



Diploma Thesis

**Quantifier Elimination-based Parametric
Solving in Term Algebras**

Christian Hoffelner

July 25, 2005

supervised and refereed by Dr. Thomas Sturm
co-refereed by Prof. Dr. Volker Weispfenning

Abstract

Quantifier elimination refers to the process of transforming a first-order formula φ into a quantifier-free formula φ' so that φ and φ' are equivalent w.r.t. the underlying structure. On the basis of [SW02], we present a quantifier elimination procedure for term algebras over suitably expanded finite first-order languages. The procedure, together with some extensions for increasing its efficiency, is implemented within the logic package REDLOG, which is part of the computer algebra system REDUCE.

The elimination technique used here was introduced by Weispfenning in 1988 and works by substitution of finitely many test terms for the quantified variables (cf. [Wei88]). The main advantage of this method is its practicability for highly parametric problems. Furthermore, it offers a straightforward extension to quantifier elimination with answers for an outermost existential quantifier block.

Due to the fact that the first-order theory of term algebras \mathbf{T} does not admit quantifier elimination - except for exactly one type of languages \mathcal{L} - we consider slightly expanded languages \mathcal{L}^* . There are several possibilities for this expansion in order to make quantifier elimination generally possible. According to [SW02], we use a purely functional expansion, so that the expanded structure \mathbf{T}^* is still an algebra. Having quantifier elimination on-hand in that way expanded term algebras is useful for miscellaneous application areas, e.g. symbolic constraint solving or logic programming.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 2 | Preliminaries | 3 |
| 2.1 | Term Algebras | 3 |
| 2.2 | Quantifier Elimination | 6 |
| 3 | Elimination Procedure | 11 |
| 3.1 | Preparatory Work | 11 |
| 3.1.1 | Normal Forms | 11 |
| 3.1.2 | Language Types | 15 |
| 3.2 | Elimination | 16 |
| 3.2.1 | Quantifier Elimination by Substitution of Parametric Test Terms | 17 |
| 3.2.2 | Depth Bounds | 18 |
| 3.2.3 | Substitutions | 21 |
| 3.2.4 | Algorithm | 23 |
| 3.3 | Complexity | 24 |
| 4 | Implementation | 29 |
| 4.1 | The REDLOG-package <code>talp</code> | 29 |
| 4.2 | Simplifications | 31 |
| 4.2.1 | Standard Simplifier | 33 |
| 4.2.2 | Special Simplifier | 41 |
| 4.3 | Elimination Algorithms | 45 |
| 4.3.1 | Quantifier Elimination by Substitution of Parametric Test Terms | 45 |
| 4.3.2 | Deep Gauss Elimination | 60 |
| 4.3.3 | Strategies for Quantifier Permutations | 62 |
| 4.4 | Illustration | 63 |

Contents

| | | |
|----------|-------------------------------|-----------|
| 5 | Application Examples | 65 |
| 5.1 | Natural Numbers | 65 |
| 5.2 | Binary Trees | 66 |
| 5.3 | Parametric Problems | 66 |
| 6 | Summary | 69 |

1 Introduction

The notion of *quantifier elimination* refers to the process of transforming a first-order formula φ into a quantifier-free formula φ' so that φ and φ' are equivalent w.r.t. the underlying structure. A procedure computing φ' from φ is called a *quantifier elimination procedure*.

In the first half of the 20th century many elimination procedures were found for different structures. The structure \mathbf{R} of real numbers, i.e., the structure with universe \mathbb{R} , together with the usual operations and relations, is by far the most important one for practical applications like constraint solving, optimization, automatic theorem proving or scheduling problems. Apart from the reals, there are many other structures within which the availability of quantifier elimination procedures would be helpful. One example of such a structure is the class of absolutely free algebras (aka. *term algebras*). Unfortunately, these structures do not admit quantifier elimination - except for very simple languages. Due to this fact, different expansions for the underlying languages have been introduced in order to make quantifier elimination generally possible (see e.g. [RV01],[Hod93]). Most of these expansions have in common that, because of the addition of relations, the resulting expanded structure is not longer an algebra. [SW02], which is the basis of this work, gets by without adding relations so that the underlying structure remains an algebra.

Having quantifier elimination on-hand in that way expanded term algebras is useful for several application areas especially in computer science. In logic programming, the elimination procedure can take the part of the constraint solver (cf. [Stu02]). Unifiers (resp. disunifiers) are fundamental tools for solving symbolic constraints, i.e. equations and inequalities between terms (cf. [CL89]). These problems can also be solved in corresponding term algebras using the provided elimination procedure.

Based on [SW02], we present a quantifier elimination procedure for purely functional expanded term algebras. The procedure, together with some extensions for increasing its efficiency, is implemented within the package REDLOG¹ [DS97a] of

¹REDLOG stands for REDuce LOGic system - a comprehensive logic package developed by Andreas Dolzmann and Thomas Sturm at the University of Passau. It is part of the computer algebra system REDUCE since Version 3.7. For more information consult <http://www.fmi.uni-passau.de/~redlog/>.

1 Introduction

the computer algebra system REDUCE².

The plan of this thesis is as follows: In order to get a formal framework for quantifier elimination in expanded term algebras, chapter 2 introduces the necessary background. The elimination procedure and related terms are described in chapter 3. Chapter 4 focusses on the implementation of the procedures within the above mentioned framework. Chapter 5 provides a collection of application examples accomplished by our implementation. The last chapter finally summarizes the main results of this thesis.

²Consult <http://www.reduce-algebra.com/> for more information.

2 Preliminaries

In this chapter, we introduce a formal framework for quantifier elimination in term algebras. In section 2.1, we start with some elementary definitions used throughout this thesis. We will focus on the basic notations necessary for understanding the procedure. For a comprehensive background see [Hod93]. In section 2.2, we describe the general notion of quantifier elimination and reveal the specific problems arising within the theory of term algebras.

2.1 Term Algebras

We start with a brief introduction of the first-order theory of term algebras. The notations used here closely follow those used in [Dol02].

Definition 2.1.1 (Algebraic Language). *An algebraic language $\mathcal{L} = \{\mathcal{F}, \alpha\}$ consists of a set \mathcal{F} of function symbols and a mapping $\alpha : \mathcal{F} \rightarrow \mathbb{N}$. For each symbol $s \in \mathcal{F}$, $\alpha(s)$ denotes the symbol's arity. Those elements c of \mathcal{F} , for which $\alpha(c) = 0$ are called constants. A language \mathcal{L} is called finite, if \mathcal{F} is finite. \blacklozenge*

In the following, we exclusively consider finite algebraic languages \mathcal{L} with at least one constant and at least one function symbol of positive arity. We abbreviate the notation for such languages $\mathcal{L} = \{\{f_1, \dots, f_m\}, \alpha\}$ with $\alpha(f_1) = k_1, \dots, \alpha(f_m) = k_m$ as $\mathcal{L} = \{f_1^{(k_1)}, \dots, f_m^{(k_m)}\}$. Furthermore, we define \mathcal{X} to be an infinite set of variables. Based on these fundamental definitions we are able to define \mathcal{L} -terms:

Definition 2.1.2 (\mathcal{L} -Terms). *The set T of \mathcal{L} -terms is inductively defined as follows:*

- $\{f \mid f \in \mathcal{F}, \alpha(f) = 0\} \in T$, i.e. every constant symbol is an \mathcal{L} -term.
- $\mathcal{X} \subseteq T$, i.e. every variable $x \in \mathcal{X}$ is an \mathcal{L} -term.
- If $t_1, \dots, t_n \in T$ and if f is an n -ary function symbol, then also $f(t_1, \dots, t_n)$ is an \mathcal{L} -term. \blacklozenge

Definition 2.1.3 (\mathcal{L} -Equations). *We define the set $\mathcal{E}_{\mathcal{L}}$ of \mathcal{L} -equations to be the set of all expressions of the form*

$$t = t',$$

where t, t' are \mathcal{L} -terms. \blacklozenge

2 Preliminaries

Let t, t_1, \dots, t_n be \mathcal{L} -terms and $x_1, \dots, x_n \in \mathcal{X}$. We denote by $t[t_1/x_1, \dots, t_n/x_n]$ the \mathcal{L} -term emerging from t by simultaneously replacing all occurrences of x_i within t by t_i . For an \mathcal{L} -equation φ , we denote by $\varphi[t_1/x_1, \dots, t_n/x_n]$ the \mathcal{L} -equation emerging from φ by simultaneously replacing all occurrences of x_i within φ by t_i . We define $\mathcal{V}(t)$ to be the set of variables occurring within t . If $\mathcal{V}(t) \subseteq \{x_1, \dots, x_n\}$, we call $t(x_1, \dots, x_n)$ an *extended \mathcal{L} -term* with extension (x_1, \dots, x_n) . \mathcal{V} and extensions are similarly defined for \mathcal{L} -equations.

In order to define arbitrary first-order formulas, we make use of special characters $\neg, \wedge, \vee, \exists, \forall$, and assign to them the meaning of the well-known logic symbols. Additionally, we make use of the Boolean constants "true" and "false".

Definition 2.1.4 (First-order Formulas). *The set \mathcal{FOF} of first-order formulas is inductively defined as follows:*

- true as well as false are elements of \mathcal{FOF} .
- $\mathcal{E}_{\mathcal{L}} \subseteq \mathcal{FOF}$.
- If $\varphi \in \mathcal{FOF}$, then also $\neg(\varphi) \in \mathcal{FOF}$.
- If $\varphi, \psi \in \mathcal{FOF}$, then also $(\varphi) \wedge (\psi) \in \mathcal{FOF}$ and $(\varphi) \vee (\psi) \in \mathcal{FOF}$.
- If $\varphi \in \mathcal{FOF}$, $x \in \mathcal{X}$, then also $\exists x(\varphi) \in \mathcal{FOF}$ and $\forall x(\varphi) \in \mathcal{FOF}$. ◆

To improve the readability of first-order formulas, we will omit parenthesis whenever convenient.

We are now able to formulate inequalities between \mathcal{L} -terms in terms of negated equations. In the following, we will represent these negated equations using the binary relation symbol " \neq ", i.e., we represent negated equations like $\neg(f(x) = y)$ as $f(x) \neq y$. For $\mathcal{E}_{\mathcal{L}}$ we define $\overline{\mathcal{E}_{\mathcal{L}}}$ to be the set of negated \mathcal{L} -equations. We define the set $\mathcal{B}_{\mathcal{L}}$ of *base formulas* to be the union of $\mathcal{E}_{\mathcal{L}}$ and $\overline{\mathcal{E}_{\mathcal{L}}}$.

For an \mathcal{L} -term t , we denote by $|t|$ the *length* of t as a word over \mathcal{L} -symbols. For instance, the term $f(g(a, x))$ has length 9. For \mathcal{L} -formulas φ , $|\varphi|$ is defined analogously. We denote the *depth* of an \mathcal{L} -term t considered as a tree by $\Delta(t)$. We define $\Delta(c)$ to be 0 for \mathcal{L} -constants or variables c . For \mathcal{L} -formulas φ , $\Delta(\varphi)$ denotes the maximum depth among all terms occurring in φ . For sets S of \mathcal{L} -terms or \mathcal{L} -formulas, $\Delta(S)$ is defined accordingly. For arbitrary sets S , we denote the number of elements contained in S by $\#S$. For an arbitrary \mathcal{L} -formula φ , we denote the number of equations contained in φ by $eq(\varphi)$.

Within a given formula φ , an *occurrence* of a variable x inside the scope of a quantifier $\exists x$ or $\forall x$ is called *bound*, all other occurrences of x within φ are called *free*. We denote by $\mathcal{V}_f(\varphi)$ ($\mathcal{V}_b(\varphi)$) the set of free (bound) variables occurring in φ . If $\mathcal{V}_f(\varphi) \subseteq \{x_1, \dots, x_n\}$, then we call φ an *extended \mathcal{L} -formula* with extension $\{x_1, \dots, x_n\}$. For extended formulas $\varphi(x_1, \dots, x_n)$, \mathcal{V}_f and \mathcal{V}_b are defined

analogously. For a first-order \mathcal{L} -formula φ , we denote by $\varphi[t_1/x_1, \dots, t_n/x_n]$ the \mathcal{L} -formula emerging from φ by simultaneously replacing all free occurrences of x_i within φ by t_i .

Let φ be an \mathcal{L} -formula. A series of successive like quantifiers is called a (*quantifier*) *block*. φ is called *prenex*, if it has several blocks of alternating quantifiers in front of a quantifier-free formula ψ . We call ψ the *matrix* of φ .

Definition 2.1.5 (Algebra). Let $\mathcal{L} = \{f_1^{(k_1)}, \dots, f_m^{(k_m)}\}$ be a finite algebraic language. An algebra \mathbf{A} is a tuple $(A, i_{\mathcal{F}})$, where A is a non-empty set, called the universe of \mathbf{A} , and $i_{\mathcal{F}}$ is a mapping (interpretation), which maps each function symbol of arity n to a corresponding function:

$$i_{\mathcal{F}} : \mathcal{F} \rightarrow \bigcup_{n \in \mathbb{N}} A^{(A^n)}.$$

For a function symbol f of arity n , we denote by $f^{\mathbf{A}} = i_{\mathcal{F}}(f) \in A^{(A^n)}$ the function named by f . \blacklozenge

We abbreviate the notation for structures \mathbf{A} over finite algebraic languages \mathcal{L} as (A, g_1, \dots, g_m) , where A is the underlying universe, and the g_i are functions.

For an extended \mathcal{L} -formula $\varphi(x_1, \dots, x_n)$ which holds in \mathbf{A} at a certain position (a_1, \dots, a_n) , we write $\mathbf{A} \models \varphi(a_1, \dots, a_n)$ which means that “ (a_1, \dots, a_n) satisfies φ in \mathbf{A} ” or “ φ is true in \mathbf{A} at the position (a_1, \dots, a_n) ”. An \mathcal{L} -sentence φ is a formula with $\mathcal{V}_f(\varphi) = \emptyset$. Here, the logical value of φ is independent of both, the position and the extension. In this case, we write $\mathbf{A} \models \varphi$ to indicate that “ φ is generally true in \mathbf{A} ”.

On the basis of this short survey of first-order logic, we are able to define a special kind of algebra which is the basis for all further considerations: the *term algebra*.

Definition 2.1.6 (Term Algebra). Let $\mathcal{L} = \{f_1^{(k_1)}, \dots, f_n^{(k_n)}\}$ be a finite algebraic language with at least one function symbol of arity 0 and at least one function symbol of positive arity. Let \mathcal{X} be an infinite set of variables. The term algebra \mathbf{T} of \mathcal{L} with basis \mathcal{X} is the structure (T, f_1, \dots, f_n) , where the universe T is the set of all variable-free \mathcal{L} -terms, and the f_i are trivially interpreted function symbols, i.e. $f_i^{\mathbf{T}} = f_i$. \blacklozenge

We conclude this section by giving an example of a first-order formula within \mathbf{T} .

Example 2.1.1 (\mathcal{L} -formulas). Let $\mathcal{L} = \{a^{(0)}, b^{(0)}, f^{(1)}, g^{(2)}\}$ be an algebraic language and let the term algebra $\mathbf{T} = (T, a, b, f, g)$ be its interpretation. So far we have to deal with first-order formulas like

$$\varphi(u, x) \equiv \left(\exists x (g(x, a) \neq g(u, x) \vee f(b) = u) \wedge f(f(u)) \neq x \right) (u, x)$$

2 Preliminaries

where $\Delta(\varphi(u, x)) = 2$, $\mathcal{V}(\varphi(u, x)) = \{u, x\}$, $\mathcal{V}_b(\varphi(u, x)) = \{x\}$, $\mathcal{V}_f(\varphi(u, x)) = \{u, x\}$. \diamond

2.2 Quantifier Elimination

In this section, we introduce the central notion of quantifier elimination. The first-order theory of term algebras admits quantifier elimination only for very special kinds of languages. We will show the requirements needed to generalize its applicability.

Within a given structure \mathbf{A} , two formulas φ and ψ are called \mathbf{A} -equivalent iff $\mathbf{A} \models (\varphi \leftrightarrow \psi)$.

Definition 2.2.1 (Quantifier Elimination). *Let \mathcal{L} be a language, and let \mathbf{A} be an \mathcal{L} -structure. We say that \mathbf{A} admits quantifier elimination, if for every \mathcal{L} -formula φ there is an \mathcal{L} -formula φ' with the following properties:*

1. φ' is quantifier-free
2. $\mathcal{V}_f(\varphi') \subseteq \mathcal{V}_f(\varphi)$
3. φ and φ' are \mathbf{A} -equivalent

An algorithm computing φ' from φ within \mathbf{A} is called a quantifier elimination procedure for \mathbf{A} . \diamond

Definition 2.2.2 (Decision Procedure). *A decision procedure for an \mathcal{L} -structure \mathbf{A} is an algorithm which takes arbitrary \mathcal{L} -sentences φ as input and decides whether $\mathbf{A} \models \varphi$ or $\mathbf{A} \not\models \varphi$. \diamond*

Note that if we can decide whether a variable-free equation is *true* or *false* within a given structure \mathbf{A} , then a quantifier elimination procedure for \mathbf{A} leads to a decision procedure for \mathbf{A} using truth tables.

Definition 2.2.3 (Definable Sets). *Let $\varphi(x_1, \dots, x_n)$ be an extended \mathcal{L} -formula and let \mathbf{A} be an \mathcal{L} -structure. We call $\varphi_{\top}^{\mathbf{A}} := \{(a_1, \dots, a_n) \in A^n \mid \mathbf{A} \models \varphi(a_1, \dots, a_n)\}$ the set defined by φ within \mathbf{A} . A subset M of A^n is called (quantifier-free) definable within \mathbf{A} , if there exists an extended (quantifier-free) \mathcal{L} -formula $\varphi(x_1, \dots, x_n)$ such that $M = \varphi_{\top}^{\mathbf{A}}$. \diamond*

Obviously, \mathbf{A} -equivalent extended \mathcal{L} -formulas $\varphi(x_1, \dots, x_n)$ and $\varphi'(x_1, \dots, x_n)$ define the same sets, i.e. for \mathbf{A} -equivalent formulas φ and φ' we have $\varphi_{\top}^{\mathbf{A}} = \varphi'_{\top}^{\mathbf{A}}$. Conversely, the equality of these sets also implies the equivalence of φ and φ' . This leads us to the following theorem (cf. [Wei03]):

Theorem 2.2.1. *An \mathcal{L} -structure \mathbf{A} admits quantifier elimination iff every definable set $M \subseteq A^n$ within \mathbf{A} is also quantifier-free definable within \mathbf{A} . \square*

As the following example illustrates, this theorem can be used to show that certain structures do not admit quantifier elimination.

Example 2.2.1 (Definable Sets). *Consider $\mathcal{L} = \{0^{(0)}, 1^{(0)}, +^{(2)}, -^{(1)}, \cdot^{(2)}\}$ and $\mathbf{R} = (\mathbb{R}, 0, 1, +, -, \cdot)$ as its well-known interpretation. We can easily show that \mathbf{R} does not admit quantifier elimination by considering the \mathcal{L} -formula*

$$\varphi(y) \equiv \exists x(x \cdot x = y)(x, y).$$

Obviously, $\varphi(y)$ defines the set of non-negative real numbers, i.e. the half-open interval $[0, \infty)$. On the other hand, extended \mathcal{L} -equations can be described as $f(y) = 0$ for polynomials $f \in \mathbb{Z}[y]$. Each of these equations defines one of the sets \mathbb{R} , \emptyset or a finite set $\{r_1, \dots, r_n\}$ of real numbers r_i . Arbitrary extended quantifier-free formulas define boolean combinations of such sets, leading to finite or co-finite sets of real numbers, but never to the infinite and co-infinite set $[0, \infty)$. Hence quantifier elimination within \mathbf{R} is impossible. \diamond

Unfortunately, we can argue similarly for term algebras \mathbf{T} introduced in Definition 2.1.6, by considering definable sets in \mathbf{T} . The proofs of the following two theorems can be found in [SW02].

Theorem 2.2.2 (Definable Sets in \mathbf{T}). *A subset S of T is definable in \mathbf{T} by a quantifier-free \mathcal{L} -formula iff S is finite or co-finite. \square*

Using this fact, we can show that the theory of term algebras introduced in Definition 2.1.6 does not admit quantifier elimination except for very special kinds of languages:

Theorem 2.2.3 (Quantifier Elimination in \mathbf{T}). *Let $\mathcal{L} = \{f_1^{(k_1)}, \dots, f_n^{(k_n)}\}$ be an algebraic language. Within the term algebra \mathbf{T} as \mathcal{L} -structure, quantifier elimination is impossible if $\sum_{i=1}^n k_i > 1$. \square*

Consider for example the language $\mathcal{L} = \{a^{(0)}, f^{(1)}, g^{(1)}\}$ and the \mathcal{L} -formula $\exists y(x = f(y))$. The set defined by this formula is obviously both, infinite and co-infinite. Thus according to Theorem 2.2.2, it cannot be equivalent to a quantifier-free \mathcal{L} -formula. For languages with at least one function symbol h with $\alpha(h) > 1$ we arrive at the same conclusion when considering the formula $\exists y(x = h(y, a, \dots, a))$.

In addition to a general quantifier elimination procedure for term algebras over slightly expanded finite languages (see the following definition), [SW02] also contains a straightforward algorithm for quantifier elimination in term algebras for finite languages $\mathcal{L} = \{c_1^{(0)}, \dots, c_n^{(0)}, f^{(1)}\}$. There, the main problem is the lack of

2 Preliminaries

flexibility which stems from the restriction to this special kind of languages. The situation changes when considering quantifier elimination within "term algebras" over slightly expanded languages:

Definition 2.2.4 (Expanded Language). *Let $\mathcal{L} = \{f_1^{(k_1)}, \dots, f_m^{(k_m)}\}$ be an algebraic language. For every n -ary function symbol $f_i \in \mathcal{L}$ and every $i \in \{1, \dots, n\}$, we add a function $inv_{f,i} : T \rightarrow T$, defined as follows:*

$$inv_{f,i}(t) := \begin{cases} a_i & \text{if } t = f(a_1, \dots, a_n), \\ t & \text{else} \end{cases}$$

Expanded languages will be denoted by \mathcal{L}^ .* ◆

The functions $inv_{f,i}$ deliver the i -th argument term for terms beginning with f , and for all other terms return the complete term itself. Due to the non-trivial interpretation of these new function symbols we can no longer speak of a term algebra \mathbf{T} , instead we shall speak of an *expanded term algebra* \mathbf{T}^* (cf. [SW02]).

With these new function symbols we are now able to express certain special properties of terms:

Example 2.2.2 (Expressiveness of Inverse Functions). *We can easily formulate the constraint that an \mathcal{L} -term x has to start with a certain function symbol f by the \mathcal{L}^* -formula $inv_{f,1}(x) \neq x$. Accordingly, the negation of this formula states that x does not start with f . Furthermore, it is possible to express that an \mathcal{L} -term x equals an \mathcal{L} -constant by the \mathcal{L}^* -formula $\bigwedge_i inv_{f_i,1}(x) = x$, for all $f_i \in \mathcal{L}$ with $\alpha(f_i) > 0$. The negation of this formula states that $\Delta(x) > 0$.*

Within \mathbf{T}^* the problem of quantifier elimination mentioned in the comment after Theorem 2.2.3 no longer exists. The formula $\exists y(x = f(y))$ is now equivalent in \mathbf{T}^* to the quantifier-free formula $inv_{f,1}(x) \neq x$, stating that x has to start with the function symbol f . For a function symbol h with $\alpha(h) = n > 1$, we have considered the formula $\exists y(x = h(y, a, \dots, a))$, which is now equivalent in \mathbf{T}^* to $inv_{h,1}(x) \neq x \wedge inv_{h,2}(x) = a \wedge \dots \wedge inv_{h,n}(x) = a$ stating that x has to start with h , and additionally, that the remaining arguments of x have to be equal to the constant a .

As the previous example indicates, the introduction of the inverse function symbols allows to express more complicated properties of terms. We will see in section 4.2, that the non-trivial interpretation of the new function symbols makes simplification of \mathcal{L}^* -formulas more difficult.

We conclude this chapter with an example of a first-order formula within \mathbf{T}^* :

Example 2.2.3 (\mathcal{L}^* -formulas). *Let $\mathcal{L} = \{a^{(0)}, f^{(1)}, g^{(2)}\}$ be an algebraic language. According to Definition 2.2.4, we get*

$$\mathcal{L}^* = \{a^{(0)}, f^{(1)}, inv_{f,1}^{(1)}, g^{(2)}, inv_{g,1}^{(1)}, inv_{g,2}^{(1)}\}$$

2.2 Quantifier Elimination

as the corresponding expanded language \mathcal{L}^* . So far we have to deal with \mathcal{L}^* -formulas like

$$\varphi(u) \equiv \forall y \exists x (f(\text{inv}_{f,1}(u)) \neq y \wedge \text{inv}_{f,1}(\text{inv}_{g,1}(g(x))) = a)(u, x, y)$$

where $\Delta(\varphi(u)) = 3$, $\mathcal{V}(\varphi(u)) = \{x, y, u\}$, $\mathcal{V}_b(\varphi(u)) = \{x, y\}$, $\mathcal{V}_f(\varphi(u)) = \{u\}$. \diamond

3 Elimination Procedure

In this chapter, we focus on expanded term algebras as introduced in the previous section. In the following, we assume to have given a purely functional expanded language \mathcal{L}^* (see Definition 2.2.4), together with its interpretation \mathbf{T}^* , the expanded term algebra introduced in section 2.2. The contents of this chapter closely adhere to our underlying paper [SW02], so we omit explicit references.

In section 3.1, we start with the introduction of a standard form for \mathcal{L}^* -terms and some further definitions necessary for the following paragraphs. In section 3.2, we present the elimination algorithm and related ideas. We conclude this chapter with section 3.3, in which we consider complexity estimations for quantifier elimination in expanded term algebras.

3.1 Preparatory Work

In the following, we introduce a convenient normal form of \mathcal{L}^* -terms for the elimination procedure described in the next section. Furthermore, we categorize the considered expanded languages in terms of their contained symbols.

3.1.1 Normal Forms

Recall that we consider expanded languages

$$\mathcal{L}^* = \{c_1^{(0)}, \dots, c_n^{(0)}, f_1^{(k_1)}, \text{inv}_{f_1,1}^{(1)}, \dots, \text{inv}_{f_1,k_1}^{(1)}, \dots, f_m^{(k_m)}, \text{inv}_{f_m,1}^{(1)}, \dots, \text{inv}_{f_m,k_m}^{(1)}\},$$

containing an arbitrary but finite number of constants c_i , an arbitrary but finite number of function symbols f_i with arities $k_i > 0$, and k_i unary inverse function symbols $\text{inv}_{f_i,\{1,\dots,k_i\}}$ for each f_i of arity k_i .

We call for $n \in \mathbb{N}$ n -fold iterations of inverse function symbols in front of a variable an *inverse term* of \mathcal{L}^* . \mathcal{L}^* -terms are said to be equivalent in \mathbf{T}^* if and only if they describe the same function in \mathbf{T}^* . Based on this equivalence, we introduce the following *normal form* for \mathcal{L}^* -terms: An extended \mathcal{L}^* -term t is in normal form, if it emanates from an extended \mathcal{L} -term $l(x_1, \dots, x_n)$ by substitution of certain inverse terms v_1, \dots, v_n for certain variables x_i . E.g. $f(\text{inv}_{f,1}(\text{inv}_{f,1}(y)))$ is in normal form, whereas $f(\text{inv}_{f,1}(f(x)))$ is not.

The following lemma states that every \mathcal{L}^* -term is algorithmically transformable into an \mathcal{L}^* -term in normal form.

3 Elimination Procedure

Lemma 3.1.1 (Normal form of \mathcal{L}^* -terms). *Every \mathcal{L}^* -term t is \mathbf{T}^* -equivalent to an \mathcal{L}^* -term u in normal form which can be computed algorithmically in polynomial time and linear space including the size of the output. Additionally, we get $\Delta(u) \leq \Delta(t)$. \square*

Applying Lemma 3.1.1 to both sides of equations between \mathcal{L}^* -terms, we can see that every such equation φ is equivalent to true, false or a conjunction of equations $w = u$, where w is either a constant or an inverse term, and u is a term in normal form. To see this, consider the only non-trivial case in which both sides of φ are terms in normal form. If the \mathcal{L} -function symbols of both sides are not identical, then we evaluate φ to false, otherwise both terms start with the same \mathcal{L} -function symbol leading to equations $f(u_1, \dots, u_n) = f(v_1, \dots, v_n)$. We can reduce these to equations $u_i = v_i$ for \mathcal{L}^* -terms u_i, v_i and $1 \leq i \leq n$. The following lemma summarizes these considerations.

Lemma 3.1.2 (Normal form of \mathcal{L}^* -equations). *Every single equation φ between \mathcal{L}^* -terms is \mathbf{T}^* -equivalent to true or false or a conjunction $\tilde{\varphi}$ of equations $w = u$, where w is either constant or an inverse term, and u is a term in normal form. We generally have $\Delta(\tilde{\varphi}) \leq \Delta(\varphi)$. If there are only unary function symbols in \mathcal{L} , then the conjunction degenerates to a single equation, and, consequently, the number of equations does not increase. If there are function symbols of arity greater than one, then the number of equations in the resulting conjunction is bounded by $2^{O(\Delta(\varphi))}$. \square*

Based on Lemma 3.1.2, we can take a step forward and arrive at the *refined normal form* of \mathcal{L}^* -equations:

Lemma 3.1.3 (Refined Normal form of \mathcal{L}^* -equations). *Every single equation φ between \mathcal{L}^* -terms is \mathbf{T}^* -equivalent to true, false or a conjunction $\tilde{\varphi}$ of equations and negated equations. The equations in $\tilde{\varphi}$ are of the form $u = v$, where u is a constant or an inverse term and v is an inverse term. The negated equations are of the form $\neg(\text{inv}_{f,1}(v) = v)$ with inverse terms v . Additionally we get $\Delta(\tilde{\varphi}) \leq \Delta(\varphi)$. The number of base formulas in the conjunction is bounded by $O(\Delta(\varphi))$ if \mathcal{L} contains only unary function symbols. Otherwise, the number of base formulas in the resulting conjunction is bounded by $2^{O(\Delta(\varphi))}$.*

Proof. With Lemma 3.1.2, we can restrict our attention to equations of the form $w = u$, where w is either a constant or an inverse term, and u is a term in normal form. Let w be a constant, and let u be not an inverse term. Then we can decide $w = u$ by inspection to be either *true* or *false*. If, in contrast, u is an inverse term, then $w = u$ is already in the desired form and we are done.

In the case that w is an inverse term, we transform $w = u$ into a conjunction of equations and negated equations of the desired form. This is done by induction

3.1 Preparatory Work

on $|u|$. If $|u| = 1$, then u is either an \mathcal{L} -constant or a variable. In the former case, $u = w$ is of the desired form. In the latter case, we are already done. Let now $|u| > 1$. If u is an inverse term, then we are done. Else u is a term in normal form of the form $f(t_1, \dots, t_n)$, where f is an n -ary \mathcal{L} -function symbol, and t_1, \dots, t_n are \mathcal{L}^* -terms. Then the equation $w = f(t_1, \dots, t_n)$, which we are considering, is equivalent to

$$\neg(\text{inv}_{f,1}(w) = w) \wedge \bigwedge_{i=1}^n \text{inv}_{f,i}(w) = t_i.$$

We can now apply the induction hypothesis to each of the equations. \square

The proof of this lemma contains an instruction for computing the refined normal form of \mathcal{L}^* -equations. Since the normal form of \mathcal{L}^* -equations plays an important role in the elimination procedure described in section 3.2, we summarize the instruction in the form of an algorithm:

Algorithm 3.1.1 (Refined Normal Form).

Input $\varphi \in \mathcal{E}_{\mathcal{L}^*}$, in normal form according to Lemma 3.1.2
Output \mathcal{L}^* -formula φ' equivalent to φ , where each equation of φ' is in refined normal form according to Lemma 3.1.3

```

1  procedure RNF( $\varphi$ )
2  begin
3    if "both hand sides of  $\varphi$  are either inverse terms or constants" then
4      return  $\varphi$ 
5    end
6     $t$  := "term in normal form of  $\varphi$  with arity  $k$  and argument
           terms  $a_i$  for  $1 \leq i \leq k$ "
7     $f$  := "function symbol  $t$  starts with"
8     $t'$  := "inverse term of  $\varphi$ "
9     $\varphi'$  :=  $\neg(\text{inv}_{f,1}(t') = t') \wedge \bigwedge_{i=1}^k \text{RNF}(\text{inv}_{f,i}(t') = a_i)$ 
10   return  $\varphi'$ 
11 end RNF

```

End of Algorithm 3.1.1

Theorem 3.1.1 (Refined Normal Form). *Let φ be an \mathcal{L}^* -equation in normal form according to Lemma 3.1.2. Algorithm 3.1.1 applied to φ terminates, and transforms φ into an equivalent \mathcal{L}^* -formula φ' solely containing inverse terms and \mathcal{L} -constants.*

3 Elimination Procedure

Proof. Consider the term t in normal form of φ as a tree. We prove the termination and the correctness of Algorithm 3.1.1 by induction on the maximum length l of a series of successive \mathcal{L} -function symbols within t .

Termination: For $l = 0$, the algorithm terminates according to lines 3 to 5. Let $l > 0$. In this case, for each of the $k \geq 1$ argument terms a_i in normal form of t , we get one recursive call of the procedure. Due to the fact that the maximum length of a series of \mathcal{L} -function symbols for each of the terms a_i is at most $l - 1$, the algorithm terminates for all $l \in \mathbb{N}$.

Correctness: For $l = 0$, the algorithm returns the input formula itself, which in this case solely consists of inverse terms or constants. Let $l > 0$. Let f be the outermost \mathcal{L} -function symbol of the term in normal form of φ . Then φ is of the form $f(a_1, \dots, a_k) = t'$, where the a_i are terms in normal form, and t' is an inverse term. In line 9, we "eliminate" the function symbol f by the following statements: the negated equation of φ' states that t' has to start with the function symbol f . The following k equations $inv_{k,i}(t') = a_i$, for $1 \leq i \leq k$, state the equality of the k argument terms $inv_{f,i}(t')$ of t' with the arguments a_i of t . The argument terms a_i start with at most $l - 1$ \mathcal{L} -function symbols and the terms $inv_{f,i}(t')$ are still inverse terms. So for the equations $inv_{k,i}(t') = a_i$ we can compute the refined normal form by the induction hypothesis. \square

In order to transform an arbitrary \mathcal{L}^* -formula into refined normal form, one has to recursively apply Algorithm 3.1.1 to each equation of the input formula. Remember that after transforming an arbitrary \mathcal{L}^* -formula φ into an equivalent formula φ' in refined normal form, φ' solely consists of constants and inverse terms. In section 3.2, we will see how this condition helps us to perform quantifier elimination in expanded term algebras.

After transforming equations in the way described in Algorithm 3.1.1, we can restrict our attention to equations of one of the following forms:

- (i) Negated equations $inv_{f,1}(v) \neq v$, stating that the subtree described by the inverse term v starts with f .
- (ii) Equations $w = v$ for a constant or a variable w and an inverse term v , stating that w occurs at the position described by v .
- (iii) Equations $w = v$ for inverse terms w and v , stating the equality of the subtrees described by w and v .

If we consider the elements of T as labelled trees, then an inverse term v assigns to each $t \in T$ a certain subtree of t specified by v . In order to become familiar with the idea of inverse terms and the content of Lemma 3.1.3, we take a look at an example. It illustrates the expressiveness of inverse functions introduced in Definition 2.2.4:

Example 3.1.1 (Refined Normal Form of \mathcal{L}^* -equations). Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, \text{inv}_{f,1}^{(1)}, g^{(2)}, \text{inv}_{g,1}^{(1)}, \text{inv}_{g,2}^{(1)}\}$ be an expanded language and let \mathbf{T}^* be its interpretation. Let $\varphi \equiv g(a, x) = y$ be an \mathcal{L}^* -formula with variables x, y . Applying Algorithm 3.1.1 to φ leads to the \mathbf{T}^* -equivalent formula

$$\varphi' \equiv \underbrace{\text{inv}_{g,1}(y) \neq y}_{(*)} \wedge \underbrace{\text{inv}_{g,1}(y) = a}_{(**)} \wedge \underbrace{\text{inv}_{g,2}(y) = x}_{(***)}.$$

φ' encodes the following: if φ holds, then y has to start with g (*). Furthermore, the first argument of y has to be equal to a (**), and the second argument of y has to be equal to x (***). \diamond

Note that using Algorithm 3.1.1, we can transform every input formula φ into a \mathbf{T}^* -equivalent formula φ' solely containing constants and inverse terms. Thus the maximal depth $\Delta(t)$ of all terms t of the transformed formula is equal to the maximal nesting of inverse function symbols.

3.1.2 Language Types

As we will see in section 3.3, the complexity of the quantifier elimination problem in \mathbf{T}^* highly depends on the considered language \mathcal{L} . Therefore, we distinguish three different types of finite languages. These language types (denoted by \mathbf{x} - \mathbf{y}) vary in the number of function symbols they contain (specified by \mathbf{x}) and in the arities that these function symbols are allowed to have (specified by \mathbf{y}):

- 1-1** \mathcal{L} is of this type, if it contains a finite nonzero number of constants, exactly one unary function symbol and no function symbols of arity greater than one.
- N-1** \mathcal{L} is of this type, if it contains a finite nonzero number of constants, at least two, but finitely many unary function symbols and no function symbol of arity greater than one.
- N-N** \mathcal{L} is of this type, if it contains a finite nonzero number of constants and an arbitrary finite number of function symbols, including at least one function symbol of arity greater than one.

Depending on the type of the considered language, the number of different terms of a certain maximum depth varies enormously:

Lemma 3.1.4 (Term Set Cardinalities). (i) Let \mathcal{L} be of type 1-1, let c be an \mathcal{L} -constant and let f be a unary \mathcal{L} -function symbol. For $n \in \mathbb{N}$ we denote

3 Elimination Procedure

by $f^{(n)}$ an n -fold iteration of the function f . For each $n \in \mathbb{N}$, we define the following set of \mathcal{L} -terms:

$$G_n = \{c, f(c), f(f(c)), \dots, f^{(n)}(c)\} \text{ for } n \in \mathbb{N}.$$

For all $n \in \mathbb{N}$, we have $\#G_n = n + 1$ and $\Delta(G_n) = n$.

(ii) Let \mathcal{L} be of type $N-1$, let c be an \mathcal{L} -constant, and let f and g be unary \mathcal{L} -function symbols. We recursively define the following sequence of sets of \mathcal{L} -terms:

$$G_0 = \{c\}, \quad G_n = \{f(s) \mid s \in G_{n-1}\} \cup \{g(s) \mid s \in G_{n-1}\} \text{ for } 0 < n \in \mathbb{N}.$$

Then for all $n \in \mathbb{N}$, we have $\#G_n = 2^n$ and $\Delta(G_n) = n$.

(iii) Let \mathcal{L} be of type $N-N$, let c be an \mathcal{L} -constant and let g be an \mathcal{L} -function symbols of arity > 1 . We recursively define the following sequence of sets of \mathcal{L} -terms:

$$G_0 = \{c\}, \quad G_n = G_{n-1} \cup \{g(s, s', t, \dots, t) \mid s, s' \in G_{n-1}\} \text{ for } 0 < n \in \mathbb{N}.$$

Then for all $n \in \mathbb{N}$, we have $\#G_n \geq 2^n$ and $\Delta(G_n) = n$. □

Next we introduce the approach for quantifier elimination in \mathbf{T}^* . We will see that in this context, the normal form for \mathcal{L}^* -equations introduced in section 3.1.1 plays an important role.

3.2 Elimination

The initial situation is as follows: We consider finite expanded languages \mathcal{L}^* with at least one constant symbol and at least one function symbol of positive arity. Within our \mathcal{L}^* -structure \mathbf{T}^* , we want to eliminate the quantifier from

$$\Theta(y_1, \dots, y_n) \equiv \exists x(\varphi)(x, y_1, \dots, y_n)$$

where φ is a quantifier-free formula possibly containing other variables y_1, \dots, y_n , besides the quantified variable x . The general elimination technique we want to use here is briefly described in the next section. Originally, it was introduced for the reals. It works by substitution of *test terms*.

3.2.1 Quantifier Elimination by Substitution of Parametric Test Terms

The method of quantifier elimination we want to use here dates back to a paper by Weispfenning in 1988 (see [Wei88]). In the following, we describe the basic idea of this method for eliminating the quantifier $\exists x$ in front of a quantifier-free formula φ over the language of rings within the reals.

Consider the formula $\exists x(\varphi)(x, u_1, \dots, u_n)$. Variables u_1, \dots, u_n occurring freely in φ are called *parameters*. Intuitively, the existential quantifier can be considered as a infinite disjunction $\bigvee_{r \in \mathbb{R}} \varphi(x, u_1, \dots, u_n)[r/x]$ over all real numbers. The basic idea of quantifier elimination by substitution is to restrict the infinite set of elements which one has to substitute for the bound variable x to a finite set E of eligible parametric terms t , called *test terms* (cf. [Dol00]), so that

$$\mathbf{R} \models \exists x(\varphi)(x, u_1, \dots, u_n) \leftrightarrow \bigvee_{t \in E} \varphi(x, u_1, \dots, u_n)[t/x].$$

A set E for which the equivalence holds in the structure of real numbers is then called an *elimination set* for $\exists x(\varphi)(x, u_1, \dots, u_n)$. An important goal of this elimination technique is to restrict E as much as possible.

One benefit of this elimination technique is that the number of parameters of the input formula plays a minor role for the complexity of the procedure. Another benefit, and for our purposes an even more important one, is the option to get sample points, additionally to an equivalent parametric quantifier-free formula. This possibility stems from the above mentioned elimination procedure together with the elimination sets: we just have to store the single results of each substitution together with the substituted test terms in a suitable data structure. Let $E = \{t_1, \dots, t_m\}$. Then this leads to a scheme

$$\left[\begin{array}{c|c} \varphi(x, u_1, \dots, u_n)[t_1/x] & x = t_1 \\ \vdots & \vdots \\ \varphi(x, u_1, \dots, u_n)[t_m/x] & x = t_m \end{array} \right].$$

For fixed values of the parameters u_1, \dots, u_n , we have $\mathbf{R} \models \exists x(\varphi)$ iff $\mathbf{R} \models \varphi[t_i/x]$ for at least one $i \in \{1, \dots, m\}$. In the positive case, the corresponding $x = t_i$ is one possible choice for x .

In order to eliminate an arbitrary number of existential quantifiers in front of a quantifier-free formula, we successively eliminate the quantifiers one by one starting with the innermost one. Using double negation $\forall x(\varphi) \leftrightarrow \neg \exists x(\neg \varphi)$, we can embark on the same strategy to eliminate universal quantifiers.

On the basis of the normal form for \mathcal{L}^* -equations introduced in section 3.1.1, we will try to use this elimination technique to get rid of the quantifier in the input formula $\Theta(y_1, \dots, y_n) \equiv \exists x(\varphi)(x, y_1, \dots, y_n)$. One crucial point on the way

3 Elimination Procedure

towards an elimination set for Θ , is the transformation described in Lemma 3.1.3. Recall that after transforming Θ into refined normal form, we only have to cope with constants or inverse terms. This fact will help us to get a finite elimination set for Θ .

3.2.2 Depth Bounds

Due to the transformation described in Lemma 3.1.3, from now on, we only have to deal with \mathcal{L}^* -equations and negated \mathcal{L}^* -equations of one of the following forms:

- (i) $v(x) = c$, where v is an inverse term and c an \mathcal{L} -constant,
- (ii) $v(x) = v'(x)$, where v, v' are inverse terms,
- (iii) $v(x) = v'(y_i)$, where v, v' are inverse terms,
- (iv) $v(y_i) = c$, where v is an inverse term and c an \mathcal{L} -constant,
- (v) $v(y_i) = v'(y_j)$, where v, v' are inverse terms.

(i) states that the unknown term x has an \mathcal{L} -constant c as a subterm at the position specified by the inverse term v . In the same way, (ii) and (iii) state that x and y_i have equal subterms at the positions specified by v and v' , respectively. (iv) and (v) are of minor interest for the elimination of the quantifier in our input formula $\Theta(y_1, \dots, y_n) \equiv \exists x(\varphi)(x, y_1, \dots, y_n)$, because these do not involve the quantified variable x .

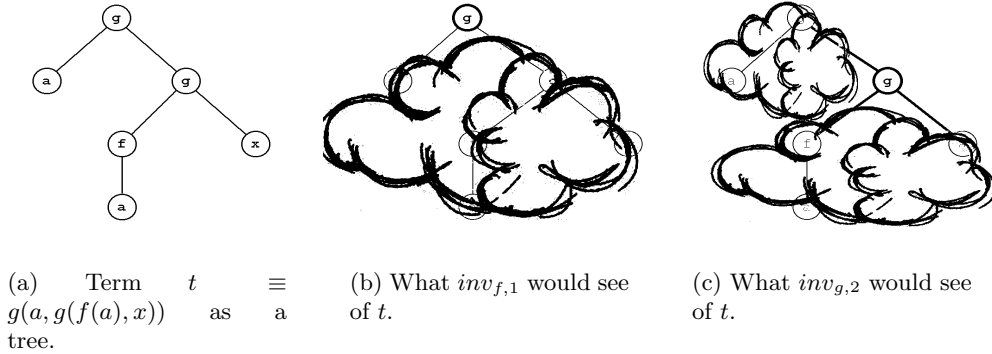


Figure 3.1: Intuitive meaning of the depth of an inverse term.

The basic idea to get finite elimination sets for getting rid of the quantifier in Θ is that equations and negated equations between inverse terms prescribe identities and non-identities, respectively, between the subterms that these terms access

in their arguments. Intuitively a single inverse term cannot look deeper into its argument than its own depth indicates (see Figure 3.1).

The strategy we embark on to get finite elimination sets, is to estimate an upper bound \mathcal{B} for the depth of the candidate terms which we have to substitute for the bound variable. The crucial thing about the set of candidate terms with depth less or equal to \mathcal{B} is that it is finite. For the estimation of this maximum depth, we have to take into account two properties of the input formula: the first one is of course the maximum depth $\Delta(\Theta)$ among all terms of the input formula. As long as the input formula exclusively contains equations, this bound is exactly the maximum depth among all inverse terms of the transformed input formula. With the occurrence of inequalities, we have to be more generous (see Definition 3.2.1).

The second property is the "maximum transitivity" that the bound variable x is involved in. Here we consider the connection of the bound variable with other variables. Therefore, we additionally take into account the number of equations of the input formula.

With these two estimations, we get an estimation of the maximum depth the \mathcal{L}^* -terms in the elimination set must have to be sure that if none of these terms satisfies φ then there exists none (see Theorem 3.2.1 in section 3.2.4). In other words, if there exists a term t satisfying φ then there also exists a term t' satisfying φ having $\Delta(t') \leq \mathcal{B}$. Lemma 3.2.1 states how we can replace occurrences of certain subterms in a term t by arbitrary terms without changing the validities of the equations in which t occurs. For a term satisfying the input formula, Theorem 3.2.1 simulates the truth values of all equations in the input formula. Therefore the boolean structure of φ can be neglected and thus from now on we can concentrate on the set Φ of equations in φ .

The approach for computing a finite elimination set R_Φ for the input formula $\theta(y_1, \dots, y_n) \equiv \exists x(\varphi)(x, y_1, \dots, y_n)$ is now as follows:

1. Transform Θ into refined normal form $\tilde{\Theta}$ according to Lemma 3.1.3.
2. Consider the set Φ of equations in $\tilde{\Theta}$ and estimate a depth bound \mathcal{B} for the candidate terms w.r.t. Φ .
3. Compute the set R_Φ of all terms t in normal form with $\Delta(t) \leq \mathcal{B}$.

In the following the terms of the elimination set R_Φ are called Φ -restricted terms. The exact definition of such a term depends on the given language \mathcal{L} . In section 3.1.2 we introduced three different types of finite languages. For each of these types we define the set R_Φ of Φ -restricted terms separately as follows:

Definition 3.2.1 (Elimination Sets). *The depth bounds and accordingly, the sets of Φ -restricted terms in normal form are defined with respect to the language type:*

3 Elimination Procedure

- For languages 1-1, we define $\mathcal{B} = 2\Delta(\Phi) + \#\Phi$.
- For languages N-1, we define $\mathcal{B} = 2\Delta(\Phi) + \log(\#\Phi + 1)$.
- For languages N-N, we define $\mathcal{B} = \Delta(\Phi) \cdot (\#\Phi + 1) + \log(\#\Phi + 1)$.

For each language type, we define the set of Φ -restricted terms separately using the type-specific depth bounds defined above:

$$R_\Phi = \{ \mathcal{L}^* \text{-term } t(y_1, \dots, y_n) \mid \Delta(t) \leq \mathcal{B} \}. \quad \blacklozenge$$

One important fact about the sets R_Φ is that they are finite. With the information on the maximal depth \mathcal{B} , we can make further statements about the elements of R_Φ . Let A be the maximal arity of all function symbols within the given language \mathcal{L} . With the bounds \mathcal{B} defined in Definition 3.2.1, we get bounds for the maximal length $|R_\Phi|$ of terms in R_Φ :

For language type 1-1:

$$|R_\Phi| \leq 2\Delta(\Phi) + \#\Phi = O(\Delta(\Phi) + \#\Phi)$$

For language type N-1:

$$|R_\Phi| \leq 2\Delta(\Phi) + \log(\#\Phi + 1) = O(\Delta(\Phi) + \log(\#\Phi))$$

In these two cases the maximal length of terms in R_Φ is equal to the depth bounds defined above, because in both cases the maximal arity of function symbols within \mathcal{L}^* is one. In the same way, we get the length bound for the remaining language type:

For language type N-N:

$$|R_\Phi| \leq A^{\Delta(\Phi) \cdot (\#\Phi + 1) + \log(\#\Phi + 1) + 1} = 2^{O(\Delta(\Phi) \cdot \#\Phi)}$$

Let m be the number of free variables occurring in φ . We can obtain estimations on the number of terms in R_Φ for each language type. In the case of type 1-1 languages, we have terms in normal form $t = f^{\Delta t - k} \text{inv}_{f,1}^k(z)$, where $k \in \{0, \dots, \Delta(t)\}$, f is the unique unary \mathcal{L} -function symbol and z is either one of the \mathcal{L} -constants or one of the variables y_1, \dots, y_m . So in this case we get the following bound:

For language type 1-1:

$$\#R_\Phi \leq (m + \#\mathcal{L}^*) \cdot (2\Delta(\Phi) + \#\Phi)^2 = O(m \cdot (\Delta(\Phi) + \#\Phi)^2)$$

For the remaining language types, the idea to approximate $\#R_\Phi$ is to estimate on the one hand, the number of nodes in the tree associated with a Φ -restricted term and on the other hand, the number of symbols that may occur at each node:

For language type N-1:

$$\#R_{\Phi} \leq (2\#\mathcal{L}^* + m)^{2\Delta(\Phi) + \log(\#\Phi + 1)} = 2^{O(\log(m) \cdot (\Delta(\Phi) + \log(\#\Phi)))}$$

For language type N-N:

$$\#R_{\Phi} \leq ((A + 1)\#\mathcal{L}^* + m)^{A\Delta(\Phi) \cdot (\#\Phi + 1) + \log(\#\Phi + 1) + 1} = 2^{\log(m) \cdot 2^{O(\Delta(\Phi) \cdot \#\Phi)}}$$

In the next section, we specify the way how to replace occurrences of certain subterms in a term by arbitrary terms without changing the validity of the equations in Φ .

3.2.3 Substitutions

In section 3.2.2 we have already mentioned the possibility to consider arbitrary \mathcal{L}^* -terms in a natural way as labelled trees. Based on this view of terms, we can identify *positions* of symbols in an arbitrary term t by describing the path from the root of the tree to the corresponding symbol by a finite sequence $\omega \in \mathbb{N}^*$:

- (i) The position of the root of the tree representing an \mathcal{L}^* -term is the empty sequence $()$.
- (ii) Let α be a symbol in t occurring at depth > 0 . Then α is the first symbol of a proper subterm of t and there is an n -ary function symbol f such that α is the i -th argument term of f for some $i \in \{1, \dots, n\}$. Let ω be the position of this f in t . Then the position of α is defined as the concatenation $\omega \circ i$.

For an \mathcal{L}^* -term t , $\Omega(t) \subset \mathbb{N}^*$ denotes the finite set of all positions in t . For an arbitrary position $\omega \in \Omega(t)$, $t\omega \in T$ denotes the subterm of t starting at position ω . $|\omega|$ denotes the length of the sequence $\omega \in \Omega(t)$. Obviously $\Delta(t\omega) = \Delta(t) - |\omega|$. Figure 3.2 gives an example of use for the just defined notions.

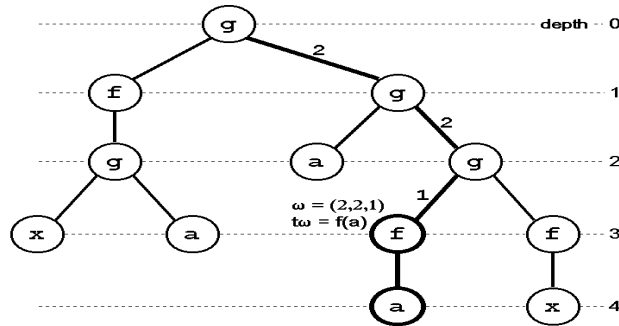


Figure 3.2: Positions and Subterms

3 Elimination Procedure

For two positions ω, ω' , we write $\omega \preceq \omega'$ if ω is an initial segment of ω' . If $\omega \preceq \omega'$ or $\omega' \preceq \omega$ then ω and ω' are said to be *comparable* otherwise *incomparable*.

With these definitions, we are now able to define the meaning of inverse terms more precisely. For an arbitrary term t , every inverse term v uniquely induces a position $pos(v, t) \in \Omega(t)$ as follows:

- (i) If v is a variable, then $v^{\mathbf{T}^*}(t) = t$, and we define $pos(v, t) = ()$.
- (ii) Let v be of the form $inv_{f,i}(v')$, where f is an n -ary function symbol, $i \in \{1, \dots, n\}$, v' is an inverse term and $pos(v', t) = \omega$. If $v'^{\mathbf{T}^*}(t)$ is of the form $f(t_1, \dots, t_n)$ then $pos(v, t) = \omega \circ i$ else $pos(v, t) = \omega$.

Note that $tpos(v, t) = v^{\mathbf{T}^*}(t)$, but of course t could contain further copies of $v^{\mathbf{T}^*}(t)$ at other positions.

In the following, we define how to transform a term $t \in T$ into an \mathcal{L}^* -term by replacing certain occurrences of some subterm of t by some \mathcal{L}^* -term: Let $t \in T$, let t' be a subterm of t , let s be an \mathcal{L}^* -term and $M \subset \mathbb{N}^*$. We obtain the \mathcal{L}^* -term $t\langle s, t', M \rangle$ by considering all $\omega \in M$ for which $t\omega = t'$ and replacing at each such position ω the subterm t' by s . The following is a formal recursive definition of this approach:

- (i) Let t be an \mathcal{L} -constant. If $t = t'$ and $() \in M$ then $t\langle s, t', M \rangle = s$ else $t\langle s, t', M \rangle = t$.
- (ii) Let t be of the form $f(t_1, \dots, t_n)$, where f is an n -ary function symbol and $t_1, \dots, t_n \in T$. If $t = t'$ then we arrive at case (i), else $t\langle s, t', M \rangle = f(t_1\langle s, t', M_1 \rangle, \dots, t_n\langle s, t', M_n \rangle)$, where $M_i = \{\omega \in \mathbb{N}^* \mid i \circ \omega \in M\}$ for $i \in \{1, \dots, n\}$.

Now let x be a variable, let Φ be a finite set of equations of the forms (i) to (iii) as introduced in the beginning of section 3.2.2. Let $t, d_1, \dots, d_m \in T$ such that $\mathbf{T}^* \models \Phi(t, d_1, \dots, d_m)$. The following Lemma specifies how to replace an occurrence ω of a subterm in t by an arbitrary \mathcal{L}^* -term e without compromising the validity of the equations in Φ . It is the basis for the elimination theorem of the following section.

Lemma 3.2.1 (Safe Substitutions). *Let x, y_1, \dots, y_m be variables. Let Φ_x be a set of equations of the form $v(x) = v'(x)$, where v, v' are inverse terms. Let Φ_y be a set of equations of the form $u(x) = u'(y_i)$, where u is an inverse term, u' is an inverse term or an \mathcal{L} -constant and $i \in \{1, \dots, m\}$. Let $t, d_1, \dots, d_m \in T$ such that*

$$\mathbf{T}^* \models v(t) = v'(t) \text{ for all } v(x) = v'(x) \in \Phi_x,$$

$$\mathbf{T}^* \models u(t) = u'(d_i) \text{ for all } u(x) = u'(y_i) \in \Phi_y.$$

Then for any position $\mu \in \Omega(t)$ with $|\mu| > \Delta_x(\Phi_x \cup \Phi_y) \cdot (\#\Phi_x + 1)$ there are sets $P_\mu, D_\mu \subseteq \Omega(t)$ with the following properties:

1. For all $\omega, \omega' \in P_\mu$ we have $t\omega = t\omega'$.
2. For all $\omega \in D_\mu$ we have that $t\omega$ is a subterm of one of d_1, \dots, d_m .
3. Any two positions $\omega, \omega' \in P_\mu \cup D_\mu$ are incomparable.
4. $P_\mu \cap D_\mu = \emptyset$.
5. Suppose that $\mu' \notin D_\mu$ for all $\mu' \preceq \mu$. Let $e \in T$. If there is $\mu' \in P_\mu$ for some $\mu' \preceq \mu$, then this μ' is unique by property 3 and we set $t' = t(t\mu' \langle e, t\mu, \{t\mu\} \rangle, t\mu', P_\mu)$. If there is no such μ' , then we set $t' = t \langle e, t\mu, \{t\mu\} \rangle$. In either case it follows that the obtained t' can take the role of the original t in all our equations:

$$\mathbf{T}^* \models v(t') = v'(t') \text{ for all } v(x) = v'(x) \in \Phi_x,$$

$$\mathbf{T}^* \models u(t') = u'(d_i) \text{ for all } u(x) = u'(y_i) \in \Phi_y.$$

6. For all $\omega \in P_\mu \cup D_\mu$ we have $|\omega| \leq \Delta_x(\Phi_y) \cdot (\#\Phi_x + 1)$. □

On the basis of this Lemma we give the elimination theorem in the next section. It is fundamental for efficient quantifier elimination in extended term algebras.

3.2.4 Algorithm

Recall that we would like to solve the following problem: Eliminate the quantifier from the given input formula $\Theta(y_1, \dots, y_n) \equiv \exists x(\varphi)(x, y_1, \dots, y_n)$, where φ is a quantifier-free formula containing besides x possibly other free variables y_1, \dots, y_n , by the elimination technique briefly described in section 3.2.1. Due to the transformation stated in Lemma 3.1.3, we only have to cope with very special kinds of equations (see section 3.2.2).

The following elimination theorem is the clue for efficient quantifier elimination in \mathbf{T}^* . It simulates the truth values for an element x satisfying φ of all equations in φ . Therefore, we neglected the boolean structure of φ and instead considered the set Φ of equations occurring in φ .

Theorem 3.2.1 (Quantifier Elimination Theorem). *Let Φ be a nonempty set of \mathcal{L}^* -equations, let x, y_1, \dots, y_m be the variables occurring in the equations in Φ and let $b, d_1, \dots, d_m \in T$. Then there exists a Φ -restricted term $t(y_1, \dots, y_m)$ such that for all equations $\psi(x, y_1, \dots, y_m) \in \Phi$ we have*

$$\mathbf{T}^* \models \psi(b, d_1, \dots, d_m) \text{ iff } \mathbf{T}^* \models \psi(t^{\mathbf{T}^*}(d_1, \dots, d_m), d_1, \dots, d_m). \quad \square$$

3 Elimination Procedure

This theorem together with the elimination sets R_Φ enable us to apply the elimination technique briefly described in section 3.2.1 to input formulas $\exists x(\varphi)$.

Corollary 3.2.1 (Quantifier Elimination in \mathbf{T}^*). *Let φ be a quantifier-free \mathcal{L}^* -formula with variables x, y_1, \dots, y_m . Let $\tilde{\varphi}$ be an equivalent and-or-combination of equations and negated equations in refined normal form according to Lemma 3.1.3. Let Φ be the set of all equations of $\tilde{\varphi}$ and denote by R_Φ the set of Φ -restricted terms for the considered type of language. Then*

$$\mathbf{T}^* \models \exists x(\varphi) \iff \bigvee_{t \in R_\Phi} \varphi[t/x]. \quad \square$$

As mentioned in section 3.2.1 using this method also enables us to eliminate universal quantifiers and blocks of arbitrary quantifiers when considering equivalent prenex formulas.

Corollary 3.2.2 (Extended Quantifier Elimination in \mathbf{T}^*). *Let φ be a quantifier-free \mathcal{L}^* -formula with variables x, y_1, \dots, y_m . Let $\tilde{\varphi}$ be an equivalent and-or-combination of equations and negated equations in refined normal form according to Lemma 3.1.3. Let Φ be the set of all equations of $\tilde{\varphi}$ and denote by R_Φ the set of Φ -restricted terms for the considered type of language. Let $R_\Phi = \{t_1, \dots, t_n\}$ and consider the scheme*

$$\left[\begin{array}{c|c} \varphi[t_1/x] & x = t_1 \\ \vdots & \vdots \\ \varphi[t_n/x] & x = t_n \end{array} \right].$$

Then for fixed values of the parameters y_1, \dots, y_m we have that $\mathbf{T}^ \models \exists x\varphi$ iff $\mathbf{T}^* \models \varphi[t_i/x]$ for at least one $i \in \{1, \dots, n\}$. In the positive case, the corresponding $x = t_i$ is one possible choice for x . \square*

By collecting the sample solutions for each quantifier, one gets sample solutions for an entire block of existential quantifiers.

In the next section we concentrate on the time and space complexity of the elimination procedure when applied iteratively to an arbitrary prenex formula.

3.3 Complexity

We consider input formulas $\exists x(\varphi)(x, y_1, \dots, y_m)$, where φ is a quantifier-free formula. As introduced in the previous chapter, $\Delta(\varphi)$ denotes the maximal depth among all terms in φ .

In order to eliminate the quantifier, we will first compute a refined normal form $\tilde{\varphi}$ of φ according to Lemma 3.1.3. As discussed in the previous section, we will switch from $\tilde{\varphi}$ to the set Φ of equations in $\tilde{\varphi}$. From Φ , we compute the set R_Φ of

3.3 Complexity

Φ -restricted terms with which we can eliminate the quantifier of the input formula as described in Corollary 3.2.1.

On the basis of the complexity bounds on R_Φ of section 3.2.2, we can bound the complexity of the entire elimination procedure in terms of the original input formula as follows: For the depth of Φ we get

$$\Delta(\Phi) = \Delta(\tilde{\varphi}) \leq \Delta(\varphi).$$

The bound on $\#\Phi$ depends on the considered language. For languages of type 1-1 and N-1 we get

$$\#\Phi \leq eq(\tilde{\varphi}) \leq \Delta(\varphi) \cdot eq(\varphi).$$

In the case of type N-N we get the weaker bound

$$\#\Phi \leq eq(\tilde{\varphi}) \leq 2^{O(\Delta(\varphi))} \cdot eq(\varphi).$$

As stated in Corollary 3.2.1, our input formula $\exists x\varphi$ is \mathbf{T}^* -equivalent to the quantifier-free disjunction $\varphi' \equiv \bigvee_{t \in R_\Phi} \varphi[t/x]$. With respect to the maximal depth, the contained terms, the number of disjunctions and the maximal word length of a single disjunction of φ' , we are going to compute some characteristic quantities for φ' for each language type:

Type 1-1:

$$\begin{aligned} \Delta(\varphi') &\leq \Delta(\varphi) + \Delta(R_\Phi) = \Delta(\varphi) + 2\Delta(\Phi) + \#\Phi \leq \\ &\Delta(\varphi) \cdot (3 + eq(\varphi)) = O(\Delta(\varphi) \cdot eq(\varphi)) \\ \#R_\Phi &= O(m \cdot (\Delta(\Phi) + \#\Phi)^2) = O(m \cdot \Delta(\varphi)^2 \cdot eq(\varphi)^2) \\ |\varphi[t/x]| &= |\varphi| \cdot |R_\Phi| = |\varphi| \cdot O(\Delta(\Phi) + \#\Phi) = O(|\varphi| \cdot \Delta(\varphi) \cdot eq(\varphi)) \end{aligned}$$

Type N-1:

$$\begin{aligned} \Delta(\varphi') &\leq \Delta(\varphi) + \Delta(R_\Phi) = \Delta(\varphi) + 2\Delta(\Phi) + \log(\#\Phi) \leq \\ &4\Delta(\varphi) + \log(eq(\varphi)) = O(\Delta(\varphi) + \log(eq(\varphi))) \\ \#R_\Phi &= 2^{O(\log(m) \cdot (\Delta(\Phi) + \log(\#\Phi)))} = 2^{O(\log(m) \cdot (\Delta(\varphi) + \log(eq(\varphi))))} \\ |\varphi[t/x]| &= |\varphi| \cdot |R_\Phi| = |\varphi| \cdot O(\Delta(\Phi) + \log(\#\Phi)) = \\ &O(|\varphi| \cdot (\Delta(\varphi) + \log(eq(\varphi)))) \end{aligned}$$

Type N-N:

$$\Delta(\varphi') \leq \Delta(\varphi) + \Delta(R_\Phi) = \Delta(\varphi) + \Delta(\Phi) \cdot (\#\Phi + 1) + \log(\#\Phi) =$$

3 Elimination Procedure

$$\begin{aligned}
& 2^{O(\Delta(\varphi))} \cdot eq(\varphi) \\
\#R_{\Phi} &= 2^{\log(m) \cdot 2^{O(\Delta(\Phi) \cdot \#\Phi)}} = 2^{\log(m) \cdot 2^{O(\Delta(\varphi)) \cdot eq(\varphi)}} \\
|\varphi[t/x]| &= |\varphi| \cdot |R_{\Phi}| = |\varphi| \cdot 2^{O(\Delta(\Phi) \cdot \#\Phi)} = |\varphi| \cdot 2^{O(\Delta(\varphi)) \cdot eq(\varphi)}
\end{aligned}$$

On this basis, we first obtain the following results for the elimination of a single quantifier block $\exists x_1 \dots \exists x_n$ in front of a quantifier-free formula. Finally, we get the results for the elimination of several quantifier blocks. For $n \in \mathbb{N}$, we denote by $exp^n(x)$ an n -fold iteration of the exponential function.

Theorem 3.3.1 (Elimination of one block). *Let φ' be the result of eliminating the prenex block of k like quantifiers from $\exists x_1 \dots \exists x_k \varphi$.*

(i) *For languages of type 1-1, we have*

$$|\varphi'| = |\varphi|^{O(k^2)}.$$

This is singly exponential in the input word length. More precisely, it is singly exponential in the number of quantifiers and polynomial in $|\varphi|$.

(ii) *For languages of type N-1, we have*

$$|\varphi'| = |\varphi| \cdot 2^{2^{O(k)} \cdot \log(m) \cdot (\Delta(\varphi) + \log(eq(\varphi)))}.$$

This is doubly exponential in the input word length. More precisely, it is doubly exponential in the number of quantifiers, singly exponential in $|\varphi|$ and polynomial in $eq(\varphi)$.

(iii) *For languages of type N-N, we have*

$$|\varphi'| = |\varphi| \cdot 2^{\log(m) \cdot exp^k(2^{O(\Delta(\varphi)) \cdot eq(\varphi)})} = exp^{k+2}(O(|\varphi|)).$$

For input length l , this is l -fold exponential in l . More precisely, it is $k+2$ -fold exponential in $\Delta(\varphi)$, $k+1$ -fold exponential in $eq(\varphi)$ and polynomial in all other possible complexity parameters. \square

Theorem 3.3.2 (Elimination of several blocks). *Let φ' be the result of eliminating the a prenex blocks of at most k like quantifiers each from*

$$\exists x_{11} \dots \exists x_{1k} \forall x_{21} \dots \forall x_{2k} \dots \exists x_{a1} \dots \exists x_{ak} \varphi.$$

3.3 Complexity

(i) For languages of type 1-1, we have

$$|\varphi'| = |\varphi|^{k^{O(a)}}.$$

This is doubly exponential in the input word length. More precisely, it is doubly exponential in the number of quantifier blocks, singly exponential in the number of quantifiers and polynomial in $|\varphi|$.

(ii) For languages of type N-1, we have

$$|\varphi'| \leq 2^{2^{O(a \cdot k)} \cdot |\varphi|^2}.$$

This is doubly exponential in the input word length. More precisely, it is doubly exponential in the number of quantifiers and singly exponential in $|\varphi|$.

(iii) For languages of type N-N, we have

$$|\varphi'| = \exp^{a \cdot (k+2)}(O(|\varphi|)).$$

For input length l , this is l^2 -fold exponential in l and reduces to l -fold exponential for $k \geq 2$. More precisely, it is $a \cdot (k+2)$ -fold exponential in the length of the input formula. \square

Finally, we get the following complexity bounds for the general quantifier elimination problem in \mathbf{T}^* . These bounds are classified according to the Grzegorzcyk hierarchy. The Grzegorzcyk hierarchy defines the following complexity classes G_i , $i \geq 1$, for primitive recursive functions w.r.t. the input length (cf. [Gri99], [Sch74]):

G_1 contains all functions having constant runtime.

G_2 contains all functions having polynomial runtime.

G_3 contains all elementary recursive functions.

G_n contains all functions where the length of the input is in G_{n-1} , $n > 3$.

Corollary 3.3.1 (Complexity of quantifier elimination). (i) For type 1-1 and type N-1 languages, the quantifier elimination problem is in at most doubly exponential time-space. This is elementary recursive and thus in the third class of the Grzegorzcyk hierarchy.

(ii) For languages of type N-N, the quantifier elimination problem is not elementary recursive. It is, however, in the fourth class of the Grzegorzcyk hierarchy. For a bounded number of quantifiers, the quantifier elimination problem is elementary recursive, that means in the third class of the Grzegorzcyk hierarchy. \square

3 Elimination Procedure

Obviously, we can evaluate boolean combinations of variable-free \mathcal{L}^* -equations in \mathbf{T}^* in polynomial time, so we also get corresponding bounds for the decision problem:

Corollary 3.3.2 (Complexity of the decision problem). *(i) For languages of type 1-1 and N-1, the decision problem is in at most doubly exponential time-space. This is elementary recursive and thus in the third class of the Grzegorzcyk hierarchy.*

(ii) For languages of type N-N, the decision problem is not elementary recursive. It is, however, in the fourth class of the Grzegorzcyk hierarchy. For a bounded number of quantifiers, the quantifier elimination problem is elementary recursive, that means in the third class of the Grzegorzcyk hierarchy. \square

In the next chapter, we give an overview of the REDLOG-package dealing with expanded term algebras. Among other things, the elimination procedure together with all transformations described in the previous sections is implemented within this package.

4 Implementation

In this chapter, we describe the implementation of the elimination procedure introduced in section 3. In order to improve this procedure, we present some simplification techniques for \mathcal{L}^* -formulas and introduce another elimination method called *deep Gauss elimination* (see [Dol00]) for special kinds of input formulas.

The theory of term algebras and all concerning procedures are implemented as a new *context* within the comprehensive first-order logic package REDLOG [DS97a] of the computer algebra system REDUCE. REDLOG is part of REDUCE since version 3.7. It was developed by A. Dolzmann and T. Sturm at the University of Passau starting in 1995.

In the following sections, we describe the internal method of quantifier elimination in expanded term algebras. We will see that additional considerations may enormously improve the elimination procedure described in section 3.2.4. In section 4.1, we start with general information about the REDLOG context `talp` which comprises the entire term algebra part of REDLOG. In section 4.2, we introduce the simplifications performed on input formulas and intermediate results. In section 4.3, we focus our attention on the implementation of the elimination procedure described in section 3.2 together with the above-mentioned deep Gauss elimination procedure for special input formulas. We finish this chapter with section 4.4, in which we consider an illustrating example for quantifier elimination in expanded term algebras.

4.1 The REDLOG-package `talp`

In this section, we focus our attention on general statements about the term algebra part of REDLOG, called `talp`¹.

```
REDUCE 3.7, 15-Apr-1999, patched to 8-Jul-2004 ...
```

```
1: load_package redlog;
```

¹`talp` stands for Term Algebra Lisp Prefix and determines the theory (term algebra) and the internal term representation (LISP Prefix). All following computations are carried out w.r.t. the selected context.

4 Implementation

After loading REDLOG into REDUCE, a context determining a first-order language and a theory has to be selected (cf. [DS97a]). This decision has to be made in order to interpret all further formulas correctly. For the theory of term algebras one possible selection is the following:

```
2:  rlset(talp, {a,0}, {b,0}, {f,1}, {g,2});
```

In addition to the context identifier (`talp`) one has to pass the language elements as extra parameters to `rlset`. In the example above, we define $\mathcal{L} = \{a^{(0)}, b^{(0)}, f^{(1)}, g^{(2)}\}$ to be our first-order language including two constants a , b and two function symbols f and g with $\alpha(f) = 1$ and $\alpha(g) = 2$. After this selection, the function symbols of the expanded language are available as algebraic operators, i.e., one can use them to build up \mathcal{L}^* -terms. Due to the selected context, the only available relations are "=" and "≠". With these basic elements, first-order formulas can be input and all the available context-specific functions for processing them can be used, e.g.:

```
3:  phi := all(x, ex(y, y=f(x)));
    phi :=  $\forall x \exists y (y = f(x))$ 

4:  rlqe phi;
+++ eliminate block ex(y)
++ try gauss elimination for y ... succeeded
+++ eliminate block all(x)

    true
```

The user can control the output of the quantifier elimination functions `rlqe` and `rlqea` via a *switch* called `rlverbose`. Switches `<sname>` play very much the role of global variables. One can change their meanings with `on <sname>` and `off <sname>`. By default the switch `rlverbose` is on, offering further information on the elimination process at runtime. In the example above, lines starting with at least one "+" contain such additional information. By turning off `rlverbose`, the output is restricted to the elimination result:

```
5:  off rlverbose;

6:  rlqe phi;

    true
```

Before having a closer look at the internal process of the implemented quantifier

elimination procedures, we turn to the topic of *simplification* in the context of expanded term algebras.

4.2 Simplifications

In the following, we consider the field of *simplification* in theories of expanded term algebras over fixed finite languages. The notion of simplification refers to the transformation of an arbitrary input formula τ to an equivalent formula τ' , such that τ' is simpler than τ . Together with the elimination technique used here (see section 3.2.1), simplification plays an important role, because this elimination method often leads to deeply nested terms and highly redundant formulas. In order to provide useful information to the user, these formulas have to undergo some simplifications (cf. [DS97b]).

The first difficulty we are confronted with: When is a certain formula simpler than another, logically equivalent one? In our context there are certainly more concrete answers to this question than in other domains like the reals, because on the one hand, we do not have any relation symbols and on the other hand, we semantically cannot distinguish one function symbol from another. For example in the reals, we may have a simplification goal that prefers logically to algebraically encoded information, so that $a = 0 \vee b = 0$ would be considered simpler than the equivalent expression $ab = 0$.

In our context, we consider three simplification goals:

- The first one is to have a lower sum of depths among all terms of the simplified formula.
- The second one is to have a lower maximum depth among all terms of the simplified formula.
- The third one is to have a lower number of base formulas in the simplified formula.

The following example shows that sometimes these goals contradict one another. Therefore, we order these goals by importance: We prefer the first goal to the second one, and the second goal to the third one. The reason for this ordering is that due to the non-trivial interpretation of the inverse function symbols, terms of lower depth are considered simpler than deeply nested ones. Similarly, formulas with even a couple of base formulas of low depth provide still more comprehensible information to the user than a single base formula containing deeply nested terms.

4 Implementation

Example 4.2.1 (Simplification goals). Let $\mathcal{L}^* = \{a^{(0)}, b^{(0)}, f^{(1)}, \text{inv}_{f,1}^{(1)}, g^{(1)}, \text{inv}_{g,1}^{(1)}\}$ be the underlying expanded language. Consider the \mathcal{L}^* -equation

$$\text{inv}_{f,1}(\text{inv}_{g,1}(x)) = x.$$

A simpler equivalent of this formula is

$$x = a \vee x = b.$$

Note that this simplification is carried out in dependence of the given language (see section 4.2.2), i.e. with a different underlying expanded language, e.g. $\mathcal{L}^* = \{a^{(0)}, b^{(0)}, f^{(1)}, \text{inv}_{f,1}^{(1)}, g^{(1)}, \text{inv}_{g,1}^{(1)}, h^{(1)}, \text{inv}_{h,1}^{(1)}\}$, this simplification is incorrect. \diamond

In the following sections, we briefly describe two simplifiers. The first one, called *standard simplifier*, uses a simplifier for base formulas to handle arbitrary boolean combinations of these (see section 4.2.1). All activities of the standard simplifier are combined in the function `rlsimpl`. The second simplifier, called *special simplifier*, also handles arbitrary complex formulas but additionally tries to simplify these by replacement of \mathcal{L}^* -terms for \mathcal{L}^* -terms. Furthermore, the *special simplifier* carries out language-specific transformations like the one performed in the previous example (see section 4.2.2). All activities of the special simplifier are combined in the function `rltrygs`. The following figure gives an overview of all available simplifiers:

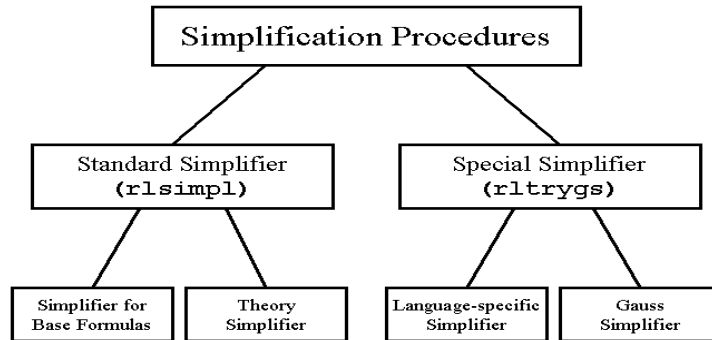


Figure 4.1: Simplification Procedures of `talp`

4.2.1 Standard Simplifier

The main focus of this section is on the basic simplification of arbitrary Boolean combinations of base \mathcal{L}^* -formulas.

On the basis of a lexicographical ordering of \mathcal{L}^* -terms, we obtain an ordering for \mathcal{L}^* -equations by first sorting w.r.t. the left hand side term and then w.r.t. the right hand side term of each equation. We use this ordering to sort equations within conjunctions and disjunctions. All input formulas are initially transformed in terms of this ordering. This helps us to identify and handle equal or inverted subformulas correctly.

The standard simplifier repeatedly applies the well-known equivalences listed next during the recursive simplification process described below (see [DS97b]):

$$\begin{array}{ll}
\neg \text{true} & \Leftrightarrow \text{false}, & \neg \text{false} & \Leftrightarrow \text{true}, \\
\text{true} \rightarrow \varphi & \Leftrightarrow \varphi, & \varphi \rightarrow \text{false} & \Leftrightarrow \neg \varphi, \\
\varphi \leftrightarrow \text{true} & \Leftrightarrow \varphi, & \varphi \leftrightarrow \text{false} & \Leftrightarrow \neg \varphi, \\
\varphi \leftrightarrow \varphi & \Leftrightarrow \text{true}, & \text{false} \rightarrow \varphi & \Leftrightarrow \text{true}, \\
\varphi \wedge \text{true} & \Leftrightarrow \varphi, & \varphi \vee \text{false} & \Leftrightarrow \varphi, \\
\varphi \wedge \text{false} & \Leftrightarrow \text{false}, & \varphi \vee \text{true} & \Leftrightarrow \text{true}
\end{array}$$

The basis of the standard simplifier for arbitrary boolean combinations of base \mathcal{L}^* -formulas is a simplifier for base formulas described next.

Simplifier for Base Formulas

We focus our attention on the simplification of arbitrary base \mathcal{L}^* -formulas, i.e. equations and negated equations between \mathcal{L}^* -terms.

The first action that we may expect from a simplifier for base formulas is to be able to transform \mathcal{L}^* -terms into the normal form introduced in section 3.1.1:

Example 4.2.2 (Simplification by transformation into normal form). *Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, \text{inv}_{f,1}^{(1)}, g^{(2)}, \text{inv}_{g,1}^{(1)}, \text{inv}_{g,2}^{(1)}\}$ be the underlying expanded language. Consider the \mathcal{L}^* -equation*

$$f\left(\text{inv}_{f,1}\left(g\left(\text{inv}_{g,1}(a), y\right)\right)\right) = \text{inv}_{g,1}\left(f\left(g(x, f(a))\right)\right).$$

According to the definition of inverse functions in section 2.2, we transform e.g. the right hand side term the following way: We compare the function symbol $\text{inv}_{g,1}$ with its argument term $f(g(x, f(a)))$. Due to the fact that the argument term does

4 Implementation

not start with the function symbol g specified in $inv_{g,1}$, the simplifier omits $inv_{g,1}$ and continues the transformation with the argument term. As an intermediate result, the simplifier for base formulas transforms the input formula into

$$f(g(a, y)) = f(g(x, f(a))). \quad \diamond$$

The second action that we may expect from the simplifier is to evaluate base formulas according to the \mathbf{T}^* -equivalence of \mathcal{L}^* -terms. Remember that two \mathcal{L}^* -terms t_1, t_2 are called \mathbf{T}^* -equivalent, if they describe the same function in \mathbf{T}^* . So we can simplify equations where the left hand side starts with a different \mathcal{L} -function symbol than the right one to **false**. Otherwise, if both hand sides start with the same \mathcal{L} -function symbols, we can restrict our attention to their arguments. This way we can continue to simplify the result of the previous example:

Example 4.2.3 (Simplification by evaluation). *Let \mathcal{L}^* be as in example 4.2.2. Consider the result of the previous example:*

$$f(g(a, y)) = f(g(x, f(a))).$$

According to the equivalence of \mathcal{L}^* -terms in \mathbf{T}^* the simplifier for base formulas transforms this into

$$x = a \wedge y = f(a). \quad \diamond$$

The procedure for simplifying arbitrary \mathcal{L}^* -equations and negated \mathcal{L}^* -equations is described in detail by Algorithm 4.2.1 on page 35.

The following last example indicates that it is important to distinguish between the function symbols of \mathcal{L} and the inverse function symbols of \mathcal{L}^* . Due to the non-trivial interpretation of the latter ones, we can not proceed the same way as in examples 4.2.2 and 4.2.3 when inverse function symbols are present. The following example indicates that even the simplification of basic \mathcal{L}^* -formulas is hard (see also section 4.2.2):

Example 4.2.4 (Simplification of inverse terms). *Let \mathcal{L}^* be as in example 4.2.2. Consider the \mathcal{L}^* -equation*

$$inv_{f,1}(inv_{f,1}(x)) = inv_{f,1}(x).$$

In this case, it is not correct to simplify the formula to $inv_{f,1}(x) = x$. Consider for example the assignment $x = f(a)$ to see that. On the other hand, take look at the formula

$$inv_{f,1}(inv_{g,1}(x)) = inv_{f,1}(x).$$

Here, this formula is actually equivalent to $inv_{g,1}(x) = x$. \diamond

Algorithm 4.2.1 (Simplification of Base Formulas).

Main procedure SBF

Input a base \mathcal{L}^* -formula φ **Output** a simplified equivalent \mathcal{L}^* -formula of φ

```

1  procedure SBF( $\varphi$ )
2  begin
3     $t_l := \text{simpterm}(\text{"left hand side term of } \varphi\text{"}, \text{nil})$ 
4     $t_r := \text{simpterm}(\text{"right hand side term of } \varphi\text{"}, \text{nil})$ 
5    if  $t_l == t_r$  then
6      return "true, if  $\varphi$  is an equation, false, if  $\varphi$  is a negated equation"
7    elseif "one of  $t_l, t_r$  starts with an inverse function" or " $t_l, t_r$  are variables"
8      or "one of  $t_l, t_r$  is a variable and the other one is a constant" then
9      return " $t_l = t_r$ , if  $\varphi$  is an equation,  $\neg(t_l = t_r)$ , if  $\varphi$  is a negated equation"
10   elseif "one of  $t_l, t_r$  is an  $\mathcal{L}$ -constant" or " $t_l$  and  $t_r$  start with different
11      $\mathcal{L}$ -function symbols" then
12     return "false, if  $\varphi$  is an equation, true, if  $\varphi$  is a negated equation"
13   elseif "one of  $t_l, t_r$  is a variable" then
14     Let w.l.o.g.  $t_l$  be a variable  $x$ .
15     if " $x$  is an immediate argument of an  $\mathcal{L}$ -function of  $t_r$ " then
16       return "false, if  $\varphi$  is an equation, true, if  $\varphi$  is a negated equation"
17     else
18       return " $t_l = t_r$ , if  $\varphi$  is an equation,  $\neg(t_l = t_r)$ , if  $\varphi$  is a negated equation"
19   end
20   elseif " $t_l, t_r$  start with the same  $\mathcal{L}$ -function symbol" then
21      $al_l := \text{"list of argument terms of } t_l\text{"}$ 
22      $al_r := \text{"list of argument terms of } t_r\text{"}$ 
23     result := true
24     while  $al_l \neq \text{nil}$  and  $al_r \neq \text{nil}$  and result  $\neq$  false do
25        $a_l := \text{first}(al_l)$ 
26        $a_r := \text{first}(al_r)$ 
27       result := SBF( $a_l = a_r$ )
28       if result  $\neq$  false and result  $\neq$  true then
29         if " $\varphi$  is an equation" then
30           list := cons(result, list)
31         else
32           list := cons( $\neg(\text{result})$ , list)
33       end
34     end
35      $al_l := \text{rest}(al_l)$ 

```

4 Implementation

```

35      $al_r := \text{rest}(al_r)$ 
36   od
37   if  $list \neq \text{nil}$  then
38     if  $|list| > 1$  then
39       if " $\varphi$  is an equation" then
40         return "conjunction of all elements of list"
41       else
42         return "disjunction of all elements of list"
43       end
44     else
45       return "the single element of list"
46     end
47   end
48   return "result, if  $\varphi$  is an equation,  $\neg(\text{result})$ , if  $\varphi$  is a negated equation"
49 end
50 end SBF

```

Sub procedure `simpterm`

Input an \mathcal{L}^* -term t and a stack s for inverse function symbols

Output an \mathcal{L}^* -term in normal form according to Lemma 3.1.1, equivalent to t

```

1 procedure simpterm( $t, s$ )
2 begin
3   if " $t$  starts with an inverse function symbol  $inv_{f,i}$ " then
4      $arg :=$  "argument term of  $t$ "
5     if " $arg$  is a variable" then
6       return "the term consisting of all inverse function symbols
7         of  $s$  in front of the variable  $arg$ "
8     elseif " $arg$  starts with an inverse function symbol  $inv$ " then
9       return simpterm( $arg, \text{push}(inv, s)$ )
10    elseif " $arg$  starts with  $f$ " then
11      return simpterm(" $i$ -th argument term of  $t$ ",  $s$ )
12    else
13      return simpterm( $arg, s$ )
14    end
15  else
16    if " $t$  starts with an  $\mathcal{L}$ -function symbol  $f$ " then
17       $t := \text{cons}(f, \text{for each } arg \text{ of } t \text{ collect } \text{simpterm}(arg, \text{nil}))$ 
18    end
19    if  $s \neq \text{nil}$  then

```

```

19         return simpterm(cons(top(s), t), pop(s))
20     else
21         return t
22     end
23 end
24 end simpterm

```

End of Algorithm 4.2.1

Theorem 4.2.1 (Simplification of Base Formulas). *Let φ be a base \mathcal{L}^* -formula. Algorithm 4.2.1 applied to φ terminates and simplifies φ w.r.t. the normal form for \mathcal{L}^* -terms stated in Lemma 3.1.2.*

Proof. **procedure simpterm(t, s):**

Termination: First of all, we can state that the **for**-loop of line 16 is finite, because we only consider functions with finite arity. We prove the termination of the procedure **simpterm** by induction on the sum n of $\Delta(t)$ and the number of inverse function symbols contained in s , denoted by **count**(s). Let $n = 0$. In this case we have no recursive calls, and the procedure terminates according to line 21. Let $n > 0$. Consider the recursive call of **simpterm**(t' , s') in line 8. Here, we still obtain n as the sum of $\Delta(t')$ and **count**(s'). Due to the fact that the depth of t is finite, we remain in this state at most $\Delta(t)$ -times until the procedure terminates according to lines 5-6, or until the argument term *arg* of t is not longer an inverse term. In the latter case, we obtain $n - 1$ as the sum of $\Delta(t')$ and **count**(s'). For the recursive calls in line 10 and 12, we immediately obtain $n - 1$ as the sum of $\Delta(t')$ and **count**(s'). The same applies to the recursive calls in line 16, so that we can apply the induction hypothesis to all these cases. For the remaining recursive call **simpterm**(t' , s') in line 19, we still obtain n as the sum of $\Delta(t')$ and **count**(s'). Due to the fact that in this case t' has the form $inv_{x,y}(f(t_1, \dots, t_k))$, we immediately obtain $n - 1$ as the sum of $\Delta(t'')$ and **count**(s'') in the subsequent call **simpterm**(t'' , s'') according to line 10 or 12. So we can also apply the induction hypothesis to this remaining recursive call, and state that **simpterm** terminates for all $n \in \mathbb{N}$.

Correctness: We prove the correctness of the procedure **simpterm** by induction on $n := |t|$. Let $n = 0$. In this case t is a variable or a constant. According to line 21, the procedure simply returns t . Let $n > 0$. If t is of the form $inv_{f,i}(t')$ for an n -ary function symbol f , an integer $i \in \{1, \dots, n\}$, and an \mathcal{L}^* -term t' , we distinguish four cases: First, if t' is a variable, then we simply return t according to line 6. Second, if t' starts with an inverse function symbol, we save $inv_{f,i}$ and restrict to the argument term t' according to line 8. Third, if t' is a constant or starts with an \mathcal{L} -function symbol different from f , we simply restrict to t' according to line 12. In the fourth case, t' is of the form $f(t_1, \dots, t_n)$ with \mathcal{L}^* -terms t_1, \dots, t_n . Here,

4 Implementation

according to line 16, we restrict to the argument term t_i . In all cases, we can apply the induction hypothesis and state the correctness of `simpTerm`. It remains to treat the case where t is of the form $f(t_1, \dots, t_n)$, where f is an n -ary function symbol, and t_1, \dots, t_n are \mathcal{L}^* -terms. In this case, we save f , and restrict to the argument terms t_i , according to line 16. By the induction hypothesis, we can compute normal forms for these terms t_i .

procedure `SBF`(φ):

Termination: First of all, we can state that the **while**-loop of line 22 is finite, because the argument lists of both terms t_l and t_r are finite. According to lines 33-34 both lists are shortened after each loop cycle. We show the termination of the procedure `SBF` by induction on $\Delta(\varphi)$. For $\Delta(\varphi) = 0$, we have no recursive calls and the algorithm terminates. Let $\Delta(\varphi) > 0$. For each pair of argument terms a_l and a_r we get one recursive call `SBF`($a_l = a_r$) with $\Delta(a_l = a_r) \leq \Delta(\varphi) - 1$ according to line 26. By the induction hypothesis all these recursive calls terminate.

Correctness: After the calls of `simpTerm` in lines 3 and 4, the left hand side term t_l and the right hand side term t_r are in normal form according to Lemma 3.1.2. We prove the correctness of the procedure `SBF` by induction on $n := \max(\{\Delta(t_l), \Delta(t_r)\})$. Let $n = 0$. If one term is a variable x (or a constant c), and the other one equals x (or c), we obtain the corresponding Boolean value according to line 6. If one term is a variable x , and the other one is a different variable y , the procedure returns $t_l = t_r$ ($\neg(t_l = t_r)$), according to line 9. The same applies to the case where one term is a variable and the other one is a constant. If one term is a constant a , and the other one is a different constant b , we obtain the corresponding Boolean value according to line 11. Let $n > 0$. According to line 6, we obtain the corresponding Boolean value for equal terms t_l and t_r . According to line 9, if (at least) one term starts with an inverse function symbol, we obtain the equation $t_l = t_r$, if φ is an equation, and the negated equation $\neg(t_l = t_r)$, if φ is a negated equation. In lines 10 to 11, `SBF` returns *false* for equations, if one hand side term is a constant and the other hand side term starts with an \mathcal{L} -function symbol. The same applies to equations $t_l = t_r$, where t_l and t_r start with different \mathcal{L} -function symbols. For negated equations, in these cases `SBF` returns *true*. In lines 12 to 18, we consider the case in which one hand side term is a variable x , and the other hand side term t starts with an \mathcal{L} -function symbol. Here, we have to distinguish two cases: if x is an immediate argument of an \mathcal{L} -function within t , we obtain *false*, if φ is an equation, and *true*, if φ is a negated equation. In the case that x is not an immediate successor of a \mathcal{L} -function within t , we return the (negated) equation consisting of the two terms x and t . In lines 19 to 49, we treat the remaining case that both hand side terms t_l and t_r of φ are terms in normal form, starting with the same \mathcal{L} -function symbol. Here, we can restrict to the argument terms of t_l and t_r , according to line 26. All equations $a_l = a_r$ of argument terms a_r, a_r ,

are recursively simplified with **SBF**, having a maximum depth of $n - 1$. So we can apply our induction hypothesis to these calls of **SBF**. The simplifications of these equations are processed and stored separately according to lines 26 to 32. In lines 37 to 47 the results of these simplifications are reassembled in a conjunction, if φ is an equation, or in a disjunction if φ is a negated equation. So we can state that each possible form of φ is handled correctly. \square

Theory Simplifier

Now back to the standard simplification of complex formulas. The standard simplifier uses the previously described simplifier for base formulas in order to handle arbitrary complex formulas. It is best described by the notion of "theory simplification". In the following, a *theory* Φ is a set of base formulas considered as a conjunction. A formula φ is simplified to a formula φ' w.r.t. a given theory Φ , if

$$\bigwedge \Phi \rightarrow (\varphi \leftrightarrow \varphi').$$

Simplification w.r.t. a given theory is based on the following observation (cf. [DS97b]):

Proposition 4.2.1. *For arbitrary quantifier-free formulas φ and ψ the following equivalences hold:*

$$\begin{aligned} \varphi \wedge (\dots \psi \dots) &\Leftrightarrow \varphi \wedge (\dots (\varphi \wedge \psi) \dots), \\ \varphi \vee (\dots \psi \dots) &\Leftrightarrow \varphi \vee (\dots (\neg \varphi \wedge \psi) \dots). \end{aligned}$$

The same applies to the corresponding dual variants

$$\begin{aligned} \varphi \wedge (\dots \psi \dots) &\Leftrightarrow \varphi \wedge (\dots (\neg \varphi \vee \psi) \dots), \\ \varphi \vee (\dots \psi \dots) &\Leftrightarrow \varphi \vee (\dots (\varphi \vee \psi) \dots). \end{aligned} \quad \square$$

The idea is to apply the implication part of the corresponding equivalence, then to simplify and to hope that the result will indeed be simpler than the original formula. More precisely we proceed as follows: The input formula is run through recursively. On every Boolean level, we enlarge the theory in dependence on the Boolean operator at this level. According to the previous proposition, in conjunctions we add all base formulas as they are. In disjunctions, we add the negations of all base formulas. We transform equivalences and replications of the input formula to implications, from which basic premises are added themselves to the theory and basic conclusions are added negated. After each addition, the theory itself is simplified. If the theory becomes inconsistent, the whole considered subformula is **false**. This technique of theory inheritance is the content of the following proposition (cf. [DS97b]):

4 Implementation

Proposition 4.2.2. *Let Θ be a theory and ψ be a formula. Let Γ be a finite set of base formulas and denote by $\bar{\Gamma}$ the same set with negated base formulas.*

(i) *Let ψ' be such that $\Theta \cup \Gamma \models (\psi \Leftrightarrow \psi')$. Then*

$$\Theta \models \left(\bigwedge \Gamma \wedge \psi \Leftrightarrow \bigwedge \Gamma \wedge \psi' \right).$$

(ii) *Let ψ'' be such that $\Theta \cup \bar{\Gamma} \models (\psi \Leftrightarrow \psi'')$. Then*

$$\Theta \models \left(\bigvee \Gamma \vee \psi \Leftrightarrow \bigvee \Gamma \vee \psi'' \right). \quad \square$$

We conclude this section with two illustrating examples for the internal process of the standard simplifier for arbitrary complex formulas:

Example 4.2.5. *Let $\mathcal{L}^* = \{a^{(0)}, b^{(0)}, f^{(1)}, \text{inv}_{f,1}^{(1)}\}$ be our expanded language.*

(i) *Consider the input formula*

$$\varphi \equiv \text{inv}_{f,1}(x) \neq x \wedge (y = b \vee (z = a \wedge \text{inv}_{g,1}(x) \neq x)).$$

*We start with the empty theory. By recursively traversing φ , we successively add $\text{inv}_{f,1}(x) \neq x$, $y = b$ and $z = a$. At the innermost level, we can apply $\text{inv}_{f,1}(x) \neq x$ of the current theory to $\text{inv}_{g,1}(x) \neq x$, yielding **false**. This is because x has to decide with which one of the two function symbols f and g it starts. So the final result of the simplification is*

$$\text{inv}_{f,1}(x) \neq x \wedge y = b.$$

(ii) *Consider the input formula*

$$\varphi \equiv w = a \wedge \underbrace{(x \neq b \vee (y = f(a) \wedge z \neq a))}_{\varepsilon} \wedge \underbrace{(z = a \vee w \neq a)}_{\sigma}.$$

Starting with the empty theory, we initially add $w = a$. With the new theory $\{w = a\}$, we recursively simplify the two remaining subformulas ε and σ of the top-level conjunction: ε does not change under the current theory $\{w = a\}$, but σ simplifies to $z = a$, which is now on the top-level of φ and thus becomes part of the theory. Finally, with the new theory $\{w = a, z = a\}$, ε simplifies to $x \neq b$, yielding as the final result

$$w = a \wedge x \neq b \wedge z = a. \quad \diamond$$

All so far mentioned simplifications of base formulas and complex formulas are combined in the REDLOG-function for standard simplification, called `rlsimpl`. The following example illustrates its use:

```

7: phi := inv_f1(x) <> x and (y = b or (z = a and inv_g1(x) <> x));
   phi := inv_f,1(x) ≠ x ∧ (y = b ∨ (z = a ∧ inv_g,1(x) ≠ x));

8: rlsimpl phi;
   inv_f,1(x) ≠ x ∧ y = b

```

In the next section, we focus on the special simplifier for the context of term algebras.

4.2.2 Special Simplifier

This section focusses on a special simplifier for arbitrary input formulas (see Figure 4.1). The main focus lies on the depth reduction of inverse terms. Internally, the special simplifier mainly consists of a simplifier called *Gauss simplifier*. It tries to simplify input formulas by replacement of \mathcal{L}^* -terms by \mathcal{L}^* -terms. The second part of the special simplifier focusses on language specific simplification, i.e. we try to simplify input formulas w.r.t. the current underlying language. As the next section suggests this first part of the special simplifier only handles special kinds of input formulas and underlying languages.

Language-specific Simplification

As already mentioned in section 4.2.1, the occurrence of deeply nested inverse functions fairly complicates the intuitive meaning even of base formulas. Unfortunately, it also complicates automatic simplifications. Even under consideration of the underlying expanded language \mathcal{L}^* , the problem of simplifying (negated) equations with deeply nested inverse terms like $inv_{f,1}(inv_{f,1}(x)) = inv_{f,1}(x)$ can't be solved.

So the language-specific part of the special simplifier so far focusses on base formulas $inv(x) \rho x$ where $inv(x)$ is an arbitrary inverse term with variable x as argument, and $\rho \in \{=, \neq\}$. The occurrence of such formulas in connection with the underlying language often allows simplifications. In the following, we give some examples for such simplifications performed by the language-specific part of the special simplifier.

Example 4.2.6 (Language-specific Simplifications (1)). Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, inv_{f,1}^{(1)}, g^{(1)}, inv_{g,1}^{(1)}\}$ be the underlying expanded language. Consider the for-

4 Implementation

mula

$$\varphi \equiv \text{inv}_{f,1}(x) = x \wedge \text{inv}_{g,1}(x) = x.$$

Based on \mathcal{L}^* , we can simplify φ to

$$x = a \vee x = b.$$

To see that, remember that each of the basic subformulas of φ states that x does not start with the corresponding function symbol. \diamond

The previous example leads to the next one, where we consider an analog situation.

Example 4.2.7 (Language-specific Simplifications (2)). Let \mathcal{L}^* be as in the previous example. Consider the formula

$$\varphi \equiv \text{inv}_{f,1}(\text{inv}_{g,1}(x)) \neq x.$$

Keeping \mathcal{L}^* in mind, we can simplify φ to

$$x \neq a \wedge x \neq b.$$

According to Example 4.2.1, we get the dual result when considering the formula $\text{inv}_{f,1}(\text{inv}_{g,1}(x)) = x$. \diamond

Remember from Definition 3.2.1 in section 3.2.2 that the maximum depth of all terms in a considered quantified input formula strongly affects the number of elements in the corresponding elimination set, so that the simplification above actually does pay. Furthermore, we will see that the resulting conjunction may be helpful for the main part of the special simplifier, the Gauss simplifier introduced in the next section.

Obviously, we can always simplify base formulas $\text{inv}_{f,1}(\dots(\text{inv}_{f,1}(x))\dots) \rho x$ to $\text{inv}_{f,1}(x) \rho x$ for $\rho \in \{=, \neq\}$, independent from the underlying language as long as all nested inverse function symbols are identical. So far, the language-specific simplifier only handles very special cases of input formulas and underlying languages. It would be desirable to have available a comprehensive simplification method dealing with arbitrary kinds of deeply nested inverse functions like $\text{inv}_{f,1}(\text{inv}_{f,1}(\text{inv}_{f,1}(x))) = \text{inv}_{f,1}(\text{inv}_{f,1}(x))$. Unfortunately, even under consideration of the underlying language, there is no facility for simplifying these kinds of formulas, even though it looks easy.

In the next section, we introduce the main part of the special simplifier. It embarks on a totally different simplification strategy.

Gauss Simplification

The idea of Gauss simplification is to use the information obtained from conjunctively combined equations between \mathcal{L}^* -terms in input formulas, intermediate or final results. So we first transform the considered input formula into an equivalent formula φ in refined normal form, where φ is an \wedge - \vee -combination of base formulas using the well-known transformation rules. Then at each Boolean level of φ , we embark on the following strategy:

- (i) If we have a conjunction at the current Boolean level, we extract from the set of equations $t = t'$ occurring at this level, the set Σ of replacement pairs t/t' . Otherwise, we proceed with $\neg\varphi$ and remember to undo this transformation at the end of the simplification for the current level.
- (ii) We enlarge Σ to the set of pairs Σ' , according to transitivity correlations between variables u, v in equations $u = v$. That means, if we have a pair u, v of variables in Σ , then every term t we replace for v also can be replaced for u . Thus, we add these pairs u/t to Σ' .
- (iii) We process the replacement pairs of Σ' one by one. For each pair t/t' , we replace all further occurrences of t' at the current and all higher Boolean levels by t , and store the replacement result simplified with the standard simplifier, if it is better than the so far best result. This is done until there are no more pairs left or until we get a Boolean value as a replacement result.

Note that in step (iii), instead of storing all replacement results, we only store the so far best result ψ according to our simplification goals.

In order to visualize the procedure, we give some illustrating examples of Gauss simplifications:

Example 4.2.8 (Gauss Simplification(1)). Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, inv_{f,1}^{(1)}, g^{(1)}, inv_{g,1}^{(1)}\}$ be the underlying expanded language. Consider the formula

$$\varphi \equiv x \neq a \wedge x = u \wedge v = u \wedge v = a.$$

We first extract the set $\{x = u, v = u, v = a\}$ of equations of φ . Accordingly, we get the set $\Sigma = \{x/u, v/u, v/a\}$ of replacement pairs for φ . Due to the transitivity correlations between x, v and u we add the pairs $u/x, u/v, x/v, v/x, u/a, x/a$ leading to the final set $\Sigma' = \{x/u, v/u, v/a, u/x, u/v, x/v, v/x, u/a, x/a\}$ of substitution pairs for φ . For each of these pairs, we replace all further occurrences of the second entry within φ by the first one. We iteratively obtain the best result by replacing u with a yielding **false**. \diamond

4 Implementation

Note that in contrast to the standard simplifier the time for simplifying arbitrary formulas φ cannot be neglected here. This is due to the fact that we obtain $O(eq(\varphi)^2)$ replacement pairs. So we have to carefully select the point in time at which we want to use the Gauss simplifier during the elimination process (see section 4.3.1). The following example makes clear that we get replacement pairs not only from equations between zero depth elements. In fact, this is the crucial point of the Gauss simplifier in order to fulfill its primary purpose, which is to decrease the depth of deeply nested inverse terms.

Example 4.2.9 (Gauss Simplification(2)). Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, inv_{f,1}^{(1)}, g^{(1)}, inv_{g,1}^{(1)}\}$ be the underlying expanded language. Consider the formula

$$\varphi \equiv inv_{f,1}(x) \neq x \vee inv_{g,1}(inv_{f,1}(x)) = inv_{g,1}(x).$$

According to (i) of the description above, we consider the negation of φ leading to the formula

$$\varphi' \equiv inv_{f,1}(x) = x \wedge inv_{g,1}(inv_{f,1}(x)) \neq inv_{g,1}(x).$$

From the first equation of φ' , we extract the replacement pairs $inv_{f,1}(x)/x$ and $x/inv_{f,1}(x)$. The first pair does not help to reduce the sum of all depths, the maximum depth or the number of base formulas. The second pair, however, leads to the following intermediate result

$$\varphi'' \equiv inv_{f,1}(x) = x \wedge inv_{g,1}(x) \neq inv_{g,1}(x).$$

After applying the standard simplifier to φ'' we get **false**. Finally, we have to negate φ'' in order to undo the transformation from φ to φ' . So we arrive at **true** as the final result. \diamond

As mentioned earlier, the special simplification procedure comprises both, the language-specific simplifier and the Gauss simplifier, whereas the former one is applied first. The application area of the special simplifier within the elimination process is described in the following section. The special simplifier has also been made available to the user via the function `rltrygs`:

```

1: load redlog;
2: rlset(talp, {a,0}, {b,0}, {f,1}, {g,1});
3: phi := inv_f1(inv_g1(x)) = x and x <> b;
   phi := inv_f1(inv_g1(x)) = x \wedge x \neq b

```

```
4: talptry phi;
   x = a
```

Now we leave the field of simplifications and turn to the implemented elimination procedures.

4.3 Elimination Algorithms

We now focus on the elimination of the quantifiers in a first-order formula ψ . First of all, we convert ψ to prenex normal form. Furthermore, the matrix φ of ψ is transformed into an \wedge - \vee -combination of base formulas according to the well-known transformation rules. That means, after this first conversion, ψ has the following form:

$$\psi(u_1, \dots, u_m) \equiv Qx_1 \dots Qx_n(\varphi)(x_1 \dots, x_n, u_1, \dots, u_m),$$

where φ is a quantifier-free \wedge - \vee -combination of \mathcal{L}^* -equations and negated \mathcal{L}^* -equations containing besides x_1, \dots, x_n possibly other free variables u_1, \dots, u_m , and $Q \in \{\exists, \forall\}$.

In section 4.3.1, we concentrate on the internal process of the elimination described in detail in section 3.2. In section 4.3.2, we introduce a very fast elimination method called *deep Gauss elimination*. Unfortunately, this method is applicable only to special types of input formulas. In section 4.3.3, we will see how we can exploit the freedom to permute quantifiers within blocks of like quantifiers.

4.3.1 Quantifier Elimination by Substitution of Parametric Test Terms

In this section, we describe the internal process of quantifier elimination in expanded term algebras as introduced in section 3.2. As mentioned above, we can concentrate on prenex input formulas of the form

$$\psi(u_1, \dots, u_m) \equiv Qx_1 \dots Qx_n(\varphi)(x_1, \dots, x_n, u_1, \dots, u_m).$$

In Corollary 3.2.1, we stated how to eliminate an existential quantifier $\exists x$ in front of a quantifier-free formula φ . In order to eliminate a universal quantifier, we made use of the equivalence $\forall x\varphi \leftrightarrow \neg\exists x\neg\varphi$. This transformation, together with the equivalence

$$\mathbf{T}^* \models \exists x\varphi \longleftrightarrow \bigvee_{t \in R_{\Phi}} \varphi[t/x]$$

of Corollary 3.2.1, enables us to eliminate the quantifiers in ψ one by one, starting with the innermost one. As the following example suggests, the main problem

4 Implementation

we have to cope with, is the size of the elimination set R_Φ . It turns out to be impossible in general to generate R_Φ as a whole, because this would exceed the available memory. Unfortunately, this already happens for quite simple input formulas and underlying languages:

Example 4.3.1 (Size of R_Φ). Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, \text{inv}_{f,1}^{(1)}, g^{(2)}, \text{inv}_{g,1}^{(1)}, \text{inv}_{g,2}^{(1)}\}$ be the underlying expanded language. Consider the input formula

$$\psi \equiv \exists x (g(x, f(y)) = g(u, x)).$$

In order to get a finite elimination set for ψ according to section 3.2.2, we have to consider the refined normal form

$$\psi' \equiv \exists x (x = u \wedge x \neq \text{inv}_{f,1}(x) \wedge y = \text{inv}_{f,1}(x))$$

of ψ . W.r.t. the set Φ of base formulas of ψ' , we get $\mathcal{B} = 6$ as the estimated depth bound for the terms of R_Φ . There are more than 10^{40} \mathcal{L}^* -terms t with $\Delta(t) \leq 6$. With each increase of \mathcal{B} from n to $n + 1$, $\#R_\Phi$ approximately increases quadratically. This is due to the fact that the maximum arity of all function symbols of \mathcal{L}^* is 2. \diamond

One way out is to enumerate R_Φ and to hope that we already get a result from one of the first substitutions. That means, we compute and finally process the terms of R_Φ one by one. The next section describes in detail our procedure for enumerating these \mathcal{L}^* -terms.

Term Enumeration

Our strategy to enumerate the set R_Φ of \mathcal{L}^* -terms is the following: Given a term t_i , we want to compute its unique successor t_{i+1} by a kind of depth-first search in the arguments of t_i . At that purpose, the procedure makes use of the following objects:

- The current term t_i .
- The underlying expanded language \mathcal{L}^* .
- The set of free variables $\mathcal{V}_f(\varphi)$ occurring in the considered formula φ .
- The depth bound \mathcal{B} estimated w.r.t. the set of base formulas in φ .

What we like to have is a procedure `next` to which we can pass these objects and which returns the new term t_{i+1} :

$$t_{i+1} := \text{NEXTT}(t_i, \mathcal{B}, l).$$

The following algorithm describes our procedure for enumerating the \mathcal{L}^* -terms of the elimination set R_Φ .

Algorithm 4.3.1 (Term Enumeration).**Main procedure NEXTT**

Input *an empty term nil or an \mathcal{L}^* -term t_{old} , an integer \mathcal{B} (depth bound), a list of variables l*

Output *an empty term nil or the \mathcal{L}^* -term following t_{old}*

```

1  procedure NEXTT( $t_{old}$ ,  $\mathcal{B}$ ,  $l$ )
2  begin
3     $fv$  := "vector of  $\mathcal{L}^*$ -function symbols  $f_i$  followed by the corresponding
           inverse function symbols"
4     $ifv$  := "vector of inverse function symbols grouped by the  $\mathcal{L}$ -function
            they belong to"
5     $cvv$  := "vector of constants and variables"
6     $vv$  := "vector of variables corresponding to the list of variables  $l$ "
7     $cd$  := "current depth, initially 0"
8     $fit$  := "boolean value, states whether the predecessor symbol of a term
            is an inverse function symbol, initially false"
9    if  $t_{old} = \text{nil}$  then
10     return first( $cvv$ )
11  else
12     return NEXTT1( $t_{old}$ ,  $cd$ ,  $\mathcal{B}$ ,  $cvv$ ,  $vv$ ,  $fv$ ,  $ifv$ , false)
13  end
14 end NEXTT

```

Sub procedure NEXTT1

Input *an \mathcal{L}^* -term t_{old} , an integer cd , an integer \mathcal{B} , vectors cvv , vv , fv , ifv and a Boolean value fit*

Output *nil or the \mathcal{L}^* -term following t_{old}*

```

1  procedure NEXTT1( $t_{old}$ ,  $cd$ ,  $\mathcal{B}$ ,  $cvv$ ,  $vv$ ,  $fv$ ,  $ifv$ ,  $fit$ )
2  begin
3    if  $\Delta(t_{old}) = 0$  then
4      if  $fit$  and " $t_{old}$  has a successor  $succ$  in  $vv$ " or
         not  $fit$  and " $t_{old}$  has a successor  $succ$  in  $cvv$ " then
5        return  $succ$ 
6      elseif  $cd < \mathcal{B}$  then
7        if  $fit$  then
8          return "first( $ifv$ ) initialized with first( $vv$ )"
9        else

```

4 Implementation

```

10         return "first(fv) initialized with first(cvv)"
11     end
12     else
13         return nil
14     end
15 end
16 if "told starts with an inverse function symbol" then
17     arg := "argument term of told"
18     newarg := NEXTT1(arg, cd+1, B, cvv, vv, fv, ifv, true)
19     if newarg ≠ nil then
20         return "told, with new argument term newarg"
21     end
22 else
23     for each "argument term arg of told" do
24         newarg := NEXTT1(arg, cd+1, B, cvv, vv, fv, ifv, fit)
25         if newarg ≠ nil then
26             tnew := "told, where arg of told is updated to newarg"
27             updated := true
28             break
29         else
30             "reset arg to first(cvv)"
31         end
32     od
33 end
34 if not updated then
35     if fit then
36         return "if next(told, ifv) exists, initialize with first(vv) else nil"
37     else
38         return "if next(told, fv) exists, initialize with first(cvv) else nil"
39     end
40 else
41     return tnew
42 end
43 end NEXTT1

```

End of Algorithm 4.3.1

Theorem 4.3.1 (Term Enumeration). *Let \mathcal{B} be an integer, let l be a list of variables and let t be either the empty term `nil` or an \mathcal{L}^* -term previously returned from the procedure `NEXTT`. Algorithm 4.3.1 applied to t terminates, and returns the next \mathcal{L}^* -term in normal form, following t . Furthermore, the algorithm can be*

used to successively generate all \mathcal{L}^* -terms t' in normal form with $\Delta(t') \leq \mathcal{B}$.

Proof. **procedure** NEXTT1(t_{old} , cd , \mathcal{B} , cvv , vv , fv , ifv , fit):

Termination: We prove the termination of procedure NEXTT1 by induction on $n := \Delta(t_{old})$. Let $n = 0$. In this case, we have no recursive calls of NEXTT1 and the procedure terminates according to lines 3 to 14. Let $n > 0$. We have to distinguish two cases: First, if t_{old} starts with an inverse function symbol, we get one recursive call NEXTT1(arg , $cd+1$, \mathcal{B} , cvv , vv , fv , ifv , $true$) in line 18, where arg is the argument term of t_{old} . So we have $\Delta(arg) = n - 1$, and we can apply our induction hypothesis to this call of NEXTT1. Second, if t_{old} starts with an \mathcal{L} -function symbol f , we get at most $\alpha(f) < \infty$ recursive calls NEXTT1(arg_i , $cd+1$, \mathcal{B} , cvv , vv , fv , ifv , fit), where arg_i is the i -th argument term of t_{old} , with $1 \leq i \leq \alpha(f)$. For each of these calls we have $\Delta(arg_i) = n - 1$, and thus we can apply our induction hypothesis to these calls.

Correctness: We prove the correctness of NEXTT1(t_{old} , cd , \mathcal{B} , cvv , vv , fv , ifv , fit) by induction on $n := \mathcal{B} - cd$. Initially, according to lines 9 to 13 of procedure NEXTT, t_{old} is equal to $\mathbf{first}(cvv)$. For fixed parameters \mathcal{B} , cvv , vv , fv and ifv , we prove that when iteratively applying $t_{old} := \mathbf{NEXTT1}(t_{old}, cd, \mathcal{B}, cvv, vv, fv, ifv, fit)$ until $t_{old} := \mathbf{nil}$, we successively obtain all \mathcal{L}^* -terms t in normal form with $\Delta(t) \leq \mathcal{B}$ exactly once. The condition that on the one hand, we do not generate an arbitrary \mathcal{L}^* -term more than once, and that on the other hand, we do not leave out any \mathcal{L}^* -terms, fundamentally relies on the fact that we have fixed orders in all vectors of function symbols and zero depth elements. In the following proof, we will explicitly point out how this fact guarantees to fulfill the above mentioned conditions:

Let $n = 0$. In this case, $\Delta(t_{old}) = 0$, because $cd = \mathcal{B}$ implies that either t_{old} itself is a zero depth element, and $\mathcal{B} = 0$, or that we descended the tree t_{old} via recursive calls of NEXTT1 to the innermost level, and $\mathcal{B} > 0$. According to lines 4 to 5, if $fit = true$, then t_{old} is a variable, and we try to return the next variable of vv following t_{old} . In doing so, we make sure to generate terms in normal form. If there is no such variable, i.e. t_{old} is the last variable of vv , we return \mathbf{nil} in line 13. This is due to the fact that $n = 0$ forbids us to generate terms of depth > 0 (see lines 6 to 11). If $fit = false$, t_{old} is either a variable or a constant. Here, we try to return the next element of cvv following t_{old} . If there is no such element, i.e. t_{old} is the last element of cvv , we are again forced to return \mathbf{nil} in line 13, according to the fact that $n = 0$. In the case $n = 0$, we can see how the fixed orders within cvv and vv guarantee us the conditions mentioned above: we exclusively process these vectors in one direction, starting from the position t_{old} has within these vectors. For $n = 0$, if this position is the last one within the corresponding vector, we have to return \mathbf{nil} .

4 Implementation

Let $n > 0$. Here, we have to distinguish two cases: First, let $\Delta(t_{old}) = 0$. As in the case $n = 0$, we first try to return the successor of t_{old} in vv or cvv , respectively, according to lines 4 to 5. In contrast to the case $n = 0$, if it fails, we proceed in line 7. If $fit = true$, stating that t_{old} is an argument of an inverse term and therefore a variable, we return the first inverse function symbol $inv_{f,1}$ of ifv initialized with the first variable from vv , according to line 8. Otherwise, if $fit = false$, we return the first function symbol f of fv initialized with the first element of cvv (line 10). Due to the fact that $n > 0$, we are now allowed to generate terms of depth > 0 . Note that, in both cases, we initialized $inv_{f,1}$ and f , respectively, each with the first element of the vectors vv and cvv , respectively. In the case $n = 0$, we stated the method of processing these vectors in one direction for each current depth level δ . Due to the initialization of $inv_{f,1}$ and f , respectively, with the first element of the corresponding vectors in line 8 and line 10, the method of processing vv and cvv solely in one direction, can be correctly applied in the next depth level $\delta + 1$.

Second, let $\Delta(t_{old}) > 0$. According to line 16, if t_{old} starts with an inverse function symbol, we try to update the argument term arg of t_{old} via the recursive call $NEXTT1(arg, cd+1, \mathcal{B}, cvv, vv, fv, ifv, true)$ in line 18. Due to the fact that we increased cd by one, we can apply the induction hypothesis to this call of $NEXTT1$. If this call was successful, we return the updated term in line 20. In the case that t_{old} starts with an \mathcal{L} -function symbol f , we start in line 23. Here, we have to consider each of the argument terms arg of t_{old} , as long as we did not update one of them. Thus, we get at most $\alpha(f)$ recursive calls $NEXTT1(arg, cd+1, \mathcal{B}, cvv, vv, fv, ifv, fit)$ to each of which we can apply our induction hypothesis. If a recursive call of $NEXTT1$ was successful for an argument term arg , we replace arg by the updated term according to line 26. Additionally, we keep in mind that we already updated t_{old} by setting the variable $updated$ to $true$ in line 27. If a recursive call was not successful for arg , we have to reset arg to the first element of cvv (see line 30). According to line 34, if we already updated t_{old} , we simply return t_{new} in line 41. Otherwise, if $updated = false$, we have to update the function symbol at the current depth level, i.e., we have already generated all \mathcal{L}^* -terms t , with $\Delta(t) \leq \mathcal{B}$ which start with the \mathcal{L}^* -function symbol t_{old} starts with. If $fit = true$, we try to return the next inverse function symbol from ifv and initialize it with the first variable of vv (line 36). Otherwise, if $fit = false$, we try to return the next \mathcal{L} -function symbol of fv and initialize it with the first element of cvv (line 38). In both cases, if it fails, we return nil . We can see that the method of solely processing cvv and vv at each depth level in one direction, starting from the current position, also applies to the vectors of \mathcal{L}^* -function symbols ifv and fv .

procedure $NEXTT(t_{old}, \mathcal{B}, l)$:

Termination: The termination of the procedure $NEXTT$ results from the termination of $NEXTT1$.

Correctness: In line 10, if t_{old} is the empty term `nil`, the procedure returns the first element of cvv . Due to the fact, that we exclusively consider finite languages with at least one constant and at least one function symbol of positive arity, such an element always exists. The correctness of NEXTT for non-empty input terms results from the correctness of NEXTT1. \square

The action of the procedure NEXTT is now briefly described with an example:

Example 4.3.2 (Term Enumeration). Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, inv_{f,1}^{(1)}, g^{(2)}, inv_{g,1}^{(1)}, inv_{g,2}^{(1)}\}$ be the underlying expanded language. Let $\mathcal{B} = 2$ be the depth bound, $l = \{u\}$ the set of variables and `nil` the current term. The zero depth elements, the variables, the inverse function symbols, as well as all \mathcal{L}^* -function symbols of positive arity, are stored and sorted within four vectors. According to these vectors and the depth bound, we would get the following sequence of terms. Due to the frequently mentioned problem with term sets, we only highlight interesting parts of the sequence:

$$\begin{aligned} a \rightarrow u \rightarrow f(a) \rightarrow f(u) \rightarrow f(f(a)) \rightarrow f(f(u)) \rightarrow f(inv_f1(u)) \rightarrow \dots \\ f(g(a,a)) \rightarrow f(g(u,a)) \rightarrow f(g(a,u)) \rightarrow f(g(u,u)) \rightarrow f(inv_g1(u)) \\ \rightarrow f(inv_g2(u)) \rightarrow inv_f1(u) \rightarrow \dots \rightarrow g(a,a) \rightarrow \dots \end{aligned} \quad \diamond$$

As the example suggests, the procedure always updates all zero depth elements at the current position before ascending the next stage. Therefore, we talked about "a kind of" depth-first search when describing the procedure. This procedure is now used to generate the terms of our elimination sets R_Φ one by one.

Back to our initial problem: We would like to eliminate the quantifiers in

$$\psi(u_1, \dots, u_m) \equiv Qx_1 \dots Qx_n(\varphi)(x_1, \dots, x_n, u_1, \dots, u_m).$$

We successively eliminate all quantifiers of ψ starting with the innermost one by passing through the following steps:

1. Consider the set Φ of base formulas of φ and estimate a depth bound \mathcal{B} w.r.t. Φ . Consider the set $R_\Phi = \{t_0, \dots, t_r\}$ with $\Delta(R_\Phi) \leq \mathcal{B}$, containing all \mathcal{L}^* -terms which we have to substitute for the current bound variable x_i .
2. While the special simplification of $\varphi[t_k/x_i]$ does not yield `true`, we consider $\bigvee_{j=0}^k \varphi[t_j/x_i]$, the disjunction of the current and all k previous substitution results. While the standard simplification of this conjunction does not yield `true`, we have to compute the next term t_{k+1} and restart at the current step.

4 Implementation

3. As long as there are quantifiers left, we restart at step 1 with $\bigvee_{j=0}^r \varphi[t_j/x_i]$ as the new input formula, otherwise we return the disjunction of step 2 as the elimination result.

Notice that for the elimination of like quantifiers, each disjunction obtained in step 2 can be handled separately according to the equivalence

$$\exists x_k(\varphi_1 \vee \dots \vee \varphi_r) \leftrightarrow \exists x_k(\varphi_1) \vee \dots \vee \exists x_k(\varphi_r).$$

Remember that the number of equations as well as the maximal depth of terms in φ strongly impact the depth bound and therefore the size of R_Φ . Interchanging the disjunction with the next quantifier decreases the complexity class for single quantifier blocks from doubly exponential to singly exponential in the number of quantifiers (cf. [Wei88]).

The following algorithm describes in detail our procedure for regular quantifier elimination in expanded term algebras \mathbf{T}^* .

Algorithm 4.3.2 (Regular Quantifier Elimination).

Input *a first-order \mathcal{L}^* -formula φ*

Output *a quantifier-free \mathcal{L}^* -formula ψ equivalent to φ*

```

1  procedure RQE( $\varphi$ )
2  begin
3     $\varphi' :=$  "prenex normal form of  $\varphi$  with quantifier blocks  $Q_b, \dots, Q_1$ ,
          where the matrix is an  $\wedge$ - $\vee$ -combination of base formulas"
4     $\psi :=$  "matrix of  $\varphi'$ "
5    if  $\psi = \text{true}$  or  $\psi = \text{false}$  then
6      return  $\psi$ 
7    end
8    for  $k := 1$  to  $b$  do
9       $bvl :=$  "list of bound variables of  $Q_k$ "
10     if " $Q_k$  is a block of universal quantifiers" then
11       $\psi := \neg\psi$ , retransformed into an  $\wedge$ - $\vee$ -combination of base formulas"
12     end
13     for each  $x \in bvl$  do
14       $\rho := \text{false}$ 
15      if " $\psi$  is a disjunction" then
16         $l :=$  "list of disjuncts of  $\psi$ "
17      else
18         $l := \{\psi\}$ 
19     end

```

```

20   for each "subproblem  $c$  of  $l$ " do
21      $\psi_G :=$  "result of trying deep Gauss elimination for  $c$  w.r.t.  $x$ "
22     if  $\psi_G \neq$  "failed" then
23        $\rho' := \psi_G$ 
24     else
25        $c :=$  "c exclusively containing inverse terms and constants"
26        $\rho' :=$  false
27        $fv :=$  "list of free variables of  $c$  without  $x$ "
28        $\mathcal{B} :=$  "depth bound estimated w.r.t.  $\mathcal{L}$ ,  $eq(c)$ ,  $\Delta(c)$  and  $fv$ "
29        $t :=$  NEXTT(nil,  $\mathcal{B}$ ,  $fv$ )
30       while  $t \neq$  nil and  $\rho' \neq$  true do
31          $\gamma :=$  "special simplification of  $c[t/x]$ "
32         if  $\gamma =$  true then
33            $\rho' :=$  true
34         elseif  $\gamma \neq$  false then
35            $\rho' :=$  "standard simplification of  $\rho' \vee \gamma$ "
36         end
37          $t :=$  NEXTT( $t$ ,  $\mathcal{B}$ ,  $fv$ )
38       od
39     end
40      $\rho :=$  "standard simplification of  $\rho \vee \rho'$ "
41     if  $\rho =$  true then
42        $\psi := \rho$ 
43       break
44     end
45   od
46    $\psi := \rho$ 
47 od
48 if " $Q_k$  is a block of universal quantifiers" then
49    $\psi :=$  " $\neg\psi$ , retransformed into an  $\wedge$ - $\vee$ -combination of base formulas"
50 end
51 if  $\psi =$  true or  $\psi =$  false then
52   return  $\psi$ 
53 end
54 od
55 return  $\psi$ 
56 end RQE

```

End of Algorithm 4.3.2

Theorem 4.3.2 (Regular Quantifier Elimination). *Let φ be a \mathcal{L}^* -formula. Algorithm 4.3.2 applied to φ terminates and transforms φ into an equivalent*

4 Implementation

quantifier-free formula ψ .

Proof. Termination: We state the termination of the procedure $\text{RQE}(\varphi)$ by considering each loop separately. At first, we state that the number of quantifier blocks, as well as the number of bound variables of each block of the input formula is finite, so that the **for**-loops of line 8 and 13 are finite. Furthermore, the **for**-loop of line 20 is finite, because if ψ is a disjunction, it only consists of finitely many disjuncts, and if ψ is a conjunction the code in lines 21 to 44 is run through exactly once for the current variable x . The **while**-loop of line 30 is also finite, because there are only finitely terms t with $\Delta(t) \leq \mathcal{B}$.

Correctness: First of all we state that the transformation of the input formula φ into φ' in line 3 can be easily done using the well-known transformation rules. If the matrix ψ of φ' is equal to *true* or *false*, then we can abort the elimination with the result ψ (see lines 5 to 7). The main part of the elimination is located within the **for**-loop starting in line 8. We prove the correctness of this loop by the following loop invariant: After the i -th pass of the loop, with $0 \leq i < b$, we have the following equivalence:

$$\varphi \leftrightarrow Q_b \dots Q_{i+1}(\psi),$$

where each Q_k is a block of existential or universal quantifiers.

We prove the loop invariant for $0 \leq i < b$: Before the first loop cycle, the equivalence trivially holds, because we defined φ' to be the prenex normal form of φ , and we defined ψ to be the matrix of φ' .

Let $\varphi \leftrightarrow Q_b \dots Q_i(\varphi)$, for $i < b$, before the i -th loop cycle. According to section 3.2.1, if Q_i is a block of universal quantifiers, we negate and retransform ψ into an \wedge - \vee -combination of base formulas in lines 10 to 12. We conclude this transformation in lines 48 to 50. Due to this transformation we can assume that each variable x in *bvl* is existentially bound within ψ . The elimination of a single variable x is located between line 14 and line 46. In lines 15 to 20, if ψ is a disjunction, we handle each disjunct c separately. If ψ is a conjunction, we proceed with the "single disjunct" $c \equiv \psi$. In line 21, we try to eliminate the current bound variable x within c using deep Gauss elimination (see section 4.3.2). If it was successful, we store the elimination result in line 23 and disjunctively combine it with previous elimination results for x in line 40. If deep Gauss elimination was not successful, we eliminate the current bound variable x by the elimination procedure introduced in section 3.2 (see Corollary 3.2.1). Therefor in line 25, we transform the current disjunct c into refined normal form, extract the list of free variables occurring in c (without x) in line 27, and estimate a depth bound \mathcal{B} in line 28. In lines 29 to 38 we finally get rid of x by substituting all terms t for x , with $\Delta(t) \leq \mathcal{B}$ according to Corollary 3.2.1. If these substitution results are not equal to *true* or *false*, we disjunctively combine the single results in line 35. If one of the substitution results equals *true* the **while**-loop is aborted (see line 33). The

results of eliminating x from each disjunct c are disjunctively combined in line 40, leading to the overall result of eliminating x . In line 46, we store this result in ψ . This way, we subsequently eliminate the bound variables of the current block Q_i . So after the i -th loop cycle we obtain $\varphi \leftrightarrow Q_b \dots Q_{i+1}(\psi)$. If ψ equals *true* or *false* after eliminating a block Q_i , then the **for**-loop of line 8 is aborted, and ψ is returned according to lines 51 to 53.

Before passing the loop the last time, we obtain $\varphi \leftrightarrow Q_b(\psi)$. Q_b is eliminated the same way as described above. So after the b -th loop cycle, ψ is quantifier-free and equivalent to φ . \square

Extended Quantifier Elimination

Corollary 3.2.2 states how we can extend the regular quantifier elimination procedure in order to get sample terms for an outermost block of existentially quantified variables. If there are further quantifier blocks inside, we first eliminate these quantifiers by the regular elimination procedure described above. So we may assume that all quantifiers other than those of an outermost block of existential quantifiers have already been eliminated. This leads to an input formula

$$\exists x_1 \dots \exists x_n (\varphi)(x_1, \dots, x_n, u_1, \dots, u_m).$$

Again, we eliminate the quantifiers one by one, starting with the innermost one. We obtain sample solutions for the remaining block by slightly modifying the regular elimination procedure of the previous section:

1. Consider the set Φ of base formulas of the remaining matrix φ and estimate a depth bound \mathcal{B} w.r.t. Φ . Consider the imaginary set $R_\Phi = \{t_0, \dots, t_r\}$ of \mathcal{L}^* -terms with $\Delta(R_\Phi) \leq \mathcal{B}$.
2. Consider the substitution results $\varphi'_{ji} \equiv \varphi[t_j/x_i]$ simplified by the special simplifier. If one of these results equals **true**, then the corresponding term t_j is a sample point for the current variable x_i . All further bound variables are then eliminated by arbitrary test terms. If none of the φ'_{ji} are equal to **true**, we have to store all pairs $\{\varphi'_{ji}, \{t_j\}\}$ where φ'_{ji} is not equal to **false**, separately.
3. As long as there are quantifiers left, we restart at step 1 with each of the pairs obtained in step 2 considering φ'_{ji} as the new matrix. If no more quantifiers are left, we return all pairs stored in step 2 as the final result.

With the approach described above, we can collect the sample solutions for the entire block of existential quantifiers.

Note that here, in contrast to regular quantifier elimination, we have to process intermediate substitution results separately. In other words, we must not build

4 Implementation

the disjunction over all these results as we did in regular quantifier elimination in order not to lose the substitution information.

The following algorithm describes in detail our procedure for extended quantifier elimination in expanded term algebras \mathbf{T}^* .

Algorithm 4.3.3 (Extended Quantifier Elimination).

Main procedure EQE

Input a prenex first-order \mathcal{L}^* -formula φ of the form $Q_b x_b \dots Q_1 x_1(\psi)$, where ψ is an \wedge - \vee -combination of base formulas and the Q_i are like quantifiers

Output a list of pairs (δ, τ) , where δ is a quantifier-free \mathcal{L}^* -formula and τ is a list of sample terms yielding δ , so that with the assignments for the bound variables stated in τ , $\delta \leftrightarrow \varphi$

```

1 procedure EQE( $\varphi$ )
2 begin
3   if  $\psi = \text{true}$  or  $\psi = \text{false}$  or  $b = 0$  then
4     return  $\psi$ 
5   end
6    $bvl :=$  "list of bound variables of  $\varphi$ "
7   if "the  $Q_i$  are universal quantifiers" then
8      $\psi :=$  " $\neg\psi$ , retransformed into an  $\wedge$ - $\vee$ -combination of base formulas"
9   end
10   $\rho :=$  EQE1( $bvl, \psi, \text{nil}$ )
11  if "the  $Q_i$  are universal quantifiers" then
12    for each answer in  $\rho$  do
13      "negate the elimination result of answer"
14    od
15  end
16  return " $\rho$ , transformed into a suitable form for output"
17 end EQE

```

Sub procedure EQE1

Input a list $bvl = \{x_1, \dots, x_n\}$ of existentially bound variables x_i , a quantifier-free formula ψ , and a list ansinfo containing pairs of the form (x_i, t) , where x_i is a variable and t is an \mathcal{L}^* -term

Output a list of pairs (δ, μ) , where δ is a quantifier-free \mathcal{L}^* -formula and μ is a list of sample terms yielding δ , so that with the assignments for the bound variables stated in μ , $\delta \leftrightarrow \exists x_n \dots \exists x_1(\psi)$

```

1  procedure EQE1(bvl,  $\psi$ , ansinfo)
2  begin
3     $x_i := \text{first}(bvl)$ 
4     $bvl := \text{rest}(bvl)$ 
5    if " $\psi$  is a disjunction" then
6       $l := \text{"list of disjuncts of } \psi \text{"}$ 
7    else
8       $l := \{\psi\}$ 
9    end
10   for each "subproblem  $c$  of  $l$ " do
11      $\psi_G := \text{"answer set of trying deep Gauss elimination for } c \text{ w.r.t. } x_i \text{"}$ 
12     if  $\psi_G \neq \text{"failed"}$  then
13        $\rho := \psi_G$ 
14     else
15        $c := \text{"} c \text{ exclusively containing inverse terms and constants"}$ 
16        $fv := \text{"list of free variables of } c \text{ without } x_i \text{"}$ 
17        $\mathcal{B} := \text{"depth bound estimated w.r.t. } \mathcal{L}, \text{eq}(c), \Delta(c) \text{ and } fv \text{"}$ 
18        $\gamma := \text{false}$ 
19        $t := \text{NEXTT}(\text{nil}, \mathcal{B}, fv)$ 
20       while  $t \neq \text{nil}$  and  $\gamma \neq \text{true}$  do
21          $\gamma := \text{"special simplification of } c[t/x_i] \text{"}$ 
22         if  $\gamma = \text{true}$  then
23            $\rho := (\gamma, \text{cons}((x_i, t), \text{ansinfo}))$ 
24         elsif  $\gamma \neq \text{false}$  then
25            $\rho := \text{cons}((\gamma, \text{cons}((x_i, t), \text{ansinfo})), \rho)$ 
26         end
27          $t := \text{NEXTT}(t, \mathcal{B}, fv)$ 
28       od
29     end
30     if  $\rho = \{(true, \{x_i, t'\})\}$  then
31        $\tau := \rho$ 
32       break
33     else
34        $\tau := \text{cons}(\rho, \tau)$ 
35     end
36   od
37    $\psi := \tau$ 
38   if  $bvl = \text{nil}$  then
39     return  $\psi$ 
40   else

```

4 Implementation

```

41     stop := false
42     while  $\psi \neq \text{nil}$  and not stop do
43         ans := first( $\psi$ )
44          $\psi := \text{rest}(\psi)$ 
45         tmp := EQE1(bvl, "first entry of ans", "second entry of ans")
46         if tmp  $\neq \{(true, eqs)\}$  then
47             stop := true
48             answerset := tmp
49         elsif tmp  $\neq \{(true, \text{nil})\}$  then
50             answerset := cons(tmp, answerset)
51         end
52     od
53 end
54 return answerset
55 end EQE1

```

End of Algorithm 4.3.3

Theorem 4.3.3 (Extended Quantifier Elimination). *Let φ be a \mathcal{L}^* -formula. Algorithm 4.3.3 applied to φ terminates and computes a list of quantifier-free equivalents of φ together with sample terms for an outermost existential quantifier block.*

Proof. **procedure** EQE1(*bvl*, ψ , *ansinfo*):

Termination: First, we state that the **for**-loop in line 10 is finite, because ψ can only contain finitely many disjuncts. The **while**-loop of line 20 is also finite, because there are only finitely many \mathcal{L}^* -terms t with $\Delta(t) \leq \mathcal{B}$. The **while**-loop of line 42 is finite, because we only obtain finitely many elimination answers from the elimination of one variable x in lines 15 to 28 or from deep Gauss elimination (see the following section 4.3.2) in line 11. We show the termination of the recursive call of EQE1 in line 45 by induction on the number n of variables in *bvl*. Let $n = 1$. The single variable x in *bvl* is eliminated either in line 11 or in lines 15 to 28. According to line 38 and 39, the procedure terminates, because we shortened *bvl* in line 4. Let $n > 1$. According to lines 41 to 52, after the elimination of the first variable in *bvl*, we get as many recursive calls of EQE1 in line 45 as there are answers in ψ , which are finitely many due to our depth bound estimation for test terms t . For each of these recursive calls, *bvl* contains $n - 1$ variables, because we shortened *bvl* in line 4. So these recursive call terminate by the induction hypothesis.

Correctness: We prove the correctness of EQE1(*bvl*, ψ , *ansinfo*) by induction on the number n of bound variables, i.e. the number of elements of *bvl*. Let $n = 1$. As in regular quantifier elimination, we handle each disjunct of ψ separately

according to lines 5 to 10. If deep Gauss elimination is possible for the current disjunct c , the procedure stores the result in line 13. If the result is of the form $\{\{true, \{(x, t')\}\}\}$, the procedure immediately returns the answer set according to lines 30 to 32 and line 39. Otherwise, the result for the current disjunct c is stored in line 34. If deep Gauss elimination fails, we first transform c into refined normal form in line 15, extract the list of free variables occurring in c without x_i in line 16, and estimate a depth bound \mathcal{B} in line 17. In lines 19 to 28, we eliminate x_i from c according to Corollary 3.2.2 of section 3.2.4 the following way: we successively substitute all terms t with $\Delta(t) \leq \mathcal{B}$ for x_i within c in line 21. If a single substitution result γ yields $true$ for an \mathcal{L}^* -term t , we abort the **while**-loop and store the result γ together with the substitution information (x_i, t) within a pair ρ in line 23. Additionally, in line 23, we add (x_i, t) to *ansinfo*, which is **nil** in the case of $n = 1$. If the substitution result $\gamma = c[t/x_i]$ in line 21 is not equal to *false*, we add the current result to the previous substitution results for x_i and also add (x_i, t) to *ansinfo* in line 25. So after leaving the **while**-loop in line 30, we correctly eliminated x_i from c according to Corollary 3.2.2. The result ρ of this elimination either stores a single pair $(true, \{(x_i, t_i)\})$ or finitely many pairs $(\psi_k, \{(x_i, t_k)\}), \dots, (\psi_1, \{(x_i, t_1)\})$, where the ψ_i are the single substitution results when substituting t_i for x_i . In lines 30 to 35, we store the single elimination results for disjuncts c within τ . According to line 4, *bvl* is now empty, so that ψ is returned in line 39.

Let $n > 1$. We eliminate the current bound variable x_i as described in the case $n = 1$. This leads to an answer set $\psi := \{(\psi_1, \{(x_i, t_1)\}), \dots, (\psi_k, \{(x_i, t_k)\})\}$ in line 37. In contrast to the case $n = 1$, we proceed with ψ in the **else**-case starting in line 41. According to line 42, for each pair (ψ_i, ans) of ψ , we get a recursive call $\text{EQE1}(bvl, \psi_i, ans)$. Due to the fact that *bvl* was shortened in line 4, we can apply the induction hypothesis to these calls of EQE1 . If one of these calls leads to a result $tmp = \{\{true, eqs\}\}$, where *eqs* is now of the form $\{(x_i, t_i), \dots, (x_1, t_1)\}$, we can abort the **while**-loop and immediately return *tmp* according to lines 46 to 48, and finally line 54. Otherwise, we proceed as usual, and add the current result *tmp* to *answerset* in line 50.

procedure $\text{EQE}(\varphi)$:

Termination: First, we state that the **for**-loop of line 12 is finite, because the procedure EQE1 yields a list with only finitely many answers. The termination of EQE results from the termination of EQE1 .

Correctness: In lines 3 to 5, we immediately return ψ as the elimination result, if ϕ is either *true*, *false* or the quantifier block of φ is empty. In lines 7 to 9, if the quantifiers in φ are universal quantifiers, we proceed with $\neg\psi$ and retransform it into an \wedge - \vee -combination of base formulas. This transformation is concluded in lines 11 to 15. According to line 10, the correctness of EQE results from the

4 Implementation

correctness of the procedure EQE1. \square

In the next section, we describe an elimination method for special kinds of input formulas. It also uses the test term approach, but if applicable, the sizes of the sets R_Φ of test terms will be much smaller.

4.3.2 Deep Gauss Elimination

The notion of *Deep Gauss Elimination* was originally introduced for the reals in [Dol00]. The main focus of the procedure is to get smaller elimination sets for eliminating the quantifier from formulas $\exists x(\psi)(x, u_1, \dots, u_n)$. This goal is achieved by extending the trivial *Gauss elimination* idea introduced for the reals in [LW93]. There, the idea is to restrict the elimination set for formulas $\exists x(x + a = 0 \wedge \psi)$ to the singleton $\{-a\}$, obtained from the equation $x + a = 0$. A set $\mathcal{S}^x(\psi)$ is called *solution set* for x of the formula $\psi(x, u_1, \dots, u_n)$, if $\mathcal{S}^x(\psi) = \{t \in A \mid \mathbf{A} \models \psi(t, a_1, \dots, a_n)\}$, for a certain structure \mathbf{A} and fixed $a_1, \dots, a_n \in A$. In the reals, a single equation which occurs conjunctively on the top-level of the considered formula and which additionally has a finite *solution set* for any admissible assignment of its parameters, is called a *Gauss equation*. The idea of *deep Gauss elimination* is to extend the procedure to recognize *Gauss formulas*.

In our context of expanded term algebras, an equation is a Gauss equation w.r.t. a certain variable, if its solution set² is a singleton. In other words, in our context a Gauss equation w.r.t. a certain variable x is of the form $x = t$, where t is an \mathcal{L}^* -term which does not contain x . The following recursive definition shows how to recognize a formula to be a *Gauss formula* within the context \mathbf{T}^* :

- Each Gauss equation is a Gauss formula. The obtained elimination set consists of the single solution w.r.t. the corresponding variable.
- Let φ_1 be a Gauss formula and let E_1 be its elimination set. Then $\varphi_1 \wedge \dots \wedge \varphi_n$ is also a Gauss formula with elimination set E_1 .
- Let $\varphi_1, \dots, \varphi_n$ be Gauss formulas and let E_1, \dots, E_n be the corresponding elimination sets. Then $\varphi_1 \vee \dots \vee \varphi_n$ is also a Gauss formula with elimination set $E_1 \cup \dots \cup E_n$.

Note that the above definition of Gauss formulas actually extends the applicability of trivial Gauss elimination as we are not restricted to the special input formulas

²In the reals, solution sets and elimination sets differ in two ways: first, in most cases solution sets are infinite, whereas elimination sets are finite by definition. Second, elimination sets contain pairs (ρ, t) , where ρ guarantees the existence of the term t in the reals, whereas solution sets contain single terms by definition. In \mathbf{T}^* we do not need to explicitly distinguish between these two terms.

mentioned in the preface of this section. The following example illustrates the situation:

Example 4.3.3 (Deep Gauss Elimination(1)). Consider the input formula

$$\Phi \equiv \exists x((x = u \vee x = v) \wedge \psi(x, u_1, \dots, u_n)).$$

Then both $x = u$ and $x = v$ are Gauss equations so that the disjunction is a Gauss formula and hence also Φ , no matter what hides behind $\psi(x, u_1, \dots, u_n)$. As a consequence $\{u, v\}$ is an elimination set for Φ and x . Without deep Gauss elimination, the question whether we can expect an elimination result would depend on the underlying language and the complexity of $\psi(x, u_1, \dots, u_n)$. \diamond

Note that when we talk about finite solution sets for equations in \mathbf{T}^* , we strictly talk about singletons. As a result of the depth bound estimation, each base formula has a finite solution set. The problem is that for an arbitrary equation it is still hard to compute this set as a whole. Consider e.g. the formula $inv_{f,1}(inv_{f,1}(x)) = inv_{f,1}(x)$ with $\mathcal{L} = \{a^{(0)}, b^{(0)}, f^{(1)}, g^{(2)}\}$ as the underlying language. The equation is solved by all constants, all terms starting with g and also some terms starting with f . As in the standard elimination procedure, we would still have trouble with the time and space required to compute and store all these terms. In addition to that problem, it would even be hard to enumerate such an "irregular" set of terms. To put it all in a nutshell, we only consider equations of the form $x = t$ as Gauss equations, where x is the currently considered bound variable and t is an arbitrary \mathcal{L}^* -term which does not contain x .

Note that we have to check both representations of an input formula, before we can characterize it as a Gauss formula. The following example demonstrates the situation:

Example 4.3.4 (Deep Gauss Elimination(2)). Consider the input formula

$$\Phi \equiv \exists x(x = f(y)).$$

Then Φ is a Gauss formula w.r.t. x , whereas the refined normal form Φ' of Φ is not:

$$\Phi' \equiv \exists x(inv_{f,1}(x) \neq x \wedge inv_{f,1}(x) = y).$$

The same holds for the converse: Consider the input formula

$$\Phi \equiv \exists x(y = f(x)).$$

Then Φ isn't a Gauss formula w.r.t. x , whereas the refined normal form Φ' of Φ is:

$$\Phi' \equiv \exists x(inv_{f,1}(y) \neq y \wedge inv_{f,1}(y) = x). \quad \diamond$$

4 Implementation

In sections 4.4 and 5, we will see the great benefit of deep Gauss elimination over standard quantifier elimination.

The next section focusses on the idea to exploit the freedom to permute quantifiers within a block of like quantifiers.

4.3.3 Strategies for Quantifier Permutations

Within a block of like quantifiers, we can make use of the equivalence

$$\exists x_i \exists x_j \longleftrightarrow \exists x_j \exists x_i.$$

The quantifiers within a block of like quantifiers are sorted according to the following criteria:

- The first criteria states whether deep Gauss elimination is applicable for a certain variable x_k .
- If the first criteria isn't met, the second one is considered. It focusses on the number of occurrences the bound variables have in the given matrix and favours higher numbers over lower ones.

The idea behind these criteria is that, on the one hand, the possibility to apply deep Gauss elimination to a certain quantifier should always be exploited. On the other hand, if deep Gauss elimination isn't applicable, we favour a higher number of occurrences of the bound variable over a lower one. This way, we can exhaust the action of the special simplifier, which is responsible for simplifying the single substitution results (see Algorithms 4.3.2 and 4.3.3).

The following example shows how these criteria lead to a rearrangement of the quantifiers within a block of like quantifiers:

Example 4.3.5 (Quantifier Permutations). *Consider the input formula*

$$\exists x \exists y \exists z (y = a \wedge z \neq a \wedge x \neq u \wedge z \neq u).$$

According to our permutation criteria, the quantifiers are rearranged to

$$\exists x \exists z \exists y (y = a \wedge z \neq a \wedge x \neq u \wedge z \neq u).$$

The first quantifier we eliminate is $\exists y$, because this can be done by deep Gauss elimination. Finally, the second criteria favours $\exists z$ over $\exists x$ because of the higher number of occurrences of z within the matrix of the input formula. \diamond

The following section illustrates both quantifier elimination procedures by means of an example computed by hand. There, we will also see the benefit of quantifier permutations when performed before the elimination process starts.

4.4 Illustration

In this section, we demonstrate the overall process of quantifier elimination in expanded term algebras \mathbf{T}^* . In the following, we describe the internal functionality of regular and extended quantifier elimination in \mathbf{T}^* with a simple example computed by hand.

Let $\mathcal{L}^* = \{a^{(0)}, f^{(1)}, \text{inv}_{f,1}^{(1)}\}$ be the underlying expanded language. Consider the input formula

$$\exists x \exists y (x = u \wedge y \neq x).$$

First of all, we permute the quantifiers according to the criteria introduced in the last section³. This leads to the following updated input formula

$$\exists y \exists x (x = u \wedge y \neq x).$$

We eliminate the quantifiers one by one starting with the innermost. For eliminating $\exists x$, we can use the deep Gauss elimination procedure. Applying the procedure leads to the following intermediate result

$$\exists y (y \neq u).$$

For eliminating the last quantifier, we have to apply the standard quantifier elimination procedure. Therefore, we estimate a depth bound \mathcal{B} and successively substitute all terms t_i with $\Delta(t_i) \leq \mathcal{B}$. All these singular results are combined disjunctively leading to

$$\bigvee_{t_i \in R_\Phi} (y \neq u)[t_i/y].$$

Note that according to section 4.3.1, we will successively substitute the terms $a, u, f(a), f(u), \dots$ generated by our enumeration procedure. This is done until the special simplification of the current substitution result or the standard simplification of the disjunction yields **true**. As you can see, when substituting $f(u)$ for y , we get **true** as the final result of the elimination.

We can use the same example to describe the process of extended quantifier elimination in \mathbf{T}^* . As in regular elimination, we first apply the permutation strategies, leading to

$$\exists y \exists x (x = u \wedge y \neq x).$$

As usual, we start with the elimination of $\exists x$ and apply the deep Gauss elimination procedure. Of course, we arrive at the same residue formula, but additionally, we keep in mind the single substitution information obtained from eliminating $\exists x$:

$$x = u : \exists y (y \neq u).$$

³Note that if we eliminated the quantifiers in the given order, we actually would have to substitute all terms of the elimination set for y enumerated w.r.t. \mathcal{L}^* and the input formula, because none of the single substitution results would cause a premature stop.

4 Implementation

For eliminating the last quantifier, we apply the standard quantifier elimination procedure. Unlike in regular quantifier elimination, we have to handle each of the singular results separately, until one of them yields **true**. This is the case when substituting $f(u)$ for y , leading to the final result

$$x = u, y = f(u) : \mathbf{true}.$$

Note that due to the possibility to apply deep Gauss elimination, the step between eliminating $\exists x$ and $\exists y$ looks harmless. The following even simpler example demonstrates the general case. Consider the input formula

$$\exists x(x \neq u \wedge x \neq v).$$

Here, we cannot apply deep Gauss elimination, so we proceed with the standard elimination procedure. We get the following singular results together with the corresponding substitution information:

$$\begin{array}{ll} x = a: & u \neq a \wedge v \neq a \\ x = f(a): & u \neq f(a) \wedge v \neq f(a) \\ x = f(u): & v \neq f(u) \\ \vdots & \vdots \\ x = \text{inv}_{f,1}(\text{inv}_{f,1}(v)): & u \neq \text{inv}_{f,1}(\text{inv}_{f,1}(v)) \wedge v \neq a \end{array}$$

In the case of further quantifiers, one has to continue the procedure for each of these singular results separately.

In the next section, we consider some application examples of quantifier elimination in \mathbf{T}^* .

5 Application Examples

In this chapter, we consider mathematical structures which can be modelled by term algebras. We try to prove some well-known properties of these structures by quantifier elimination. Additionally, we consider some parametric examples from the area of logic programming. At the end of this chapter, the single regular quantifier elimination results of the following examples are summarized.

5.1 Natural Numbers

We can model the structure of natural numbers $\mathbf{N} = \{\mathbb{N}, 0, succ\}$ as a term algebra by considering the language $\mathcal{L} = \{o^{(0)}, s^{(0)}\}$, including a single constant o (zero) and a single unary function symbol s modelling the successor function.

Now we can prove that "every natural number has a successor". Therefore, we first have to transform this statement into the corresponding first-order formula to which we afterwards apply the elimination procedure:

$$\forall x \exists y (y = s(x)) \quad (\text{NN1})$$

The regular quantifier elimination procedure, as well as the extended quantifier elimination procedure applied to example NN1 yield **true** after 0 s¹. We can refine the last statement and prove that "the successor of each natural number is unique":

$$\forall x ((y = s(x) \wedge z = s(x)) \Rightarrow z = y) \quad (\text{NN2})$$

Obviously, this should result in **true**. The elimination procedure yields $inv_{s,1}(y) \neq inv_{s,1}(z) \vee y = o \vee z = o \vee z = y$ after 0 s. On closer examination, we can see that actually the resulting formula is equivalent to **true**. This again indicates the necessity of more sophisticated simplification strategies, especially for base formulas like the first one of the elimination result above.

Next, we try to prove that "every natural number is a successor", which certainly does not apply to the constant o :

$$\forall x \exists y (x = s(y)) \quad (\text{NN3})$$

The elimination procedure yields **false** after 0 s. After the same time, the extended quantifier elimination procedure applied to NN2 yields $\{\mathbf{false}, x = o\}$.

¹This means that the computation time is smaller than the smallest measurable time of 10 ms.

5.2 Binary Trees

Similarly to the last section, we can model binary trees $\mathbf{T} = \{T, \text{root}, \text{left}, \text{right}\}$ as a term algebra by considering the language $\mathcal{L} = \{o^{(0)}, l^{(1)}, r^{(1)}\}$, including a single constant o (the root of the tree) and two unary function symbols l and r modelling the left and right subtrees.

Now we can prove that "every binary tree has a left and a right subtree". The translation of this statement leads to the following first-order formula:

$$\forall x \exists y \exists z (y = l(x) \wedge z = r(x)) \quad (\text{BT1})$$

The regular quantifier elimination procedure, as well as the extended quantifier elimination procedure applied to BT1 yield **true** after 0.1 s. Similar to the structure of natural numbers, we can prove the uniqueness of left and right subtrees by considering the formula

$$\forall x \left((u = l(x) \wedge v = l(x) \wedge y = r(x) \wedge z = r(x)) \Rightarrow (u = v \wedge y = z) \right) \quad (\text{BT2})$$

The regular elimination procedure yields a quantifier free equivalent containing 9 base formulas after 0 s. As in the corresponding example of the last section, this result turns out to be equivalent to **true**. We slightly modify example BT1 and try to prove that "every binary tree is a left or a right son":

$$\forall x \exists y \exists z (x = l(y) \vee x = r(z)) \quad (\text{BT3})$$

The regular quantifier elimination procedure yields **false** after 0 s. After the same time, the extended quantifier elimination procedure applied to BT2 yields $\{\mathbf{false}, x = o\}$.

Next, we have a look at some general parametric problems as they could appear in the area of logic programming. We will see that parametric problems are well suited to point out the benefits of quantifier permutations and especially of deep Gauss elimination.

5.3 Parametric Problems

Let $\mathcal{L} = \{a^{(0)}, f^{(1)}, g^{(2)}\}$ be the underlying language. We consider the input formula

$$\exists x \exists y (g(x, f(y)) = g(u, x)) \quad (\text{PP1})$$

We want to find out the conditions for the free variable u under which there exist corresponding terms x and y , via quantifier elimination. The regular quantifier elimination procedure yields the condition $inv_{f,1}(u) \neq u$ in 0 s due to deep Gauss

elimination. After the same time, the extended quantifier elimination procedure additionally yields for x and y the corresponding sample points. As you can see in the table at the end of this chapter, we wouldn't get any result without the help of deep Gauss elimination.

Note that internally, the previous input formula is immediately transformed into $\exists x \exists y (x = u \wedge x = f(y))$ due to standard simplification. So we can try to solve the same problem with $\mathcal{L} = \{a^{(0)}, f^{(1)}\}$ as the underlying language:

$$\exists x \exists y (x = u \wedge x = f(y)) \quad (\text{PP2})$$

This time we get $u \neq a$ as the necessary condition for u after 0 s. The crucial point is that with the new underlying language we do not depend on deep Gauss elimination as you can see in the summarizing table at the end of this chapter.

We finish this section with another simple parametric example. Let $\mathcal{L} = \{a^{(0)}, f^{(1)}, g^{(2)}\}$ be our underlying language. Consider the following input formula:

$$\exists x \exists y \exists z (x = u \wedge y = a \wedge z \neq x) \quad (\text{PP3})$$

The regular quantifier elimination procedure yields **true** after 0 s, due to deep Gauss elimination and the quantifier permutation strategies. Additionally, the extended quantifier elimination procedure yields the substitution information $\{x = u, y = a, z = f(u)\}$ after the same time. As you can see in the summarizing table at the end of this section, applying the permutation strategies is fundamental for getting an elimination result for PP3 at all.

The following table summarizes the regular quantifier elimination results of the previous examples in terms of the required elimination time² and the output length³. In addition, the results are opposed according to the state of the switches **TALPQEGAUSS** (deep Gauss elimination) and **TALPQP** (quantifier permutation strategies). If **TALPQEGAUSS** is off, we do not make use of deep Gauss elimination even if it would be possible. If **TALPQP** is off, we eliminate the quantifiers of the input formula in the given order. An entry " ∞ " means that the user does not get any information on the elimination result. This could happen because either the elimination procedure does not yield a result within a reasonable amount of time or **REDUCE** runs out of memory.

²measured in seconds (rounded)

³measured in the number of base formulas

5 Application Examples

| Regular Quantifier Elimination Examples | | | | | | | | |
|---|----------------|------------|-----------------|------------|---------------------|------------|-----------------|------------|
| Name | Runtime [s] | | | | Output length [#eq] | | | |
| | TALPQEGAUSS on | | TALPQEGAUSS off | | TALPQEGAUSS on | | TALPQEGAUSS off | |
| | TALPQP on | TALPQP off | TALPQP on | TALPQP off | TALPQP on | TALPQP off | TALPQP on | TALPQP off |
| NN1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| NN2 | 0.1 | 0.1 | 0.5 | 0.5 | 4 | 4 | 504 | 504 |
| NN3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| BT1 | 0 | 0 | ∞ | ∞ | 0 | 0 | ∞ | ∞ |
| BT2 | 0.2 | 0.2 | 159 | 159 | 9 | 9 | 18610 | 18610 |
| BT3 | 0 | 0 | 1.7 | 1.7 | 0 | 0 | 0 | 0 |
| PP1 | 0 | 0 | ∞ | ∞ | 1 | 1 | ∞ | ∞ |
| PP2 | 0 | 0 | 2.4 | 6.1 | 1 | 1 | 1 | 1 |
| PP3 | 0 | 32.8 | 11.2 | ∞ | 0 | 0 | 0 | ∞ |

6 Summary

In this thesis, we have concerned ourselves with quantifier elimination in term algebras \mathbf{T} as introduced in [SW02].

After an introduction chapter giving an overview over the contents of this thesis, we have started in chapter 2 with a brief introduction of the fundamental mathematical framework needed throughout the following sections. In detail, we have given a short outline of the first-order theory of term algebras and introduced the central notion of quantifier elimination. We have presented the necessary structural expansion in order to make quantifier elimination generally possible. In contrast to other reports, the expansion used here is kept purely functional so that the underlying basic structure could be maintained.

In chapter 3, we have focussed on quantifier elimination in expanded term algebras \mathbf{T}^* and the concerning framework, based on [SW02]. In detail, we have introduced normal forms for terms and equations according to the functional expansion. Furthermore, we have given an outline of the elimination technique that we have chosen to use for quantifier elimination in \mathbf{T}^* . One crucial point for the applicability of this technique was the previously introduced normal form for equations. This normal form enabled us to estimate depth bounds for test terms, which in turn lead to the necessary elimination sets. These sets, together with the fundamental theorem for quantifier elimination in \mathbf{T}^* , ensured the applicability of the chosen method. We closed this chapter with detailed complexity estimations of the quantifier elimination and decision problem for \mathbf{T}^* .

In chapter 4, we have concerned ourselves with the implementation part of regular and extended quantifier elimination in \mathbf{T}^* . This chapter has started with a brief description of how the term algebra package is embedded in the logic framework REDLOG. Within this framework all procedures have been implemented as a new context. In the following subsection, we have focussed on the simplification part of the implementation. This area comprised the standard simplifications partially provided generically by REDLOG, as well as \mathbf{T}^* -specific special simplifications. In the main part of this chapter, we have dealt with the implemented quantifier elimination procedures. Additionally to the elimination procedure introduced in [SW02], we have described another approach applicable for special kinds of input formulas. We have illustrated the interaction of the simplifiers and the elimination procedures. In addition, we have mentioned the benefit of making use of quantifier permutations within blocks of like quantifiers. We have closed

6 Summary

this chapter with an illustrating example for the overall process of regular and extended quantifier elimination in expanded term algebras \mathbf{T}^* .

In chapter 5, we have examined some application examples. There, we have started to prove some well-known properties of mathematical structures via quantifier elimination. In addition, we have dealt with some parametric examples from the area of logic programming. These examples have confirmed the enormous complexity of quantifier elimination in \mathbf{T}^* when function symbols of arity greater than one are present.

Bibliography

- [CL89] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.
- [Dol00] A. Dolzmann. *Algorithmic Strategies for Applicable Real Quantifier Elimination*. Doctoral Dissertation at the University of Passau, 2000.
- [Dol02] A. Dolzmann. *Anwendungen der Computerlogik*. Lecture at the University of Passau, summer term 2002.
- [DS97a] A. Dolzmann and Th. Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.
- [DS97b] A. Dolzmann and Th. Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209–231, 1997.
- [Gri99] E. R. Griffor. *Handbook of Computability Theory*. Elsevier, Amsterdam, 1999.
- [Hod93] W. Hodges. *Model Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1993.
- [LW93] R. Loos and V. Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993.
- [RV01] T. Rybina and A. Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*, 2(2):155–181, 2001.
- [Sch74] C. P. Schnorr. *Rekursive Funktionen und ihre Komplexität*. Teubner, 1974.
- [Stu02] Th. Sturm. Quantifier elimination-based constraint logic programming. *Technical Report MIP-0202, FMI, University of Passau, Germany*, January 2002.

Bibliography

- [SW02] Th. Sturm and V. Weispfenning. Quantifier elimination in term algebras - the case of finite languages. *Proceedings of the Fifth International Workshop on CASC*, pages 285–300, 2002.
- [Wei88] V. Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1-2):3–27, 1988.
- [Wei03] V. Weispfenning. *Eliminations-Algorithmen mit Anwendungen*. Lecture at the University of Passau, summer term 2003.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Inhalte, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Diese Arbeit habe ich in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, März 2005

Christian Hoffelner