*, and no experiment is performed.

The command `?r` returns the index of the current rule, or -1 if no such rule has been specified.

Remark 32. Performing any kind of timing experiments automatically switches tracing off, and vice versa.

A.10 Printing all Symbols

The command `?symbols` returns the list of *all* (function-/constant-) symbols that have been loaded into the program up to now, either because they occur in the rule set, or they occurred in an expression.

A.11 Printing all Constants*

In Version 1, the command `?constants` returns the list of *all* constant-symbols that have been loaded into the program up to now, either because they occur in the rule set, or they occurred in an expression.

A.12 Printing all Rule Sets*

In Version 1, the command `?rulesets` returns the list of *all* rule sets that have been loaded into the program. Although only one rule set can be explicitly specified at the start of the program, that rule set might contain various `Needs[...]` commands (see Section 2.5 for more information on the `Needs[...]` command).

A.13 Printing all Rules*

In Version 1, the command `?rules` returns the list of all rules that are contained in any of the loaded rule sets, in their original order. The output format of the rules follows the definition of expressions (Section 2.2; similar to rules in rule sets):

- The outermost symbol is **Rule**
- The first argument is the number of variables used in the rule
- The second argument is the LHS of the rule; Variable i is written as $\#i$
- The third argument is the RHS of the rule; Variable i is either written as $\#i$ or as $\text{VP}j$, where j is the position (Section 2.2.2) of such a variable on the LHS; Function symbols are either written just as they are, or as $\text{FP}j(\chi)$, where j is the position of a function of the same arity on the LHS, and χ is the name of the function on the RHS.

This rather complicated way of writing down rules is due to mappings (Section 2.8): In Version 1, every rule is associated with exactly one arity preserving mapping from its LHS to its RHS (this is explained in far more detail in Chapter 4.1). $\text{VP}j$ at position k in the RHS means that $j \rightarrow k$ is an element of that mapping, and $\text{FP}m(\chi)$ at position n in the RHS means that $m \rightarrow n$ is an element.

A.14 Printing a single Rule*

In Version 1, the command `?i` returns the i -th rule in the rule set in the same format as described above. If i is less than 0 or greater or equal to the number of rules, the output is **Index out of bounds**.

Appendix B

Rule Sets

Every *reasonable* rule set that appears in this thesis, except `Sorting.txt`, `SequentCalculus.txt` and `SC.mauve`, is taken (sometimes in a slightly modified form) from [12]. The subsequent sections show all of them, together with a short description.

B.1 Addition

Rule set **Addition** can be used to define addition of `Int`-objects in terms of rewrite rules. It does not depend on any other rule set and contains in total 61 rules.

```
1 #MAKE_NOTHING
2 Rule[VarList[a], Int[a, DC[8,1]], Int[a, Int[8,9]]]
3 Rule[VarList[a,b], Int[DC[a,b], DC[8,1]], Int[a, 9]]
4 Rule[VarList[i,j], +[9, Int[i,j]], +[Int[9,9], Int[i,j]]]
5 Rule[VarList[i,j], +[Int[i,j], 9], +[Int[i,j], Int[9,9]]]
6 Rule[VarList[i,j,k,l], +[Int[i,j], Int[k,l]], Int[+[i,k], +[j,l]]]
7 Rule[VarList[i,j,k,l], Int[DC[i,j], Int[k,l]], Int[i, +[j, Int[k,l]]]]
8 Rule[VarList[i,j,k], +[i, Int[j,k]], Int[+[i,j], k]]
9 Rule[VarList[i,j,k], +[Int[j,k], i], Int[+[j,i], k]]
10 Rule[VarList[i,j,k,l], +[DC[i,j], DC[k,l]], DC[+[i,k], +[j,l]]]
11 Rule[VarList[i,j,k], +[DC[i,j], k], DC[+[i,k], j]]
12 Rule[VarList[i,j,k], +[k, DC[i,j]], DC[+[i,k], j]]
13 Rule[VarList[i,j], Int[DC[i,j], 0], Int[i, Int[j,0]]]
14 Rule[VarList[i,j], Int[DC[i,j], 9], Int[i, 0]]
15 Rule[VarList[m], +[m, 0], m]
16 Rule[VarList[m], +[0, m], m]
17
18 Rule[VarList[], +[1, 1], 2]
19 Rule[VarList[], +[1, 2], 3]
20 Rule[VarList[], +[1, 3], 4]
21 Rule[VarList[], +[1, 4], 5]
22 Rule[VarList[], +[1, 5], 6]
```

```

23 Rule[VarList[], +[1, 6], 7]
24 Rule[VarList[], +[1, 7], 8]
25 Rule[VarList[], +[1, 8], 9]
26 Rule[VarList[], +[1, 9], DC[0, 1]]
27
28 Rule[VarList[], +[2, 2], 4]
29 Rule[VarList[], +[2, 3], 5]
30 Rule[VarList[], +[2, 4], 6]
31 Rule[VarList[], +[2, 5], 7]
32 Rule[VarList[], +[2, 6], 8]
33 Rule[VarList[], +[2, 7], 9]
34 Rule[VarList[], +[2, 8], DC[0, 1]]
35 Rule[VarList[], +[2, 9], DC[1, 1]]
36
37 Rule[VarList[], +[3, 3], 6]
38 Rule[VarList[], +[3, 4], 7]
39 Rule[VarList[], +[3, 5], 8]
40 Rule[VarList[], +[3, 6], 9]
41 Rule[VarList[], +[3, 7], DC[0, 1]]
42 Rule[VarList[], +[3, 8], DC[1, 1]]
43 Rule[VarList[], +[3, 9], DC[2, 1]]
44
45 Rule[VarList[], +[4, 4], 8]
46 Rule[VarList[], +[4, 5], 9]
47 Rule[VarList[], +[4, 6], DC[0, 1]]
48 Rule[VarList[], +[4, 7], DC[1, 1]]
49 Rule[VarList[], +[4, 8], DC[2, 1]]
50 Rule[VarList[], +[4, 9], DC[3, 1]]
51
52 Rule[VarList[], +[5, 5], DC[0, 1]]
53 Rule[VarList[], +[5, 6], DC[1, 1]]
54 Rule[VarList[], +[5, 7], DC[2, 1]]
55 Rule[VarList[], +[5, 8], DC[3, 1]]
56 Rule[VarList[], +[5, 9], DC[4, 1]]
57
58 Rule[VarList[], +[6, 6], DC[2, 1]]
59 Rule[VarList[], +[6, 7], DC[3, 1]]
60 Rule[VarList[], +[6, 8], DC[4, 1]]
61 Rule[VarList[], +[6, 9], DC[5, 1]]
62
63 Rule[VarList[], +[7, 7], DC[4, 1]]
64 Rule[VarList[], +[7, 8], DC[5, 1]]
65 Rule[VarList[], +[7, 9], DC[6, 1]]
66
67 Rule[VarList[], +[8, 8], DC[6, 1]]
68 Rule[VarList[], +[8, 9], DC[7, 1]]
69
70 Rule[VarList[], +[9, 9], DC[8, 1]]
71
72 Rule[VarList[m,n], +[m, n], +[n, m]]

```

B.2 Multiplication

Rule set **Multiplication** can be used to define multiplication of **Int**-objects in terms of rewrite rules. It depends on rule set **Addition** and contains 47 new rules, making in total 108 rules.

```

1 Needs[Addition.txt]
2 #MAKE_NOTHING
3 Rule[VarList[d1, d2, c1, c2],
4     *[Int[d1,c1], Int[d2,c2]], Int[*[d1,d2], +[**[d1,c2], *[c1,Int[d2,c2]]]]]
5 Rule[VarList[d1, d2, c2], **[d1, Int[d2,c2]], Int[*[d1,d2], **[d1,c2]]]
6 Rule[VarList[m], *[m, 0], 0]
7 Rule[VarList[m], *[0, m], 0]
8 Rule[VarList[m], *[m, 1], m]
9 Rule[VarList[m], *[1, m], m]
10 Rule[VarList[d], **[d, 0], 0]
11 Rule[VarList[d], **[0, d], 0]
12 Rule[VarList[d], **[d, 1], d]
13 Rule[VarList[d], **[1, d], d]
14
15 Rule[VarList[], *[2, 2], 4]
16 Rule[VarList[], *[2, 3], 6]
17 Rule[VarList[], *[2, 4], 8]
18 Rule[VarList[], *[2, 5], DC[0, 1]]
19 Rule[VarList[], *[2, 6], DC[2, 1]]
20 Rule[VarList[], *[2, 7], DC[4, 1]]
21 Rule[VarList[], *[2, 8], DC[6, 1]]
22 Rule[VarList[], *[2, 9], DC[8, 1]]
23
24 Rule[VarList[], *[3, 3], 9]
25 Rule[VarList[], *[3, 4], DC[2, 1]]
26 Rule[VarList[], *[3, 5], DC[5, 1]]
27 Rule[VarList[], *[3, 6], DC[8, 1]]
28 Rule[VarList[], *[3, 7], DC[1, 2]]
29 Rule[VarList[], *[3, 8], DC[4, 2]]
30 Rule[VarList[], *[3, 9], DC[7, 2]]
31
32 Rule[VarList[], *[4, 4], DC[6, 1]]
33 Rule[VarList[], *[4, 5], DC[0, 2]]
34 Rule[VarList[], *[4, 6], DC[4, 2]]
35 Rule[VarList[], *[4, 7], DC[8, 2]]
36 Rule[VarList[], *[4, 8], DC[2, 3]]
37 Rule[VarList[], *[4, 9], DC[6, 3]]
38
39 Rule[VarList[], *[5, 5], DC[5, 2]]
40 Rule[VarList[], *[5, 6], DC[0, 3]]
41 Rule[VarList[], *[5, 7], DC[5, 3]]
42 Rule[VarList[], *[5, 8], DC[0, 4]]
43 Rule[VarList[], *[5, 9], DC[5, 4]]
44
45 Rule[VarList[], *[6, 6], DC[6, 3]]
46 Rule[VarList[], *[6, 7], DC[2, 4]]
47 Rule[VarList[], *[6, 8], DC[8, 4]]
48 Rule[VarList[], *[6, 9], DC[4, 5]]
49
50 Rule[VarList[], *[7, 7], DC[9, 4]]
51 Rule[VarList[], *[7, 8], DC[6, 5]]
52 Rule[VarList[], *[7, 9], DC[3, 6]]
53
54 Rule[VarList[], *[8, 8], DC[4, 6]]
55 Rule[VarList[], *[8, 9], DC[2, 7]]
56
57 Rule[VarList[], *[9, 9], DC[1, 8]]
58
59 Rule[VarList[m,n], *[m, n], *[n, m]]

```


B.3 NatAddition

Rule set `NatAddition` can be used to define addition of `succ`-objects in terms of rewrite rules. It does not depend on any other rule set and contains in total only two rules.

```
1 #MAKE_NOTHING
2 Rule[VarList[m, n], +[m, succ[n]], succ[+[m, n]]]
3 Rule[VarList[m], +[m, 0], m]
```

B.4 NatMultiplication

Rule set `NatMultiplication` can be used to define multiplication of `succ`-objects in terms of rewrite rules. It depends on rule set `NatAddition` and contains two new rules, making in total only four rules.

```
1 Needs[NatAddition.txt]
2 #MAKE_NOTHING
3 Rule[VarList[m, n], *[m, succ[n]], +[*[m, n], m]]
4 Rule[VarList[m], *[m, 0], 0]
```

B.5 NatSum

Rule set `NatSum` can be used to define summation of the first n Natural numbers, represented as `succ`-objects, in terms of rewrite rules. It depends on rule set `NatAddition` and contains two new rules, making in total four rules.

```
1 Needs[NatAddition.txt]
2 #MAKE_NOTHING
3 Rule[VarList[], Sum[0], 0]
4 Rule[VarList[m], Sum[succ[m]], +[succ[m], Sum[m]]]
```

B.6 PolyAddition

Rule set `PolyAddition` can be used to define addition of univariate polynomials in terms of rewrite rules. Polynomials have to be represented in the following way:

If f is defined as

$$f = \sum_{i=0}^d c_i x^i$$

then, in order to be treated correctly by this rule set, f has to be written as

$$\text{Poly}[c_0, \text{Poly}[c_1, \dots \text{Poly}[c_d, 0] \dots]]$$

and the coefficients c_i have to be represented as `Int`-objects.

This rule set depends on rule set **Addition** and contains three new rules, making in total 64 rules. Note that the rules from **Addition** are not included at the beginning, but at the very end.

```

1 #MAKE_NOTHING
2 Rule[VarList[c1,c2,p2], +[Poly[c1,Int[0,0]], Poly[c2,p2]], Poly[+[c1,c2], p2]]
3 Rule[VarList[c1,c2,p1], +[Poly[c1,p1], Poly[c2,Int[0,0]]], Poly[+[c1,c2], p1]]
4 Rule[VarList[p1,p2,c1,c2], +[Poly[c1,p1], Poly[c2,p2]], Poly[+[c1,c2], +[p1,p2]]]
5 Needs[Addition.txt]
```

B.7 Sorting

Rule set **Sorting** can be used to sort arbitrary sequences of Natural numbers, represented as `succ`-objects, where the sequences do not have *holes*: Every number between the sequence-minimum and -maximum has to be part of the sequence as well. Otherwise, the result is not necessarily sorted.

This rule set does not depend on any other rule set and contains in total 2 rules.

```

1 #MAKE_NOTHING
2 Rule[VarList[h_, c_, t_, a], sort[h_, succ[a], c_, a, t_],
3     sort[h_, a, c_, succ[a], t_]]
4 Rule[VarList[x_], sort[x_], sorted[x_]]
```

B.8 SequentCalculus

Rule set **SequentCalculus** can be used to determine the truth value of propositional formulae. A well-formed (w.r.t. this rule set) propositional formula is an expression where only function symbols `not` with arity 1, and `and`, `or`, `impl` and `equiv` with arity 2 occur. Constants `TRUE` and `FALSE` are used to represent the two truth values, all other constants are regarded as propositional variables.

This rule set does not depend on any other rule set and contains in total 16 rules.

```

1  #MAKE_NOTHING
2
3  //initialization
4  Rule[VarList[A], prove[A], prooftree[proofsit[kb[]],goal[A]]]
5
6  //terminal situation: proofsit proved
7  Rule[VarList[rest_, kb1_, kb2_, goal1_, goal2_, A],
8        prooftree[proofsit[kb[kb1_,A,kb2_],goal[goal1_,A,goal2_]], rest_],
9        prooftree[rest_]]
10
11 //terminal situation: FALSE in kb
12 Rule[VarList[rest_, kb1_, kb2_, goal_],
13        prooftree[proofsit[kb[kb1_,FALSE,kb2_],goal[goal_]], rest_],
14        prooftree[rest_]]
15
16 //terminal situation: TRUE in goal
17 Rule[VarList[rest_, kb_, goal1_, goal2_],
18        prooftree[proofsit[kb[kb_],goal[goal1_,TRUE,goal2_]], rest_],
19        prooftree[rest_]]
20
21 //not[A] in kb
22 Rule[VarList[rest_, kb1_, kb2_, goal_, A],
23        prooftree[proofsit[kb[kb1_,not[A],kb2_],goal[goal_]], rest_],
24        prooftree[proofsit[kb[kb1_,kb2_],goal[A,goal_]], rest_]]
25
26 //not[A] in goal
27 Rule[VarList[rest_, kb_, goal1_, goal2_, A],
28        prooftree[proofsit[kb[kb_],goal[goal1_,not[A],goal2_]], rest_],
29        prooftree[proofsit[kb[A,kb_],goal[goal1_,goal2_]], rest_]]
30
31 //and[A,B] in kb
32 Rule[VarList[rest_, kb1_, kb2_, goal_, A, B],
33        prooftree[proofsit[kb[kb1_,and[A,B],kb2_],goal[goal_]], rest_],
34        prooftree[proofsit[kb[A,B,kb1_,kb2_],goal[goal_]], rest_]]
35
36 //or[A,B] in goal
37 Rule[VarList[rest_, kb_, goal1_, goal2_, A, B],
38        prooftree[proofsit[kb[kb_],goal[goal1_,or[A,B],goal2_]], rest_],
39        prooftree[proofsit[kb[kb_],goal[A,B,goal1_,goal2_]], rest_]]
40
41 //impl[A,B] in goal
42 Rule[VarList[rest_, kb_, goal1_, goal2_, A, B],
43        prooftree[proofsit[kb[kb_],goal[goal1_,impl[A,B],goal2_]], rest_],
44        prooftree[proofsit[kb[A,kb_],goal[B,goal1_,goal2_]], rest_]]
45
46 //equiv[A,B] in kb
47 Rule[VarList[rest_, kb1_, kb2_, goal_, A, B],
48        prooftree[proofsit[kb[kb1_,equiv[A,B],kb2_],goal[goal_]], rest_],
49        prooftree[proofsit[kb[kb1_,kb2_,impl[A,B],impl[B,A]],goal[goal_]], rest_]]
50
51 //or[A,B] in kb
52 Rule[VarList[rest_, kb1_, kb2_, goal_, A, B],
53        prooftree[proofsit[kb[kb1_,or[A,B],kb2_],goal[goal_]], rest_],
54        prooftree[proofsit[kb[A,kb1_,kb2_],goal[goal_]],
55                  proofsit[kb[B,kb1_,kb2_],goal[goal_]], rest_]]
56
57 //and[A,B] in goal
58 Rule[VarList[rest_, kb_, goal1_, goal2_, A, B],
59        prooftree[proofsit[kb[kb_],goal[goal1_,and[A,B],goal2_]], rest_],
60        prooftree[proofsit[kb[kb_],goal[A,goal1_,goal2_]],
61                  proofsit[kb[kb_],goal[B,goal1_,goal2_]], rest_]]

```

```

62
63 //impl[A,B] in kb
64 Rule[VarList[rest_, kb1_, kb2_, goal_, A, B],
65     prooftree[proofsit[kb[kb1_,impl[A,B],kb2_],goal[goal_]], rest_],
66     prooftree[proofsit[kb[kb1_,kb2_],goal[A,goal_]],
67         proofsit[kb[B,kb1_,kb2_],goal[goal_]], rest_]]
68
69 //equiv[A,B] in goal
70 Rule[VarList[rest_, kb_, goal1_, goal2_, A, B],
71     prooftree[proofsit[kb[kb_],goal[goal1_,equiv[A,B],goal2_]], rest_],
72     prooftree[proofsit[kb[kb_],goal[impl[A,B],goal1_,goal2_]],
73         proofsit[kb[kb_],goal[impl[B,A],goal1_,goal2_]], rest_]]
74
75 //terminal situation: disproved
76 Rule[VarList[head, rest_], prooftree[head, rest_], disproved]
77
78 //terminal situation: proved
79 Rule[VarList[], prooftree[], proved]

```

B.9 SC

SC cannot be used together with the programs of this thesis, but rather together with the rewriting-system Maude ([2]). Just as rule set **SequentCalculus**, **SC** consists of rules (more precisely: equations) that define the propositional sequent calculus. Moreover, it was attempted to be as close as possible to rule set **SequentCalculus** and therefore the order and meaning of the equations was left unchanged. For a thorough description of the syntax and semantics of modules in Maude let me refer to [7].

One interesting aspect of the following listing is the way how sequence variables in **SequentCalculus** are transformed into individual variables in **SC**: The knowledge base in a proof situation, for instance, is nothing else than a sequence of formulae, built from the binary operator `_ _`. This operator is both associative and commutative, which means that in the equations one can always assume that the formula one wants to have a look at is the very first one, even if this is not the case. Maude is smart enough to realize, due to the operator's attributes, that the order of formulae does not matter.

```

1 fmod SC is
2   sorts Formula FList ProofSit SList .
3   subsort Formula < FList .
4   subsort ProofSit < SList .
5
6   op a : -> Formula .
7   op b : -> Formula .
8   op t : -> Formula .
9   op f : -> Formula .
10  op not_ : Formula -> Formula .
11  op _and_ : Formula Formula -> Formula .
12  op _or_ : Formula Formula -> Formula .

```

```

13  op _impl_ : Formula Formula -> Formula .
14  op _equiv_ : Formula Formula -> Formula .
15
16  op _ _ : FList FList -> FList [assoc comm] .
17  op _>_ : FList FList -> ProofSit .
18  op _;_ : SList SList -> SList [assoc comm] .
19  op prove : Formula -> SList .
20  op proved : -> SList .
21  op disproved : -> SList .
22
23  vars G H : Formula .
24  vars L1 L2 : FList .
25  var P : SList .
26
27  eq prove(G) = (t > G f) ; proved .
28  eq (G L1 > G L2) ; P = P .
29  eq (f L1 > L2) ; P = P .
30  eq (L1 > t L2) ; P = P .
31  eq (not(G) L1 > L2) ; P = (L1 > G L2) ; P .
32  eq (L1 > not(G) L2) ; P = (G L1 > L2) ; P .
33  eq ((G and H) L1 > L2) ; P = (G H L1 > L2) ; P .
34  eq (L1 > (G or H) L2) ; P = (L1 > G H L2) ; P .
35  eq (L1 > (G impl H) L2) ; P = (G L1 > H L2) ; P .
36  eq ((G equiv H) L1 > L2) ; P = (L1 (G impl H) (H impl G) > L2) ; P .
37  eq ((G or H) L1 > L2) ; P = (G L1 > L2) ; (H L1 > L2) ; P .
38  eq (L1 > (G and H) L2) ; P = (L1 > G L2) ; (L1 > H L2) ; P .
39  eq ((G impl H) L1 > L2) ; P = (L1 > G L2) ; (H L1 > L2) ; P .
40  eq (L1 > (G equiv H) L2) ; P = (L1 > (G impl H) L2) ; (L1 > (H impl G) L2) ; P .
41  eq (L1 > L2) ; P = disproved [owise] .
42 endfm

```

Bibliography

- [1] C++ Reference. <http://www.cplusplus.com>.
- [2] The Maude System. <http://maude.cs.uiuc.edu>.
- [3] Wolfram Mathematica. <http://www.wolfram.com/mathematica/>.
- [4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.
- [5] M. Bezem, J.W. Klop, and Roel de Vrijer. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.
- [6] Alonzo Church. *Introduction to Mathematical Logic*. Princeton Landmarks in Mathematics and Physics. Princeton University Press, 1996.
- [7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.6)*, 2011. <http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf>.
- [8] S. Eker. Term Rewriting with Operator Evaluation Strategy. In *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.
- [9] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, third edition, 1997.
- [10] T. Kutsia and B. Buchberger. Predicate Logic with Sequence Variables and Sequence Function Symbols. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Proceedings of the 3rd International Conference on Mathematical Knowledge Management, MKM'04*, volume 3119 of *Lecture Notes in Computer Science*, pages 205–219, Białowieża, Poland, Sep 19–21 2004. Springer Verlag.
- [11] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, 2002.

-
- [12] Christina Paul. A Rewriting System in C++. Master's thesis, JKU Linz, 2009.
 - [13] Guy L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990.
 - [14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.
 - [15] F. Winkler. *Polynomial Algorithms in Computer Algebra*. Texts and Monographs in Symbolic Computation. Springer-Verlag Wien New York, first edition, 1996.