FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Algorithms for Quantified Cut-Introduction

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Christoph W. Spörk, BSc
Matrikelnummer 0727500

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ. Prof. Dr.phil. Alexander Leitsch
Mitwirkung: Dr. Giselle Reis
Dr. Sebastian Eberhard

Wien, 20. April 2015

(Unterschrift Verfasser/in) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Algorithms for Quantified Cut-Introduction

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computational Intelligence

by

Christoph W. Spörk, BSc
Registration Number 0727500

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ. Prof. Dr.phil. Alexander Leitsch
Assistance: Dr. Giselle Reis
Dr. Sebastian Eberhard

Vienna, 20. April 2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Christoph W. Spörk, BSc
Dobrowskygasse 4, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser/in)

Danksagung

Ich möchte an dieser Stelle Univ.-Prof. Dr.phil. Alexander Leitsch aufrichtig danken, dass er mir dieses Thema angeboten und mir notwendige Räumlichkeiten, Kontakte und Informationen für diese Arbeit zuteil werden ließ. Besonders danken möchte ich auch Dr. Giselle Reis für ihre Geduld und Unterstützung während der gesamten Entstehung dieser Arbeit. Ebenso möchte ich Dr. Sebastian Eberhard meinen Dank aussprechen für seine, für diese Arbeit notwendige, theoretische Vorarbeit.

Den Rest dieser Seite widme ich meiner Familie und meinen Freunden, welche mich in den schwierigsten Tagen meines Lebens unterstützt und mir wesentliche Hilfe angeboten haben. Ohne sie wäre ich nicht so weit gekommen. Außerdem will ich Thomas Mosandl danken, da er mir eine große Stütze war und mir and den dunkelsten Tagen wieder auf die Beine half.

In stillem Gedenken an Ursula Spörk †
Danke für alles.

Acknowledgements

I wish to express my sincere thanks to Univ.-Prof. Dr.phil. Alexander Leitsch, for offering me this topic and providing me with all necessary facilities and connections for the research. A special thank goes to Dr. Giselle Reis for her patience and support throughout the whole process of this thesis. I also want to thank Dr. Sebastian Eberhard for his groundwork, which was necessary for this thesis.

I want to dedicate the rest of this page to my family and all of my friends, which supported me and provided crucial help during the hardest days of my life. Without them I would not have come so far. Additionally I want to thank Thomas Mosandl, who not just endorsed me, but helped me stand up in my darkest days.

In loving memory of Ursula Spörk †
Thank you for everything.

Abstract

The possibility of quantified cut-introduction, which this thesis is dedicated to, offers the possiblity of compressing a proof in sequent calculus, to improve readability and provide interesting new insights into proof theory. The problem of introducing quantified cuts into a cut-free proof in sequent calculus is a quite complex procedure, which can be accomplished in various ways. The method described in this thesis realised this by applying several consecutive steps. First essential parts of the proof are extracted, namely the Herbrand sequent, from which a term language is generated. This term language represents all needed instantiations of quantified formulas and will be tried to be compressed by a minimal grammar. The problem of finding a minimal grammar for this language is achieved by reducing it to the MinCostSAT problem and thus generate it via the resulting interpretation. This grammar forms the base for the further construction of an extended Herbrand sequent, which again illustrates an intermediate step to build a proof with cuts. On the basis of [3] an algorithm was constructed performing this minimal grammar computation, which was tested via a large series of experiments. Although previously developed methods, capable of cut-introduction, were not outperformed by this new method, the methods were extended by the possibility of introducing not just one single-quantified cut, but multiple single-quantified cuts at once. Possible new insights may be gained by this implemented method, where simultaneously a small step towards cut-introduction for cuts with more complex cut-formulas was made.

Kurzfassung

Diese Arbeit beschäftigt sich mit einer Methode zur Einführung von Schnitten in Beweise des Sequential-Kalküls und bietet damit Möglichkeit zur Kompression der Beweise, dadurch eine eventuelle Verbesserung der Lesbarkeit und impliziert möglicherweise neue Einblicke in die Beweistheorie Das Problem der Schnitteinführung in schnittfreie Beweise des Sequential-Kalküls ist recht komplex, kann bereits durch verschiedene Methoden durchgeführt werden. Die, in dieser Arbeit beschriebene, Methode realisiert die Kompression mittels einiger aufeinanderfolgenden Schritte. Zuerst werden essentielle Teile des Beweises extrahiert, welche auch als Herbrand Sequent bezeichnet werden, und in eine Termsprache umgewandelt. Diese Sprache repräsentiert alle notwendigen Instanzen der quantifizierten Formeln. Es wird weiters versucht jene durch eine minimale Grammatik zu komprimieren. Das Problem eine minimale Grammatik für eine gegebene Sprache zu finden wird auf das MinCostSAT Problem reduziert und aus einer eventuell resultierenden Interpretation generiert. Diese Grammatik stellt das Fundament für die weitere Konstruktion eines erweiterten Herbrand Sequents dar, welches ein notwendiger Zwischenschritt ist um schlussendlich einen Beweis mit eingeführten Schnitten zu generieren. Auf Basis von [3] wurde ein Algorithmus zur Beweiskomprimierung konstruiert, welcher durch eine umfangreiche Serie an Experimenten getestet wurde. Obwohl die Effizienz bereits existierender Methoden zur Schnitteinführung in Beweise des Sequentialkalküls nicht übertroffen wurden, bietet diese neue Methode eine Erweiterung zu der bestehenden Funktionalität, durch die Möglichkeit mehr als einen einfach quantifizierten Schnitt auf einmal einzuführen. Während ein kleiner Schritt in der Richtung der Schnitteinführung gemacht werden konnte, welche in Zukunft darauf abzielen wird Schnitte mit immer komplexeren Schnittformeln einzuführen, könnte diese neue Methode ebenso neue Einblicke in die Beweistheorie geben.

Contents

CHAPTER 1

Introduction

1.1 Motivation and aim of the work

In Mathematics, as well as in Computer Science and many other scientific disciplines, it is of prior importance to provide proofs for one's theorems. In times of advanced technological progress, we are living nowadays, one can make good use of automated provers to get reasonable proofs for theorems. These proofs can, depending on the complexity of the theorems, reach a quite enormous size and are therefore hard to understand. It is a common didactic practice to introduce lemmas on the way to prove a particular complex theorem. This technique can also be used in a particular calculus, namely Gentzen's *sequent calculus* for *classical first order logic* (*FOL*) [4]. By introducing lemmas, or *cuts* as they are called in sequent calculus, one can gain knowledge of how a proof works and what exactly is needed to prove a theorem. Additionally the proof potentially becomes compressed, s.t. the readability and thus the possible understanding of the proof is improved.

The task of introducing cuts into sequent calculus proofs is not new territory. There are various works related to this topic such as [22] which describes a method based on *resolution* to introduce *atomic cuts*, i.e. cuts with an atomic interpolant or lemma. A cut-formula (i.e. the interpolant which represents the lemma in sequent calculus) can furthermore appear quantified, whereas the underlying cut is then called a quantified cut. This kind of cuts have to be treated in a different way, whereas different layers of complexity arise for the introduction of quantified cuts when considering quantifications of higher complexity of a formula. Those layers appear when considering not just purely universal or existential cut-formulas, but also formulas with alternating quantifiers. A proper subclass of quantified cuts are for instance Σ_1 and Π_1 cuts, which are called *universal cuts* and are characterized by their cut-formulas, which do not contain quantifier alternations. In [9] a method is described which is able to introduce universal cuts with blocks of quantifiers, which got extended by the capability of handling formulas in *first-order equational logic* through *paramodulation* in [8].

To accomplish the introduction of this universal cuts into a sequent calculus proof, many different intermediate steps have to be made. First considering a proof in sequent calculus with-

out introduced lemmas as a whole is not a suitable starting point, due to its possible enormous size and variance. Therefore we make use of a method to extract certain important notions out of a proof, i.e. *substitutions* generating a *Herbrand sequent*, which is a propositional valid representation of the proof, and proceed by handling only those. The substitutions are further interpreted as a set of *terms* to finally reduce the problem of introducing universal cuts to the problem of finding a compressed representation of this *term* set. This compressed representation can be computed in several ways, whereas one method capable of eventually introducing a single universal cut with a block of quantifiers into a sequent calculus proof in predicate logic with equality at once was described in [8]. A more promising one (described in [3]) was implemented in the course of this thesis, which is able to introduce several universal cuts with one quantifier at once, by reducing the problem of compressing a set of terms to a *SAT* problem. Unfortunately this method is at the moment only capable of handling a small subset of Σ_1 and Π_1 cuts, i.e. those with single-quantified cut-formulas. The advantage of this new procedure is a possible improvement in efficiency and runtime. This thesis focuses on the implementation and testing of the latter algorithm for introducing several universal cuts (with single-quantified cut-formulas) into a given proof in sequent calculus by using knowledge from formal language theory.

The *Theory and Logic Group* at the *Technical University of Vienna* developed a proof system, called *GAPT* [20], which is capable of a variety of functionalities. Besides the possibility of formulating sentences and proofs in FOL and *Higher Order Logic*, the functionality of the system comprises methods of cut-elimination [19] [2] and previously mentioned methods for introducing a single universal cut into a proof in sequent calculus. It is even possible to access proofs found in the *TPTP* [18], a library of proofs created for *automated theorem proving*. To understand the underlying proof-theoretic setting the cut-introduction method in [9] had to be considered. The former compression method in *GAPT*, described in [8] had to be considered as well as the new compression method [3], which was implemented in the course of this thesis.

Since the other cut-introduction method was already implemented, exhaustive testing and benchmarking was performed to compare the new method with the old one. For this purpose various example proofs were generated from scratch or taken from the TPTP library as a test basis for the method.

1.2 Structure of the work

We will first fix notations and describe certain definitions in Chapter 2, which are necessary for the further chapters. In Chapter 3 we will discuss the current state of the art consisting of existing literature, which will be of high importance, since the thesis will rest upon this work. An overview of current techniques of cut-introduction, as well as already implemented solutions will be provided. We will get to the main part of the thesis in chapter 4, were the implementation will be discussed. Starting from a set of terms we will show how we construct a compressed representation according to [3] in detail. Subsequently the algorithm for compressing a set of terms will be embedded into the existing environment of cut-introduction in *GAPT*. Problems encountered during the implementation and their solutions will also be a part of the explanation. Chapter 5 will provide a critical reflection and a comparison between related methods and the

accomplished work of this thesis. Open issues will also find their place in this chapter of the thesis. Finally a summary will be given in Chapter 6, which will also address starting points for future work in this area.

CHAPTER 2

Preliminaries

Consider the following definitions, which will help us forming a base for the methods of cut-introduction to be discussed. We will operate mainly in the language of *First Order Logic* unless indicated otherwise. Let us recall the signature of FOL, namely

- function symbols $f \in F^n$, where $n \geq 0$
- free and bound variables V_f (notation u, v) and V_b (notation x, y), respectively
- logical connectives $\wedge, \vee, \rightarrow, \neg$
- quantifiers $\forall, \exists$
- auxiliary symbols $(\cdot)\cdot$,
- atoms $P, Q, \ldots$

2.1 Terms and formulas

We start by defining *terms* in FOL in Definition 1.

Definition 1 (term, semi-term)
We define the set of semi-terms inductively:

- *bound and free variables are semi-terms*
- *constants are semi-terms*
- *if $t_1, \ldots, t_n$ are semi-terms and f is an n-place function symbol then $f(t_1, \ldots, t_n)$ is a semi-term.*

Semi-terms which do not contain bound variables are called terms

As we will need a way to determine certain positions of subterms within a term we define them in Definition 2.

Definition 2 (positions of terms)
Let t be a term, then ϵ is the position of t. Assume p to be the position of a term $f(t_1, \ldots, t_n)$ in t, then, for every $i \in \{1, \ldots, n\}$, $p.i$ is the position of t_i in t.
e.g. If $t = f(f(a, b), f(c, d))$, then $\epsilon.2.1 (= 2.1)$ is the position of c in t.
We write $t|_p$ to denote a subterm in t at position p.

For an example of terms at positions within a term consider Example 1.

Example 1 (Example for term positions)
Let $t = f(x, g(a, b))$ be a term, then
$t|_\epsilon = f(x, g(a, b))$ *is the term at position ϵ in t,*
$t|_1 = x$ *is the term at position* 1,
$t|_2 = g(a, b)$ *is the term at position* 2,
$t|_{2.1} = a$ *is the term at position* 2.1 *and*
$t|_{2.2} = b$ *is the term at position* 2.2.

Consider Definition 3 of a *formula*, which introduces logical connectives, atoms and quantifiers into the signature of FOL.

Definition 3 (formula, semi-formula)

- $\top, \bot$ *are formulas*
- *If $t_1, \ldots, t_n$ are terms and P is an n-place predicate symbol, then $P(t_1, \ldots, t_n)$ is an (atomic) formula*
- *If A is a formula then $\neg A$ is a formula.*
- *If A, B are formulas then $(A \to B)$, $(A \wedge B)$, $(A \vee B)$ are formulas.*
- *If $A\{x \leftarrow \alpha\}$ for $x \in V_b$ and $\alpha \in V_f$ is a formula then $\forall x A$ and $\exists x A$ are formulas.*

Semi-formulas are defined in the same way, except for the definition of atomic formulas where semi-terms are admitted.

For the idea of replacing all occurrences of a particular variable by terms we describe *substitutions* in Definition 4.

Definition 4 (substitution)
A substitution σ is a mapping from $V_f \cup V_b$ to the set of terms *s.t. $\sigma(v) \neq v$ for only finitely many $v \in V_f \cup V_b$. Substitutions are written in postfix, i.e. we write $A\sigma$ instead of $\sigma(A)$, where A is an expression (i.e. either a formula or a term). We denote a substitution as $A[x\backslash t]$, for an expression A, a variable x and a term t, s.t. x is substituted by t in A. If σ is a substitution with $\sigma(x_i) = t_i$ for $x_i \neq t_i (1 \leq i \leq n)$ and $\sigma(v) = v$ for $v \neq \{x_1, \ldots, x_n\}$ then we denote σ by $[x_1\backslash t_1], \ldots, [x_n\backslash t_n]$. We will use the notation $A[x\backslash T]$ to describe a set obtained by substituting x in A with all terms $t \in T$, i.e. $\{A[x\backslash t] | t \in T\}$.*

Since we will need the following concept of a *subformula*, we describe it in Definition 5.

Definition 5 (subformula)
Let F be a formula. Then the function $subformula(F)$ generates the set of all subformulas of F as follows.

- *if $F = A$, whereas A is an atom, then $subformula(F) = \{A\}$*
- *if $F = \neg A$, then $subformula(F) = \{\neg A\} \cup subformula(A)$*
- *if $F = A \otimes B$, where $\otimes \in \{\wedge, \vee, \rightarrow\}$, then*
  $subformula(F) = \{A \otimes B\} \cup subformula(A) \cup subformula(B)$
- *if $F = \forall x A$, then*
  $subformula(F) = \{\forall x A\} \cup \bigcup_{t \in T} subformula(A[x \backslash t])$,
  where T is the set of terms
- *if $F = \exists x A$, then*
  $subformula(F) = \{\exists x A\} \cup \bigcup_{t \in T} subformula(A[x \backslash t])$,
  where T is the set of terms

We denote the set of all variables occurring in a formula F by $V(F)$.

2.2 Grammars and languages

In order to define the concept of a *term language*, we have to clarify the notion of a signature in Definition 6, which provides a way to restrict the appearance of a *term language*.

Definition 6 (Signature Σ)
Let F be a set of function symbols, V be the set of variable symbols ($F \cap V = \{\}$) and $A = \{(\cdot)\cdot, \}$ be the set of auxiliary symbols, then

$$\Sigma = F \cup V \cup A$$

*represents the corresponding **signature**.*
We say that a term t satisfies a signature Σ, if it is build of function symbols $f \in F$, variables $v \in V$ and auxiliary symbols A.
We write $\Sigma \vdash t$ if a term t satisfies a signature Σ.

Consider Definition 7 of a *term language*, which comprise a set of terms satisfied by a given signature.

Definition 7 (Language)
Let Σ be a signature and T be a potentially infinite set of terms satsifying Σ, then the term set

$$L = \{t \mid t \in T, \Sigma \vdash t\}$$

*is called a **language**.*
For a language comprising a finite set of terms $\{t_1, \ldots, t_n\}$, i.e. a finite language *we write*

$$L = \{t_1, \ldots, t_n\}$$

We may ommit the definition of a signature if it can be inferred out of the context.

A language can conceivably be represented by a *term grammar*. Before we can define a term grammar we have to consider first particular parts of its components, i.e. the definition of *non-terminals*, *terminal terms* and *non-terminal terms* in Definition 8, and the *production* or *rule* in Definition 9.

Definition 8 (Non-/Terminal terms)
*Let $\Sigma = F \cup V \cup A$ be a signature and N be a set of symbols, where forall $n \in N$ holds $n \notin F$, $n \notin V$ and $n \notin A$. Then N denotes the set of **non-terminals**.*
*Let t be a term s.t. $\Sigma \vdash t$, i.e. without comprising an occurrence of a non-terminal $n \in N$, then t is called a **terminal term**.*
*Let s be a term s.t. $\Sigma \nvdash s$ but $\Sigma \cup N \vdash s$, i.e. s comprises at least one non-terminal $n \in N$, then s is called a **non-terminal term**.*

Definition 9 (Production/Rule)
Let s be a non-terminal term *and t be either a* non-terminal term *or a* terminal term*, then a rule is an expression of the form*

$$s \to t$$

and represents a transition where a term s within an arbitrary term u is replaced by a term t .

Consider Definition 10 which clarifies the application of a rule.

Definition 10 (Application of a production rule)
Let $s \to t$ be a rule and u be a non-terminal term *comprising at least one occurrence of s.*

$$u \overset{s \to t}{\Rightarrow} u'$$

denotes the application of said rule to a non-terminal term u, which leads to a term u', s.t. one occurrence of s in u is replaced by t. Note: s can occur several times as a subterm in a term u, e.g.

$$u = f(g(b), g(g(s)), c, h(s))$$

Then an application of above production would either lead to the non-terminal term u'

$$u' = f(g(b), g(g(t)), c, h(s))$$

or to the non-terminal term u''

$$u'' = f(g(b), g(g(s)), c, h(t))$$

We denote a particular application of a rule onto a term a *derivation step* and define it in Definition 11.

Definition 11 (Derivation step $\succ$)
Let R be a set of rules and t an arbitrary term, then

$$t \succ_r t', \text{ where } r \in R$$

denotes the application of the rule $r \in R$ on t resulting in a new term t', i.e.

$$t \overset{r}{\Rightarrow} t'$$

A subsequent application of rules is called a *derivation*, which consists of a sequent of derivation steps and is described in Definition 12.

Definition 12 (Derivation $\succ^*$)
Let R be a set of rules, then the sequence $t, \ldots, t'$ is called a derivation and can be written as

$$t \succ_{r_1} \cdots \succ_{r_m} t', \text{ for } r_i \in R$$

It represents a progression of multiple applications of rules $r_i \in R$ on a term t and produces a term t'. Instead of enumerating all derivation steps of the derivation, we can simply write

$$t \succ^* t'$$

Finally we can describe a term grammar in Definition 13, which is able to represent a language L in a possibly more condensed way.

Definition 13 (Term grammar G)
A term grammar G is defined by a 4-tuple (S, N, τ, P), where

- *S denotes the set of* terminal *symbols, i.e. variable and function symbols, which typically is a set $S \subseteq (V_f \cup F^l)$ where $l \geq 0$. We explicitly denote the arity of a function symbol by f/k if $f \in F^k$ in case it cannot be inferred from the context*
- *N is the set of* non-terminal *symbols as defined in 8 s.t. for every $n \in N$ holds $n \notin (V_f \cup V_b \cup F^l)$.*
- *P be a finite set of rules*
- *τ represents the* axiom *or* starting symbol*, which is a non-terminal ($\tau \in N$) and occurs on the left-hand side of the first rule applied in every possible derivation of G*

A term grammar G produces a term t, denoted by $\succ^*$, iff there exists a derivation producing this term, i.e.

$$G \succ^* t$$

iff

$$\exists r_1, \ldots, \exists r_m \text{ s.t. } \tau \succ_{r_1} \cdots \succ_{r_m} t \text{ where } r_1, \ldots, r_m \in P$$

The language L produced by G consists of all terms possibly derivable in G, i.e.

$$\mathcal{L}(G) = \{t \mid G \succ^* t\}$$

Since this thesis focuses on term grammars only, we will from now on call it simply a *grammar*, although there are many types of grammars. Before we proceed by defining a particular grammar, needed for this thesis, consider following possible properties of a grammar.

context-free

A *context-free* grammar defines restrictions regarding the left-hand sides of the production rules and thus implies somehow a monotone term growth when considering grammars with out the empty word ϵ as defined in Definition 14.

Definition 14 (context-free grammar)
*Let t be a term and $G = (S, N, \tau, P)$ be a **context-free** grammar, iff for all production rules*

$$\alpha \to t$$

holds that the left-hand side consists only of a non-terminal $\alpha \in N$.

Such a grammar is also called a *Type 2* grammar according to the *Chomsky Hierarchy*. Since it is context-free and we consider in this thesis only grammars without ϵ-rules, which would allow us to remove a non-terminal within a term, this property ensures that an already introduced terminal cannot be removed from a generated term, which leads to a monotone term-growth.

acyclic

Definition 15 describes the acyclicity property of a grammar by an existing ordering in the productions of a grammar.

Definition 15 (acyclicity)
Let $G = (S, N, \tau, P)$ be a context-free grammar, then G is called acyclic if there exists a strict total ordering $\ll$ on N s.t. for all rules $\alpha \to A$ in P holds

$$\textit{If } \beta \in V(A) \textit{ then } \alpha \ll \beta$$

totally rigid

Consider Definition 16 for a *totally rigid* grammar, which somehow represents the behavior of term substitutions of all occurrences of a certain variable in a term.

Definition 16 (total rigidity)
*Let $G = (S, N, \tau, P)$ be a grammar, then G is called **totally rigid** if the application of a rule $A \to B$ enforces all occurrences of A to be substituted with B. In other words let t be a term and $t|p_1 = t|p_2 = \cdots = t|p_n$ be subterms of t equal to A. Then all subterms at positions $p_1, \ldots, p_n$, representing the left-hand side of the rule to be applied, are replaced by the right-hand side B, s.t. $t|p_1 = t|p_2 = \cdots = t|p_n = B$.*

After describing acyclicity, context-freeness and total rigidity w.r.t. to a grammar we proceed by Definition 17 of a $trat$ grammar comprising all those properties.

Definition 17 ($trat$-grammar)
Let G be a $trat$-grammar, iff it is **context-free**, **acyclic** *and* **totally rigid**.
*Note: $trat$-grammar is an abbreviation for **t**otally **r**igid **a**cyclic **t**ree-grammar.*

These properties may seem artificial yet, but will become clearer when considered in a proof-theoretic context later on. Lastly we define a *trat-n* grammar in Definition 18, which merely adds a restriction regarding the non-terminals to the definition of a $trat$ grammar.

Definition 18 (trat-n grammar)
*A **trat-**n grammar is a* trat *grammar containing exactly $n+1$ non-terminals. W.l.o.g. we will denote the non-terminals in a $trat$-n grammar as $N = \{\alpha_0, \alpha_1, \ldots, \alpha_n\}$ s.t. $\alpha_i \ll \alpha_j$ for $i < j$, where α_0 is the start symbol or axiom of the grammar.*

A grammar defined in Definition 18 can be rewritten in a more condensed way, i.e. as a decomposition which is defined through the $\circ_\alpha$ operator (Definition 19).

Definition 19 ($\circ_\alpha$-operator)
Let S_1, S_2 be sets of terms, then

$$S_1 \circ_\alpha S_2$$

represents the set

$$\{t[\alpha \backslash s] \mid t \in S_1, s \in S_2\}$$

whereas all occurrences of α in t get substituted by s at once.

Let $G = (S, N, \alpha_0, P)$ be a *trat-n*-grammar, which can be represented (due to its linear shape given by its properties) as a decomposition. Furthermore let

$$\begin{aligned} P = \{ \quad & \alpha_0 \Rightarrow \quad t_{0,1} | \ldots | t_{0,m_0} \\ & \alpha_1 \Rightarrow \quad t_{1,1} | \ldots | t_{1,m_1} \\ & \quad \vdots \\ & \alpha_n \Rightarrow \quad t_{n,1} | \ldots | t_{n,m_n} \\ \} & \end{aligned}$$

be the set of rules, then we can rewrite G as a decomposition by interpreting the right-hand sides as a set of terms. Each set will then be concatenated with its successor set by the $\circ_{\alpha_i}$ operator, which stands for the cartesian product of substituting all α_i occurrences, in a totally rigid manner, in every term of the left set by every term of the right set. This transformation results in following grammar decomposition

$$\{t_{0,1}, \ldots, t_{0,m_0}\} \circ_{\alpha_1} \{t_{1,1}, \ldots, t_{1,m_1}\} \circ_{\alpha_2} \cdots \circ_{\alpha_n} \{t_{n,1}, \ldots, t_{n,m_n}\}$$

Note: Since α_0 is our starting symbol it will not explicitly appear in this representation.

Example 2 (Example of a decomposition)
Let $L = \{f(0), f(s(0)), f(s^2(0)), \ldots, f(s^7(0))\}$ be the given language and
$G = (\{0/0, f/1, s/1\}, \{\alpha_0, \alpha_1, \alpha_2\}, \alpha_0, P)$ a $trat$-3 grammar generating it, where

$$\begin{aligned} P = \{ \quad & \alpha_0 \Rightarrow \quad f(\alpha_1) | f(s(\alpha_1)) \\ & \alpha_1 \Rightarrow \quad \alpha_2 | s^2(\alpha_2) \\ & \alpha_2 \Rightarrow \quad 0 | s^4(0) \\ \} & \end{aligned}$$

is its corresponding set of rules, then

$$\{f(\alpha_1), f(s(\alpha_1))\} \circ_{\alpha_1} \{\alpha_2, s^2(\alpha_2)\} \circ_{\alpha_2} \{0, s^4(0)\}$$

is its decomposition representation. Note that in case we want to generate $\mathcal{L}(G)$ we iteratively proceed by applying $\circ_{\alpha_i}$ ($1 \leq i \leq 2$) of the decomposition to retrieve first

$$\{f(\alpha_2), f(s(\alpha_2)), f(s^2(\alpha_2)), f(s^3(\alpha_2))\} \circ_{\alpha_2} \{0, s^4(0)\}$$

and thus

$$\mathcal{L}(G) = \{f(0), f(s(0)), f(s^2(0)), f(s^3(0)), f(s^4(0)), f(s^5(0)), f(s^6(0)), f(s^7(0))\}$$

Note that grammar decompositions can be categorized w.r.t. to their starting term set, as *simple* or even *trivial* grammars and are defined in Definitions 20 and 21, respectively.

Definition 20 (Simple grammar)
*A grammar decomposition $G = U \circ_{\alpha_1} S_1 \circ_{\alpha_2} \cdots \circ_{\alpha_n} S_n$ is called **simple** iff $|U| = 1$, i.e. U consists of one term only.*

Definition 21 (Trivial grammar)
*A grammar decomposition $G = U \circ_{\alpha_1} S_1 \circ_{\alpha_2} \cdots \circ_{\alpha_n} S_n$ is called **trivial** iff $U = \{\alpha_1\}$.*

2.3 Sequent calculus LK

We proceed by defining Gentzen's sequent calculus ***LK*** [4]. A proof in **LK** is a tree, where the root is represented as a sequent, as defined in Definition 22, which is to be proven and is called the *end-sequent* of the proof.

Definition 22 (Definition of a sequent)
Let Γ, Δ are finite multi-sets of formulas then

$$S : \Gamma \vdash \Delta$$

*is a **sequent**, where Γ is called the **antecedent** and Δ the **succedent** of S.*

Consider 3 of a sequent in **LK**.

Example 3 (Example of a sequent)
Let N be an atom, s a unary function symbol and 0 a constant, then

$$\forall x(N(x) \supset N(s(x))), N(0) \vdash N(s(s(0)))$$

is an example of a sequent.

Semantically a sequent

$$S : A_1, \ldots, A_n \vdash B_1, \ldots, B_m$$

as described in Definition 22, can be interpreted as

$$(A_1 \wedge \cdots \wedge A_n) \rightarrow (B_1 \vee \cdots \vee B_m)$$

Located in an **LK** proof's leafs are axioms which are further processed and merged by rules, s.t. all branches end up in the root. Before we proceed by defining the rules of **LK** let us focus on **LK**'s axioms which make up the leafs of every proof in sequent calculus as described in Definition 23.

Definition 23 (Axiom)
An axiom *is a sequent of the form*

$$A \vdash A$$

where A is an atom.

All rules in **LK** consist of a conclusion (bottom half) and one to two premises (upper half) depending on the arity of the rule. Furthermore they can be divided into two groups: the *logical rules* and the *structural rules*. Both types can again be separated into two groups. Those which apply on the antecedent(s) and those who apply on the succedent(s) of the affected sequent(s). For the logical rules of **LK** see Figure 2.1 and Figure 2.2 and for the structural rules consider Figure 2.3 and Figure 2.4.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \neg\colon l \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge\colon l$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \vee B \vdash \Delta, \Delta'}\ \vee\colon l \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \supset B \vdash \Delta, \Delta'}\ \supset\colon l$$

$$\frac{A[x\backslash t], \Gamma \vdash \Delta}{\Gamma, \forall x A \vdash \Delta}\ \forall\colon l \qquad \frac{\Gamma, A[x\backslash \alpha] \vdash \Delta}{\Gamma, \exists x A \vdash \Delta}\ \exists\colon l$$

Figure 2.1: LK logical rules left

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \neg\colon r \qquad \frac{\Gamma \vdash \Delta, A \qquad \Gamma' \vdash \Delta', B}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \wedge B}\ \wedge\colon r$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\ \vee\colon r \qquad \frac{A, \Gamma \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\ \supset\colon r$$

$$\frac{\Gamma \vdash A[x\backslash \alpha], \Delta}{\Gamma \vdash \forall x A, \Delta}\ \forall\colon r \qquad \frac{\Gamma \vdash A[x\backslash t], \Delta}{\Gamma \vdash \exists x A, \Delta}\ \exists\colon r$$

Figure 2.2: LK logical rules right

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\, c\colon l \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\, w\colon l$$

Figure 2.3: LK structural rules left

$$\frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\, c\colon r \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash B, \Delta}\, w\colon r$$

Figure 2.4: LK structural rules right

Please note that in this definition of **LK** we consider a sequent to be a tuple of multi sets, therefore we do not need an *exchange* rule, in contrast to the original definition of sequent calculus in [4]. There are two logical rules, $\forall\colon r$ and $\exists\colon l$, which are subjected to restrictions, namely the *Eigenvariable condition*. This restriction prohibits the substituting variable α to occur in the sequent of the conclusion of the rule. The quantifiers introduced in those two rules, are called *strong quantifiers*. Analogously the quantifiers introduced via $\forall\colon l$ and $\exists\colon r$ are called *weak quantifiers* and do not require the Eigenvariable condition. Instead those weak quantifiers underlie another, weaker restriction, namely that the variable can only be instantiated by a semi-term t, i.e. t must not contain a variable bound in the quantified formula A. Let us take a look on Example 4 and observe, that $f(z)$ does not contain variable y and thus does not violate the mentioned restriction.

Example 4 (Example application of $\forall\colon l$)
Let P be an atom, f be a unary function symbol and z a free variable, then

$$\frac{\dfrac{\dfrac{\vdots\,\pi}{\exists y P(f(z), y) \vdash \exists y P(f(z), y)}}{\forall x \exists y P(x, y) \vdash \exists y P(f(z), y)}\, \forall\colon l$$

represents an example for an application of the $\forall\colon l$ rule. Note that we may abbreviate multiple subsequent applications of contraction and quantifier-rules by $\forall : l^*, \forall : r^*, \exists : l^*, \exists : r^*$ if needed.

For a clearer understanding of proofs in sequent calculus consider Example 5 illustrating a proof closing with two axioms.

Example 5 (Example of a proof)
Let A, B be atoms, then

$$\dfrac{\dfrac{\dfrac{A \vdash A}{A \vdash B, A}\, w{:}r \qquad \dfrac{B \vdash B}{B \vdash B, A}\, w{:}r}{A \vee B \vdash B, A}\, \vee\colon l}{A \vee B \vdash B \vee A}\, \vee\colon r$$

is a proof closing with two axioms $A \vdash A$ and $B \vdash B$.

The last rule, namely the **cut** rule is not required for completeness of **LK**. Gentzen showed that each proof with cuts can be transformed into a proof without cuts [4], which is known as the famous *Gentzen Hauptsatz*. The opposite, namely introducing cuts into a cut-free proof is subject of this thesis.

$$\frac{\Gamma \vdash \Delta, A \qquad A, \Gamma' \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \text{ cut}$$

Figure 2.5: cut rule

In Figure 2.5 one can see an abstract application of a cut, which is a binary rule, i.e. containing two sequents in the premise and one sequent in the conclusion. By applying the cut rule, the *cut-formula* A is removed from the succedent of the first sequent and the antecedent of the second sequent in the premise. The sequent in the conclusion is then constructed through the union of the antecedents and succedents of the premises, respectively. Cuts performed on quantified cut-formulas are called *quantified cuts*. Consider Theorem 1 of the *subformula property*, which holds for every rule in **LK** except for the cut-rule.

Theorem 1 (Sub formula property)
All formulas occurring in a cut-free derivation are sub-formulas of the end sequent.

For a proof of Theorem 1 consider [4]

In simple words this property claims that no formula within a cut-free derivation in **LK** gets lost. The cut-rule violates this property, since the cut-formula A is eliminated when performing the cut. Thus applying the cut-rule backwards requires to know a suitable cut-formula A, which is a hard task to find out. In order to guess a cut-formula A for a particular cut, we would have to decide whether A is a cut-formula. Since first-order logic is undecidable, we cannot decide for a formula A if it is a cut-formula of a particular cut.

2.4 The MinCostSAT and partial weighted MaxSAT Problem

In the course of this thesis we will need to make use of two particular problems, namely the *MinCostSAT Problem* and the *partial weighted MaxSAT Problem*. Before we can deal with those problems we start by describing their common basis, namely in the *SAT* Problem in Definition 24.

Definition 24 (SAT)
The SAT *Problem consists of deciding whteter a given Boolean formula F is satisfiable, which we assume to be given in* conjunctive normal form*. A solution for an instance of the* SAT *Problem is an interpretation I s.t. $I \vDash F$.*

Given a satisfiable formula, in general there need not to be a unique solution, i.e. a valid interpretation (or model) for an instance of the *SAT* Problem. Unfortunately its definition does not provide a proper way to decide whether a model is better than another one, e.g. via an

objective function. Since we will need to rate any valid interpretation w.r.t. a Boolean formula F, we overcome this issue by extending the *SAT* Problem by assigning costs to all appearing variables in a formula F and thus obtain Definition 25 of the *MinCostSAT Problem*.

Definition 25 (MinCostSAT problem)
An instance of the MinCostSAT Problem consists of a Boolean formula F *and costs for all variables $x_i \in V(F)$. The variables are associated with individual costs $c_i \in \mathbb{N}$, where $1 \leq i \leq n$, whereas c_i represents the costs of variable x_i.*
A solution of such an instance is an interpretation I satisfying F s.t. $\sum_{I(x_i)=true} c_i$ is minimal, i.e.

$$\operatorname*{argmin}_I \{ \sum_{\substack{x_i \in V(F) \text{ s.t.} \\ I(x_i)=true}} c_i \mid I \vDash F \}$$

Note that the corresponding decision problem, namely if there exists an interpretation satisfying F of costs

$$c = \sum_{\substack{x_i \in V(F) \text{ s.t.} \\ I(x_i)=true}} c_i$$

within a particular cost bound $n \in \mathbb{N}$, i.e. $c \leq n$, is **NP**-complete [3] [14].

To introduce a distinctive feature for models of an instance of the SAT Problem, it can be extended in another way. Besides the Boolean formula F, another set of propositional formulas g_i ($1 \leq i \leq n$) with individual costs c_i assigned can be given. The aim of the problem is to maximize the sum of all costs c_i, where their corresponding g_i are fulfilled by the model, i.e. $I \vDash g_i$. This extension is called the *partial weighted MaxSAT Problem* and is described in Definition 26.

Definition 26 (partial weighted MaxSAT Problem)
Let F be a propositional logic formula and $\{g_1, \ldots, g_n\}$ be a set of Boolean formulas, where to each formula g_i individual costs c_i are assigned for $1 \leq i \leq n$, i.e. $G = \{(g_1, c_1), \ldots, (g_n, c_n)\}$. Then an instance of the partial weighted MaxSAT Problem consists of F and G. A solution of the problem is an interpretation I, which entails F while maximizing the sum of all c_i, where for the corresponding g_i holds $I \vDash g_i$, i.e.

$$\operatorname*{argmax}_I \{ \sum_{\substack{(g_i,c_i) \in G \\ I \vDash g_i}} c_i \mid I \vDash F \}$$

We will see later that the MinCostSAT Problem can be reduced to the *partial weighted MaxSAT Problem*, for which a variety of solvers exist.

CHAPTER 3

State of the art

In order to understand which steps are necessary to introduce cuts into a cut-free proof in **LK**, we will describe the procedure in an intuitive manner. Starting from a cut-free proof, the first idea was to consider the proof as a whole, which did not work out very well. The variations in structure and size of a proof induce a level of complexity, which complicates efficient processing in general. Therefore a better way for describing a proof and its particular peculiarities was needed. Before we go deeper into the procedure of extracting appropriate information from a proof, let us consider the example proof π with cuts in Figure 3.1 the proof π^* without cuts in Figure 3.2, respectively.

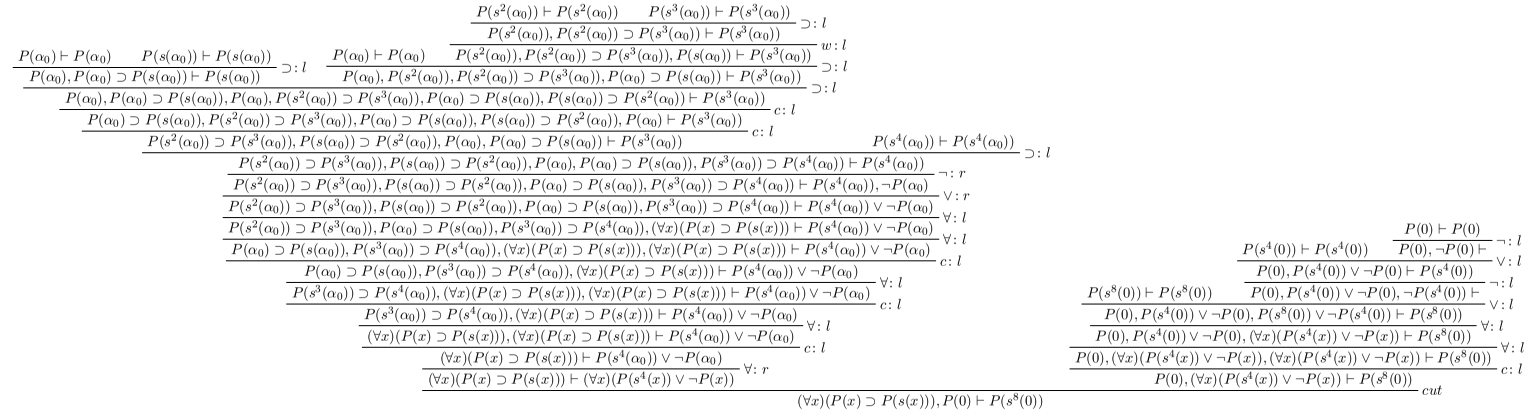

Figure 3.1: Proof π with cuts

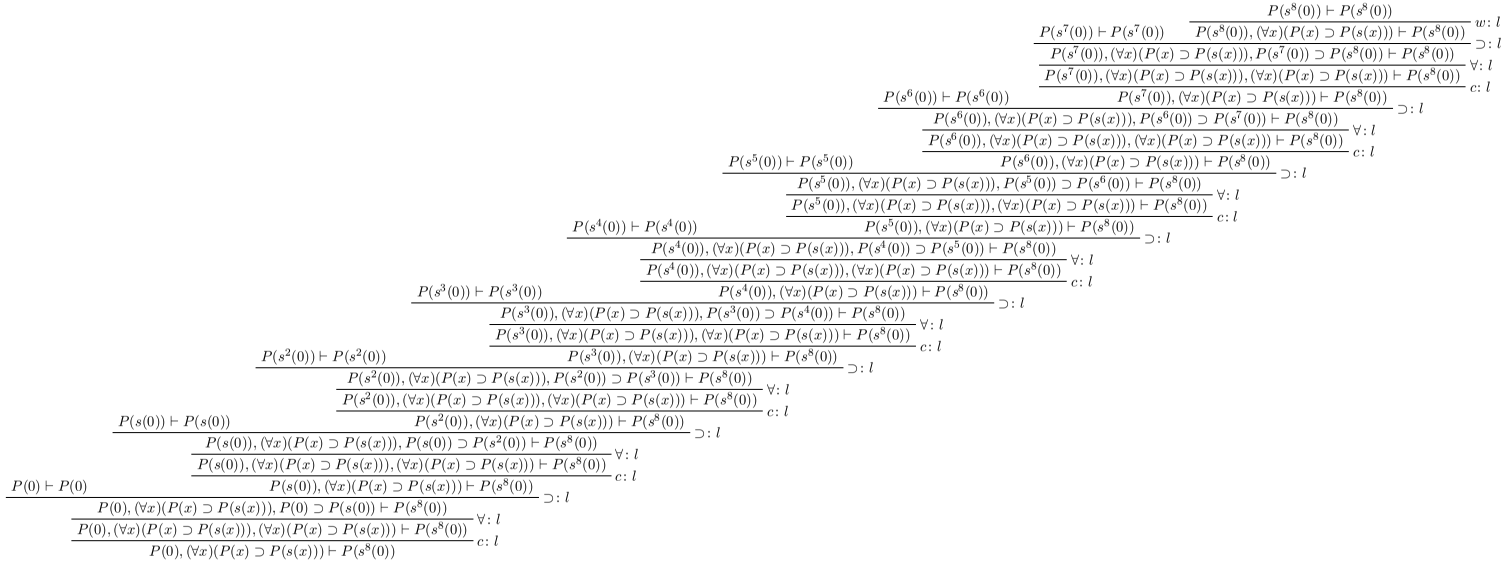

Figure 3.2: Proof π^* without cuts

Both π an π^* prove the same theorem, namely $P(0), (\forall x)(P(x) \supset P(s(x))) \vdash P(s^8(0))$, but the proofs differ in size and shape. Observe that in order to prove the mentioned theorem we have to instantiate the quantified formula $(\forall x)(P(x) \supset P(s(x)))$ eight times to derive $P(s^8(0))$ in the cut-free proof π^*. On the other side, consider π which has a single cut and differs in the amount of instantiations of the quantified formula. It merely skips the instantiations of $P(s^4(0)), P(s^5(0)), P(s^6(0))$ and $P(s(^7(0))$ and has only six applications of the $\forall : l$ rule. Instead it contains two additional rule applications, where one represents the instantiation $P(s^8(0))$ generated from $P(s^4(0))$ and the other rule application performs the cut. Consider the cut-formula of π, namely

$$(\forall x)(P(s^4(x)) \vee \neg P(x))$$

This formula allows us to take a shortcut while deriving the succedent by resting on an intermediate step, namely a lemma. Since we already derived from our quantified formula $(\forall x)(P(x) \supset P(s(x)))$ the above cut-formula, we do not necessarily have to derive again the last four steps to complete our proof, but can avoid this by performing a cut. Although the previous example proofs do not unveil the method's full potential of compression, it is capable of reducing a proof's size and improving its readability by bringing out lemmas.

We know by now that there is a difference between the instantiations of such quantified formulas within a cut-free and a proof with cuts. For defining those instantiations needed for building a proof in general, we present the definition of a *Herbrand sequent*.

3.1 Herbrand sequent extraction

From every proof in **LK** we can extract a *Herbrand sequent*, which is quantifier-free and instead of the end-sequent contains exactly those instantiations of quantified formulas necessary to build the proof. To extract a Herbrand sequent from a proof consider the method described in [1]. The method rests upon the extension of **LK** by so called *array-rules*. Quantifier rules (i.e. $\forall l, \forall r, \exists r, \exists l$) will be removed from a given proof and replaced by *array-formulas* containing their instances. After this transformation, the end-sequent is transformed into an ordinary sequent containing no array-formulas, which is already the *Herbrand sequent* of the proof, i.e. $H(\pi)$.

Before we are able to define the term of a Herbrand sequent, we have to raise the subject of a proper hierarchy of formulas or sequents. Consider Definition 27 which describes a classification of first-order formulas into partitions regarding their quantifier complexity.

Definition 27 ($\boldsymbol{\Sigma_n}$, $\boldsymbol{\Pi_m}$ formula)
Let ϕ be a first-order formula, which is logically equivalent to a quantifier-free first-order formula.
Then ϕ is classified as Σ_0 and Π_0.
For every $n > 0$ we define Σ_n and Π_n:
If ϕ is logically equivalent to a formula of the form $\exists x_1 \exists x_2 \ldots \exists x_k \psi$, where ψ is Π_n, then ϕ is assigned the classification Σ_{n+1}.
If ϕ is logically equivalent to a formula of the form $\forall x_1 \forall x_2 \ldots \forall x_k \psi$, where ψ is Σ_n, then ϕ is assigned the classification Π_{n+1}.

Note that there exist formulas which can be classified by two classes simultaneously, due to their logical equivalence w.r.t. possible quantifier permutations.

In Definition 28 we describe a hierarchy of quantifier complexity regarding sequents and merely describe the class of Σ_1-sequents, which consist of formulas of two particular complexity classes, namely Σ_1 and Π_1.

Definition 28 ($\mathbf{\Sigma_1}$-sequent)
Let $\phi_1, \ldots, \phi_m$ be Π_1-formulas and $\psi_1, \ldots, \psi_n$ Σ_1-formulas, then

$$\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$$

is called a Σ_1-sequent.
E.g. for formulas ϕ' in Π_0 and ψ' in Σ_0, we likewise obtain a Σ_1-sequent

$$\forall x_1 \ldots \forall x_k \phi' \vdash \exists y_1 \ldots \exists y_l \psi'$$

for $k, l \geq 0$.
Note:

- *Σ_1-sequents contain weak quantifiers only*
- *Each first-order sequent can be transformed into a Σ_1-sequent through skolemization and prenexing*

We proceed by defining the concept of a Herbrand sequent in Definition 29, which comprises in a way the structure of a proof, by enumerating all instantiations of quantified formulas necessary to derive the end-sequent.

Definition 29 (Herbrand sequent)
Let

$$\forall x_1^1, \ldots, \forall x_{l_1}^1 F_1, \ldots, \forall x_1^p, \ldots, \forall x_{l_p}^p F_p \vdash \exists x_1^{p+1}, \ldots, \exists x_{l_{p+1}}^{p+1} F_{p+1}, \ldots, \exists x_1^q, \ldots, \exists x_{l_q}^q F_q$$

with $l_i \geq 0$ and F_i quantifier free be a valid Σ_1-sequent. Furthermore we write $\overline{x}$ for a vector $(x_1, \ldots, x_n)$ of variables, $\overline{t}$ for a vector $(t_1, \ldots, t_n)$ of terms and $[\overline{x} \backslash \overline{t}]$ for the substitution $[x_1 \backslash t_1] \ldots [x_n \backslash t_n]$. A valid sequent of the form

$$\{F_i[\overline{x}_i \backslash \overline{t}_{i,j}] | 1 \leq i \leq p, 1 \leq j \leq n_i\} \vdash \{F_i[\overline{x}_i \backslash \overline{t}_{i,j}] | p < i \leq q, 1 \leq j \leq n_i\}$$

is then called a Herbrand sequent*, where $\overline{t}_{i,j} = (t_{i,1}, \ldots, t_{i,n_i})$ are the vectors of instances of F_i. Note that by Herbrand's Theorem [7] every valid sequent containing only weak quantifiers possesses a Herbrand sequent.*

To extract a Herbrand sequent from a given proof π, we have to consider all instantiations of quantified formulas, by simply reading off the instantiated terms from all quantifier-rules applied in π. Lastly collecting all formulas obtained by substituting corresponding bound variables in respective quantified formulas results in a propositional valid sequent and thus in the Herbrand

sequent of π. A Herbrand sequent extracted from a proof π will from now on be denoted by $H(\pi)$. When generating the *Herbrand sequent* of a proof with cuts, we retrieve (in contrast to the Herbrand sequent of a cut-free proof) a specifically looking sequent, namely the *extended Herbrand sequent*. In [9] the definition of an extended Herbrand sequent is described and can be applied to Σ_1-sequents. To make the following definitions and construction methods more transparent, we proceed as in [9] and merely discuss sequents of the form $\forall x F \rightarrow$, although it is possible to generalize the mentioned methods to work with Σ_1-sequents. Consider Definition 30 of an extended Herbrand sequent of the form $\forall x F \rightarrow$.

Definition 30 (extended Herbrand sequent)
Let $u_1, \ldots, u_m$ be terms, let $A_1, \ldots, A_n$ be quantifier-free formulas, let $\alpha_1, \ldots, \alpha_n$ be variables, and let $s_{i,j}$ for $1 \leq i \leq n, 1 \leq j \leq k_i$ be terms s.t.

1. *$V(A_i) \subseteq \{\alpha_i, \ldots, \alpha_n\}$ for all i, and*
2. *$V(s_{i,j}) \subseteq \{\alpha_{i+1}, \ldots, \alpha_n\}$ for all i, j.*

then

$$\begin{aligned} H = \quad & F[x\backslash u_1], \ldots, F[x\backslash u_m], \\ & A_1 \supset (\bigwedge_{j=1}^{k_1} A_1[\alpha_1\backslash s_{1,j}]), \\ & \vdots \\ & A_n \supset (\bigwedge_{j=1}^{k_n} A_n[\alpha_n\backslash s_{n,j}]) \rightarrow \end{aligned}$$

is called an extended Herbrand sequent of $\forall x F \rightarrow$ if H is a tautology. We call formulas within an extended Herbrand sequent of the form

$$A_i \supset (\bigwedge_{j=1}^{k_i} A_i[\alpha_i\backslash s_{i,j}])$$

cut-implications
Note:

- *Given a proof with cuts, we can extract the extended Herbrand sequent*
- *From a given extended Herbrand sequent, we can construct the underlying proof with cuts*

Observe the instantiated terms are represented by the sets

$$U = \{u_1, \ldots, u_m\}$$

and

$$S_1 = \{s_{1,1}, \ldots, s_{1,k_1}\}, \ldots, S_n = \{s_{n,1}, \ldots, s_{n,k_n}\}$$

An extended Herbrand sequent encodes the cuts in a proof, since all instantiations of the universal formulas are represented for every cut. As already mentioned this essential information comprised by an extended Herbrand sequent is sufficient to build a proof with cuts. Alternatively from a given proof with cuts, we can extract an extended Herbrand sequent which contains all necessary information for building again a proof with cuts. Assume we successfully extracted a Herbrand sequent from our proof π containing one cut in Figure 3.1 and consider it in Example 6.

Example 6 (Example of an extended Herbrand sequent)
Let π be the proof in Figure 3.1, then the extended Herbrand sequent of π is constructed by considering the ground formulas of the end-sequent and all $\forall : \ l$ applications and their respective quantifier instantiations. We abbreviate the formula $P(s^4(x)) \vee \neg P(x)$ by $F(x)$, where $\forall x F(x)$ is our used cutformula.
Then $H(\pi) =$

$$P(0), (P(\alpha_0) \supset P(s(\alpha_0))), (P(s(\alpha_0)) \supset P(s^2(\alpha_0))),$$
$$(P(s^2(\alpha_0)) \supset P(s^3(\alpha_0))), (P(s^3(\alpha_0)) \supset P(s^4(\alpha_0))),$$
$$F(\alpha_0) \supset (F(0) \wedge F(s^4(0)))$$
$$\vdash P(s^8(0))$$

is its extended Herbrand sequent.

Example 6 above shows an *extended Herbrand sequent*, which differs from an ordinary Herbrand sequent by its instantiations of the quantified formula. A sequent of the above form represents a proof with n Π_1-cuts, whose cut-formulas are $\forall \alpha_1 A_1, \ldots, \forall \alpha_n A_n$. The α_i are the eigenvariables of the universal quantifiers in these cut-formulas, the $s_{i,j}$ the terms of the instances of the cut-formulas on the right-hand side of the cut and the u_i the terms of the instances of our end-formula $\forall x F$.

3.2 Building a proof from an extended Herbrand sequent

In order to build a proof with cuts, we have to generate an extended Herbrand Sequent. This sequent can be build from an underlying Herbrand Sequent (previously extracted from a cut-free proof) through reducing the problem to that of finding a minimal grammar for a specific language encoding the sequent. Since this represents the main component of this thesis, we will focus on it later on (in Section 3.3 et sequentes) and assume for now that we successfully extracted a Herbrand sequent from a cut-free proof π^* and transformed it into an extended Herbrand sequent. As an extended Herbrand sequent is merely a representation of a proof with cuts, we may build a proof from it and its corresponding cut-free proof π^*, which is possible due to the description in [9] and is achieved in an iterative manner. Let us abbreviate a cut-implication as previously shown in Definition 30, i.e.

$$A_i \supset \bigwedge_{j=1}^{k_i} A_i[\alpha_i \backslash s_{i,j}]$$

simply as CI_i and let $U = \{u_1, \ldots, u_m\}$. As previously mentioned in Definition 30 consider following conditions fulfilled by the sets $S_1, \ldots, S_n$ and formulas $A_1, \ldots, A_n$:

1. $V(A_i) \subseteq \{\alpha_i, \ldots, \alpha_n\}$ for all i, and
2. $V(S_i) \subseteq \{\alpha_{i+1}, \ldots, \alpha_n\}$ for all i.

Again we will describe the procedure by means of a sequent of the form $\forall x F \to$, because we already know that we can handle sequents of higher complexity analogously. The sequent in Definition 30 is then rewritten in a condensed way as

$$F[x\backslash U], \mathrm{CI}_1, \ldots, \mathrm{CI}_n \to$$

We will see that the sequent

$$H \;=\; F[x\backslash U], A_1 \supset \bigwedge_{j=1}^{k_1} A_1[\alpha_1\backslash s_{1,j}], \ldots, A_n \supset \bigwedge_{j=1}^{k_n} A_n[\alpha_n\backslash s_{n,j}] \to$$

has a proof of the following form:

$$\dfrac{\dfrac{\dfrac{\vdots}{F[x\backslash U] \to A_1, \ldots, A_n} \quad \dfrac{\vdots}{\bigwedge_{j=1}^{k_1} A_1[\alpha_1\backslash s_{1,j}], F[x\backslash U_1] \to A_2, \ldots, A_n}}{F[x\backslash U], \mathrm{CI}_1 \to A_2, \ldots, A_n}\supset_l \quad \dfrac{\vdots}{F[x\backslash U], \mathrm{CI}_1, \ldots, \mathrm{CI}_{n-1} \to A_n} \quad \dfrac{\vdots}{\bigwedge_{j=1}^{k_n} A_n[\alpha_n\backslash s_{n,j}], F[x\backslash U_n] \to}}{F[x\backslash U], \mathrm{CI}_1, \ldots, \mathrm{CI}_n \to}\supset_l$$

Note that $F[x\backslash U_i]$, where $U_i = \{u \in U \mid V(u) \subseteq \{\alpha_{i+1}, \ldots, \alpha_n\}\}$, denotes the left-hand side of the current cut-implication and is built from the quantified formula F and substitutions depending only on terms with non-terminals α_j of indexes $j > i$. Starting from this base we can simply introduce cuts and corresponding quantifiers by replacing a segment of the form

$$\dfrac{F[x\backslash U], \mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1} \to A_i, \ldots, A_n \quad \bigwedge_{j=1}^{k_i} A_i[\alpha_i\backslash s_{i,j}], F[x\backslash U_i] \to A_{i+1}, \ldots, A_n}{F[x\backslash U], \mathrm{CI}_1, \ldots, \mathrm{CI}_i \to A_{i+1}, \ldots, A_n}\supset_l$$

by

$$\dfrac{\dfrac{\dfrac{F[x\backslash U_{i-1}], \forall x\, F \to A_i, \ldots, A_n}{F[x\backslash U_i], \forall x\, F \to A_i, \ldots, A_n}\forall_{\mathrm{l}}^*}{F[x\backslash U_i], \forall x\, F \to \forall x\, A_i[\alpha_i\backslash x], A_{i+1}, \ldots, A_n}\forall_{\mathrm{r}} \quad \dfrac{A_i[\alpha_i\backslash S_i], F[x\backslash U_i] \to A_{i+1}, \ldots, A_n}{\forall x\, A_i[\alpha_i\backslash x], F[x\backslash U_i] \to A_{i+1}, \ldots, A_n}\forall_{\mathrm{l}}^*}{F[x\backslash U_i], \forall x\, F \to A_{i+1}, \ldots, A_n}\,\mathrm{cut}$$

Finally the proof is finished at its root

$$\dfrac{F[x\backslash U_n], \forall x\, F \to}{\forall x\, F \to}\forall_{\mathrm{l}}^*.$$

Of course, intermediate derivation steps are necessary to derive sequents s.t. we are able to form up segments of the above form, which can be replaced then by cut-rule applications and corresponding additional rules. Let us focus on the last part, namely the construction of the A_i.

$$L_i \quad \text{for} \quad \mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}, F[x\backslash U] \to A_i, \ldots, A_n, \text{ and}$$
$$R_i \quad \text{for} \quad \bigwedge_{j=1}^{k_i} A_i[\alpha_i\backslash s_{i,j}], F[x\backslash U_i] \to A_{i+1}, \ldots, A_n.$$

Observe that L_i and R_i denote the left and right hand side of the implications and depend only on those A_j with $j \geq i$, where $1 \leq i \leq n$. Furthermore we clearly see that with increasing i, the complexity of R_i decreases. Thus we derive L_{n+1}, which is the extended Herbrand sequent H and thus the base case of above sketch for building a proof.

3.3 Abstraction of an extended Herbrand sequent

We just described in Definition 30 an extended Herbrand sequent with particular instantiations via term sets $U, S_1, \ldots, S_n$ of a quantified formula F and the individual cut-formulas, respectively. Since we want to reduce the task of finding an extended Herbrand sequent to another problem, namely finding exactly instantiated term sets $U, S_1, \ldots, S_n$, we need a more general definition to describe merely the structure of it. This leads to Definition 31 of a *schematic extended Herbrand sequent*, which allows us to consider the structure of an extended Herbrand sequent separately from its contained instantiations. This is achieved by defining it according to a grammar decomposition $G(H)$, i.e. a $trat$-n grammar defining the Herbrand sequent of a proof.

Definition 31 (schematic extended Herbrand sequent)
Let $G(H)$ be a grammar decomposition generating th term set T of a Herbrand sequent of the sequent $\forall x F \to$. $G(H)$ can be represented as $U \circ_{\alpha_1} S_1 \circ_{\alpha_2} \cdots \circ_{\alpha_n} S_n$ s.t. the language $\mathcal{L}(G) = T$ generated from $G(H)$ contains all instantiated terms. Recall that

$$U = \{u_1, \ldots, u_m\}$$

and

$$S_i = \{s_{i,1}, \ldots, s_{i,k_i}\}$$

Let $u_1, \ldots, u_m$ be terms, $X_1, \ldots, X_n$ be monadic second-order variables, $\alpha_1, \ldots, \alpha_n$ be variables and $s_{i,j}$, where $1 \leq i \leq n$ and $1 \leq j \leq k_i$ be terms.
Again assume following restriction holds, which we already know from previous definitions

$$V(S_i) \subseteq \{\alpha_{i+1}, \ldots, \alpha_n\} \text{ for all } i$$

Then the sequent

$$\begin{aligned} H = \quad & F[x\backslash u_1], \ldots, F[x\backslash u_m], \\ & X_1(\alpha_1) \supset \bigwedge_{j=1}^{k_1} X_1(s_{1,j}), \\ & \vdots, \\ & X_n(\alpha_n) \supset \bigwedge_{j=1}^{k_n} X_n(s_{n,j}) \end{aligned}$$

is called a schematic extended Herbrand sequent *of* $\forall x F \rightarrow$

Note that

$$\bigwedge_{t \in \mathcal{L}(G)} F[x\backslash t] \rightarrow \tag{3.1}$$

represents a Herbrand sequent with all instantiations for a quantified formula F, which can be used to successfully build a proof without cuts. All those instantiations, or terms, $t \in \mathcal{L}(G)$ are contained in a particular language and can be abbreviated by an underlying grammar decomposition

$$G = (U \circ_{\alpha_1} (S_1 \circ_{\alpha_2} \cdots \circ_{\alpha_n} S_n))$$

Let us take a closer look on Definition 31 and observe the second-order variables $X_1, \ldots, X_n$. Those variables are just placeholders and will be replaced via a substitution σ, which introduces instead of those variables, Lambda-expressions describing the cut-formulas. I.e. the second-order variables represent the not yet known cut-formulas, which will be instantiated by terms contained in the grammar decomposition accordingly. As written in the definition we consider the amount of those second-order variables to be n, i.e. we assume to introduce n cuts. Note that the first part of the schematic extended Herbrand sequent, i.e. $F[x\backslash u_1], \ldots, F[x\backslash u_m]$, is nothing less than the necessary part which has to be considered for the first cut introduced into the proof. Furthermore observe that Definition 31 of a schematic extended Herbrand sequent is an abstraction of an extended Herbrand sequent and roughly describes the structure of the sequent and its size. This sequent also allows us to consider essential parts of an extended Herbrand sequent, besides its size, namely its instantiations of used quantified formulas $U, S_1, \ldots, S_n$.

Let us reconsider this previously mentioned substitution σ, which helps us to separate the instantiations from the schematic extended Herbrand sequent. Such a substitution σ is can be a *solution* of a schematic extended Herbrand sequent and is described in Definition 33.

Definition 32 (canonical substitution σ)
A canonical substitution of a schematic extended Herbrand sequent H of $\forall x F \rightarrow$ is the substitution

$$\sigma = [X_i\backslash \lambda \alpha_i . C_i]_{i=1}^{n}$$

where

$$C_1 = \bigwedge_{j=1}^{m} F[x\backslash u_j]$$

is the first cut-formula and

$$C_{i+1} = \bigwedge_{j=1}^{k_i} C_i[\alpha_i \backslash s_{i,j}]$$

denotes the $i^{th} + 1$ *cut-formula, where* $1 \leq i \leq n$

Definition 33 (canonical solution σ)
Let H *be a schematic extended Herbrand sequent of* $\forall x F \rightarrow$ *and* $\sigma = [X_i \backslash \lambda \alpha_i . C_i]_{i=1}^n$ *be its canonical substitution as described in 32. The substitution* σ *is then called a* ***solution*** *iff* $V(C_i) \subseteq \{\alpha_i, \ldots, \alpha_n\}$ *and* $H\sigma$ *is a tautology.*

Note that such a sequent H always has a solution σ (Definition 33) w.r.t. to a proof, which forms an extended Herbrand sequent. For a proof consider [9] where also a method, based on resolution, is described which allows to improve such a canonical substitution, but will be omitted in this thesis. As of now we can combine Definition 31 of a schematic extended Herbrand sequent, term sets $U, S_1, \ldots, S_n$ and Definition 33 to retrieve an *extended Herbrand sequent*, where the previously introduced second-order variables $X_1, \ldots, X_n$ successfully got replaced by Lambda-expressions of our *canonical substitution*. Starting from a schematic extended Herbrand sequent

$$\begin{aligned} H = \quad & F[x\backslash u_1], \ldots, F[x\backslash u_m], \\ & X_1(\alpha_1) \supset (\bigwedge_{j=1}^{k_1} X_1(s_{1,j})), \\ & \vdots, \\ & X_n(\alpha_n) \supset (\bigwedge_{j=1}^{k_n} X_n(s_{n,j})) \end{aligned}$$

where the bound variable x in F is already substituted by the individual terms of the first term set U. We proceed by applying the canonical substitution on the schematic extended Herbrand sequent by individually applying it on each subformula, i.e.

$$\begin{aligned} H = \quad & F[x\backslash u_1], \ldots, F[x\backslash u_m], \\ & (X_1(\alpha_1) \supset (\bigwedge_{j=1}^{k_1} X_1(s_{1,j})))[X_1 \backslash \lambda \alpha_1 . C_1], \\ & \vdots \\ & (X_n(\alpha_n) \supset (\bigwedge_{j=1}^{k_n} X_n(s_{n,j})))[X_n \backslash \lambda \alpha_n . C_n] \end{aligned}$$

After substituting all necessary parts in $H\sigma$ we retrieve our extended Herbrand sequent as described in Definition 30 of a particular proof, where all necessary instantiations $U, S_1, \ldots, S_n$ are considered.

But what exactly are those laboriously looking term sets $U, S_1, \ldots, S_n$ and how can we retrieve them from a cut-free proof? Let us recall the definitions of a language, grammar and the notation of a decomposition, respectively.

Reconsider Formula 3.1 of all instantiations to form up a proof for a sequent $\forall x F \rightarrow$, where L is the language or term set comprising exactly all instantiations of F of a cut-free proof. We will later on see that there is a method to efficiently compressing the representation of L through a grammar decomposition and assume it to produce following grammar

$$G = U \circ_{\alpha_1} S_1 \circ_{\alpha_2} \cdots \circ_{\alpha_n} S_n$$

s.t. $L = \mathcal{L}(G)$. This would imply that we could generate out of an arbitrary term set L a grammar decomposition G, compute previously the mentioned canonical substitution and thus transform the Herbrand sequent of a cut-free proof π^* via a schematic extended Herbrand sequent into an extended Herbrand sequent. Furthermore this would lead to the possibility of again building a proof π from this extended Herbrand sequent with n cuts. This very procedure of generating a grammar decomposition out of a language, forming up *exactly* this language is not really efficient, although it is not even necessary. It suffices to build a grammar generating a language $\mathcal{L}(G)$, which merely contains the original term set L, i.e.

$$L \subseteq \mathcal{L}(G)$$

We will later on see that there exist in fact methods for generating a grammar decomposition of this kind, which are able to even go beyond the simple production by simultaneously trying to minimize its amount of rules. This leads to a compressed representation of the generated language and consequently to a possible smaller representation as a proof in **LK**.

3.4 Grammar decomposition

Before we will deal with the process of constructing such a grammar decomposition let us get an overview of the previously described and the remaining steps in Figure 3.3. The method described in [9] compresses a cut-free proof π^* to a proof π containing several cuts within a few steps. We already discussed building a proof with cuts from an extended Herbrand sequent in Section 3.2, as illustrated in step 5 of Figure 3.3 and proceeded by generalizing an extended Herbrand sequent to separate the instantiating terms from the sequent in Section 3.3 as one can see in step 4 in Figure 3.3. In the latter case we saw that we can find a canonical substitution, which can be applied onto a schematic extended Herbrand sequent, to retrieve an extended Herbrand sequent. This substitution, however needs term sets $U, S_1, \ldots, S_n$ for the introduction of n cuts, which remain to be computed. To generate such term sets, we will start from a given cut-free proof π^* and describe needed steps, illustrated in Figure 3.3 steps 1-3, to compute a grammar decomposition, which contains exactly those needed term sets $U, S_1, \ldots, S_n$. We already mentioned in Section 3.1 the possibility to extract particular details necessary to proof a sequent $\forall x F \rightarrow$ from a proof, namely its Herbrand sequent. This extraction method, as shown in Figure 3.3 step 1, returns for an arbitrary cut-free proof π^* its Herbrand sequent and thus enables us to continue generating a language of instantiated terms, which we will later on try to compress by a grammar decomposition.

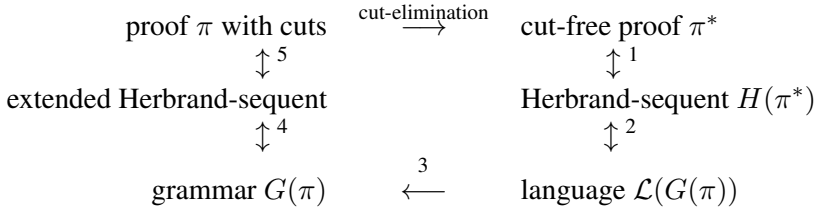

Figure 3.3: Sketch of cut introduction method [9]

Extracting the term set

Since we have extracted the Herbrand sequent $F[x\backslash L] \rightarrow$ out of a cut-free proof π^* by simply reading off all instantiations L of F, we can go on by interpreting L as a language which we will further try to decompose (see Figure 3.3 step 2). Since a Σ_1-sequent possibly contains several quantified formulas, it is necessary to distinguish which term in T originates in which quantified formula and thus we will have to mark them accordingly. This is done by possibly introducing artificial function symbols $f_1, \ldots, f_m$ encapsulating the particular terms in T, where all terms starting with f_i arise from the i^{th} quantified formula in the sequent.

Computing a minimal grammar

The goal of the next step, namely the efficient computation of a grammar decomposition generating at least all term instantiations L (see Figure 3.3 step 3) with a minimal number of production rules, is a quite complex task and can be achieved in different ways. One method is described in [9] and was already implemented in *GAPT*. Since a cut-free proof in **LK** is shaped like a tree and expects various other properties to hold, e.g. the *subformula property* in Theorem 1, the underlying grammar of a term set L of a proof has to fulfill again certain properties. Recall Definition 18 of a $trat$-n grammar, which is an abbreviation for *totally rigid acyclic tree grammar* with n non-terminals and reconsider previous Definitions 14, 15 and 16 from a proof-theoretic point of view. Those laboriously looking properties are necessary to ensure a certain behavior of our soon-to-be computed grammar to fit our proof-theoretic needs. The properties induced by Definition 18 of a *trat-*n grammar are crucial to represent the grammar as a decomposition and thus act as a compressed representation for our needed instantiations of quantified formulas contained in the proof. This means that considering particular terms to be generated through out the grammar via rules $\alpha_i \rightarrow t_1 | \ldots | t_{k_i}$, have to grow in a monotone manner. Note that all non-terminals α_i in an underlying grammar decomposition, will be later on introduced variables fulfilling the *Eigenvariable condition* within our proof with cuts. Therefore we have to demand acyclicity as described in Definition 15, which is necessary since a proof represents a finite struc-

[1] Herbrand sequent extraction [10]
[2] extract language/termset from Herbrand sequent [9]
[3] compute a decomposition/grammar from termset [3]
[4] generating canonical solution and thus an extended Herbrand sequent from a grammar [9]
[5] rebuild proof with computed extended Herbrand sequent [9]

ture and thus contains only finitely many term instantiations. Finally let us reconsider Definition 16, which comprises the characteristics of a substitution. Total rigidity of a grammar ensures that if applying a rule $\alpha_i \to s_j$ to a term t, all occurrences of α_i in t are going to be substituted. This behavior can be found analogously for substituting a variable within a term, e.g.

$$f(g(\alpha_1), \alpha_1)[\alpha_1 \backslash c] = f(g(c), c)$$

As we have all necessary preliminaries defined to proceed by computing a grammar decomposition w.r.t. a given language L, we will start with the first method, which had already been implemented and had the ability to introduce a single cut having a multiple quantified cut-formula.

Grammar decomposition via Δ-Table

The method described in [8] rests upon two essential structures: the Δ-*vector* and the Δ-*table*. Let us start with the definition of the Δ-vector in Definition 34.
Note that in this method termsets are interpreted as termsequences to guide the search for a grammar and prune the search space.

Definition 34 (Δ-vector)
Let $t_1, \ldots, t_n \in L$ be a termsequence, then

$$\Delta(t_1, \ldots, t_n) = \begin{cases} (f(u_1, \ldots, u_m), (s_1, \ldots, s_n)) & \text{if all } t_i = f(t_1^i, \ldots, t_m^i) \text{ and} \\ & \Delta(t_j^1, \ldots, t_j^n) = (u_j, (s_1, \ldots, s_n)) \\ & \forall\, j \in \{1, \ldots, m\} \\ (\alpha, (t_1, \ldots, t_n)) & \text{otherwise} \end{cases}$$

is its computed Δ-vector, where α is a non-terminal.

The Δ-vector of a termsequence is computed by shifting all common leading function symbols to the left. If there exists a position, where this operation is not possible anymore, a new non-terminal α gets introduced, which acts as a placeholder for the remaining rests of the terms. Consider Example 7, which illustrates an application of a Δ-vector to a particular termsequence.

Example 7 (Example application of Δ-vector)
Let $L = (f(g(a), a), f(g(b), b), f(g(c), c))$
*Then $\Delta(L) = (f(g(\alpha), \alpha), (a, b, c))$ represents a **simple** grammar $\{f(g(\alpha), \alpha)\} \circ_\alpha \{a, b, c\}$*

The Δ-vector application performed on a particular termsequence will already result in a simple grammar decomposition $U \circ_\alpha S$. As defined in Definition 21 a trivial grammar, where $U = \{\alpha\}$, can also be a result of the Δ-vector. This happens if there exists no common leading function symbol for the terms $t_i \in L$ within the termsequence. Since the goal of this method is to get more complex grammar decompositions, i.e. $|U| \geq 1$, a second structure is needed, namely the Δ-table.

Definition 35 (Δ-table)
Let L be a termsequence. The Δ-table stores for a generating subset of terms $S \subseteq L$ a list of possible decompositions.

For every subsequence $L' \subseteq L$ the resulting grammar decomposition $\Delta(L') = (U, S)$ is stored in the Δ-table, s.t. $(U, L') \in \Delta_{table}(S)$. The Δ-table comprises all possible decompositions indexed by their generating term subsets S.

The Δ-table will be build iteratively by iterating through all subsequences of the underlying sequence L. Note that the search space (induced by the exponential number of subsequences of L) is in general vast and can be pruned by means of Theorem 2.

Theorem 2
Let T be a set of terms. If $\Delta(T) = (\alpha, T)$ (trivial grammar), then $\Delta(T) = (\alpha, T')$ for every $T' \supset T$.

Proof
Follows from Definition 34 of Δ-vector.

The theorem induces, that if we iteratively build up the subsequences (by stepwise adding a term of $t \in L$ to L' and call Δ-vector) we can skip further computations of a subsequence (i.e. forall $L'' \supset L'$) if we already obtain a trivial decomposition to form up the sequence. In short, if the terms already contained in L' are not sharing any leading function symbols, they will never, even less, when adding further terms from the underlying sequence L. Therefore we can skip further steps for this subsequence and continue by considering other subsequences build up recursively.

Starting from a given termsequence L, subsets $L' \subseteq L$ will be considered and individually decomposed, s.t. $\Delta(L') = (U, S)$. Since for all possible subsequences of L a decomposition will be generated, we achieve a map s.t.

$$\Delta_{table}(S) = \{(U_1, L'_1), \ldots, (U_n, L'_n)\} \tag{3.2}$$

where $\Delta(L'_i) = (U_i, S)$ for all $1 \leq i \leq n$. In other words, we will receive a table which lists for all possible term subsets $S \subseteq L$, their generating parts L'_i regarding their U_i. Since we want to find a minimal grammar w.r.t. the number of rules, we are interested in finding an S in the generated Δ-table s.t. regarding Equation 3.2 following holds

$$\bigcup_{i=1}^{n} L'_i \supseteq L$$

I.e. for a particular S we can take all its grammar decompositions and are able to generate at least the termset L. Usually there is not a uniquely defined S fulfilling this property and we choose the one which is minimal regarding their components (or when speaking in the context of a grammar; its number of rules), i.e.

$$\underset{S}{\operatorname{argmin}} \left\{ |\bigcup_{i=1}^{n} U_i| + |S| \;\middle|\; (U_i, L'_i) \in S \text{ and } \bigcup_{i=1}^{n} L'_i \supseteq L \right\}$$

Note that those minimal grammars have not to be unique. In fact they are, for our proof-theoretic purpose, not even equally good as we will see in Chapter 5. In this method of computing a minimal grammar decomposition all minimal grammars are transformed into proofs. Finally we take the proof with the least number of applied rules.

This method merely introduces one cut into a proof, due to finding a single decomposition of a term set L. To eventually achieve an introduction of several cuts one could call this procedure iteratively, i.e. after a decomposing $L \subseteq (U_i \circ_{\alpha_i} S)$ again execute the algorithm to further decompose the term set S. This will eventually lead to a decomposition

$$L \subseteq (U_1 \circ_{\alpha_1} (U_2 \circ_{\alpha_2} \cdots \circ_{\alpha_{n-1}} (U_n \circ_{\alpha_n} S') \dots))$$

and thus to an introduction of n cuts, as previously described. Note that $(U_n \circ_{\alpha_n} S') = S_n$ and for all $1 \leq i \leq n$ holds $(U_i \circ S_i) = S_{i-1}$, whereas $S_0 = L'$.

Grammar Decomposition via MinCostSAT formulation

In contrast to the previously described method of constructing a minimal grammar in Section 3.4, there exists another one described in [3] which does not generate all possible grammars and merely selects a minimal one. Instead it uses the definition of a normal form for *keys*, which paraphrase a set of terms within a term set. The method requires the generation of a *sufficient set of keys* in normal form. This set is subsequently used to reduce the search for a minimal grammar to the MinCostSAT Problem.

Preliminary definitions First let us focus on Definitions 36 and 37 of keys of a grammar and a language.

Definition 36 (key of grammar)
A term k is called a key *of a grammar G iff it occurs on the right-hand side of a production rule $p \in P_G$.*

Definition 37 (Key of language)
A term k containing non-terminals $\alpha_1, \dots, \alpha_n$ is a key *of a language L if there is a set R (of n-tuples) such that*

$$k \circ_{\alpha_1,\dots,\alpha_n} R = L$$

A key can be used to generate a particular term w.r.t. a term vector as described in Definition 38.

Definition 38 (Term generation)
Let L be a language and k be a key of L containing non-terminals $\alpha_1, \dots, \alpha_n$ and $s_1, \dots, s_n$ be terms, s.t. for a particular term $t \in L$ holds

$$t = k\sigma_{L,k,t}$$

where $\sigma_{L,k,t} = [\alpha_1 \backslash s_1] \dots [\alpha_n \backslash s_n]$. $\sigma_{L,k,t}$ can be rewritten as σ_t if L and k can be inferred from the context. We can denote this substitution by

$$k \circ_{\alpha_1,\dots,\alpha_n} \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix} = t$$

and say that t is produced by k.

Consider Example 8 for a possible term generation based on a language and a key.

Example 8 (Example term generation)
Let $L = \{f(g(a), b), f(g(c), c)\}$ and $k = f(g(\alpha_1), \alpha_2)$, then

$$f(g(\alpha_1), \alpha_2) \circ_{\alpha_1,\alpha_2} \left\{ \begin{pmatrix} a \\ c \end{pmatrix}, \begin{pmatrix} b \\ c \end{pmatrix} \right\} = L$$

induces the existence of the set $R = \left\{ \begin{pmatrix} a \\ c \end{pmatrix}, \begin{pmatrix} b \\ c \end{pmatrix} \right\}$

To efficiently compute those decompositions of a particular language we will have to define also a *normal form* for keys, which will be given in association with *equations*. The validity of an *equation* within a term set and w.r.t. a key is given in Definition 39.

Definition 39 (An equation relative to a key)
Let k be a key of a language L containing non-terminals $\alpha_1, \dots, \alpha_n$. Let $q_0, q_1 \in L$ be terms containing at most non-terminals $\alpha_1, \dots, \alpha_n$.
Then L fulfills an equation $q_0 \cong q_1$ relative to k, iff for all $t \in L$ holds that $q_0\sigma_t \cong q_1\sigma_t$, where σ_t is the corresponding substitution producing t as mentioned in 38.
Furthermore a term t falsifies *an equation relative to k if $q_0\sigma_t \ncong q_1\sigma_t$ for some t.*
Again, if k is clear from the context, we merely write that L satisfies/falsifies an equation.

Let us consider Definition 39 of an equation in practice in Example 9.

Example 9 (Example of an equation)
Let $L = \{f(g(c), g(c)), f(g(d), g(d))\}$ and $k = f(\alpha_1, g(\alpha_2))$.
We obtain the following substitutions
$\sigma_{f(g(c),g(c))} = [\alpha_1 \backslash g(c)][\alpha_2 \backslash c]$
$\sigma_{f(g(d),g(d))} = [\alpha_1 \backslash g(d)][\alpha_2 \backslash d]$
Thus L fulfills the equation $\alpha_1 \cong g(\alpha_2)$.

Definition 40 (normal form of a key)
Let k be a key of a term set L containing non-terminals $\alpha_1, \dots, \alpha_l$, where $l \in \mathbb{N}$. Then k is in normal form to L and $\alpha_1, \dots, \alpha_l$ iff
for all satisfied equations $q \cong \alpha_i$ of L, where $1 \leq i \leq l$ and q is a either a subterm of k or a closed term, q equals α_i. We say that k is in normal form to L exactly if k is in normal form relative to L and to all non-terminals occurring in k.

Definition 40 above induces that every non-terminal α_i within a key does not coincide with another's non-terminal α_j underlying termstructure, where $i \neq j$. Furthermore a non-terminal α_i does not comprise a single subterm q, but a set of subterms. Note that the normal form of a key k relative to a term set L does not require terms within L to have different leading function symbols. For example consider the language $L = \{f(g(a), a), f(g(b), b)\}$ and following term $f(\alpha_1, \alpha_2)$, which is already a key although all terms comprised by α_1 have coinciding function head symbols, i.e.

$$f(\alpha_1, \alpha_2) \circ_{\alpha_1,\alpha_2} \left\{ \begin{pmatrix} g(a) \\ a \end{pmatrix}, \begin{pmatrix} g(b) \\ b \end{pmatrix} \right\} = L$$

Furthermore consider the language $L = \{f(g(c), g(c)), f(g(d), g(d))\}$ and the key $f(\alpha_1, g(\alpha_2))$ from Example 9, which is not a key in normal form, since it fulfills the equation $\alpha_1 \cong g(\alpha_2)$. This fact becomes clearer when considering its decomposed representation

$$f(\alpha_1, g(\alpha_2)) \circ_{\alpha_1,\alpha_2} \left\{ \begin{pmatrix} g(c) \\ c \end{pmatrix}, \begin{pmatrix} g(d) \\ d \end{pmatrix} \right\} = L$$

In contrast to above key, consider the term $f(\alpha_1, \alpha_1)$ which fulfills no equation and thus is a key in normal form

$$f(\alpha_1, \alpha_1) \circ_{\alpha_1} \left\{ \big(g(c)\big), \big(g(d)\big) \right\} = L$$

Computing a sufficient set of keys Let us focus on the computation of a set of keys for a language L which is proven to suffice for constructing a grammar G of minimal size [3]. We denote $S_{L,n}$ as a *sufficient set of keys* in Definition 41, which compresses a language L by a *trat-n* grammar (see Definition 18).

Definition 41 (sufficient set of keys $S_{L,n}$)
Let L be a term set and $n \in \mathbb{N}$.
A set $S_{L,n}$ is called a sufficient set of keys of n non-terminals for L iff following holds:
Let $m \in \mathbb{N}$ be the size of a minimal trat-n grammar G s.t. $L \subseteq \mathcal{L}(G)$.
Then there is a trat-n grammar G' with $L \subseteq \mathcal{L}(G')$ of size m containing only keys in $S_{L,n}$.

The computation of such a sufficient set of keys for a term set L and an integer n is defined in [3]. The method is based upon Algorithm 1, which takes a term set L and an integer n as input and returns the mentioned sufficient set of keys. Note that following algorithms are written in a descriptive way in [3] and their implementation will result in an adapted version, which will be discussed later on in Chapter 4.

The algorithm for a sufficient set of keys (see Algorithm 1) computes all keys in normal form for each subset $L' \subseteq st(L)$ of subterms of L, where the size of each subset is bounded by $n + 1$. This bound is crucial for the runtime of the algorithm. Since n denotes the number of non-terminals, i.e. the number of cuts to be introduced it suffices to generate just keys in normal form compressing a set of subterms of L of size $n + 1$. The proof for this statement among others can be found in [3].

The function NORMFORM computes a set of keys in normal form for a set of subterms L' and an integer n, which defines L'. To fully understand Algorithm 3 of NORMFORM, we have to define some important notions.

Algorithm 1 Generate a sufficient set of keys for a language

function SUFFKEYS(L: language, n: number of non-terminals)
  $S_{L,n} = \{\}$ ▷ initialize output set
  for each $L' \subseteq st(L)$, where $1 \leq |L'| \leq n+1$ **do**
    $S_{L,n} = S_{L,n} \cup$ NORMFORM(L', n) ▷ see Algorithm 3 of normal form
  end for
  return $S_{L,n}$
end function

Algorithm 2 Compute a key in normal form

function GDV(L': set of terms)
  $k = \Delta(L')$
  for all pairs α_i, α_j, where $i < j$, of k **do**
    if $\alpha_i \cong \alpha_j$ is satisfied **then**
      $k = k[\alpha_j \backslash \alpha_i]$ ▷ purge redundant α_j
    end if
  end for
end function

First let us consider Algorithm 2 of the *generalized delta vector*, which describes the generation of a key regarding a particular term set. It uses the Δ-vector (see Definition 34), which merely performs a left shift until no matching function head symbols can be found. Afterwards we have to purge redundant non-terminals, i.e. α_j which describes the same subterm set as another α_i, where $i < j$. Another important structure is the *characteristic Partition* of a term.

Definition 42 (characteristic Partition $\mathcal{P}$)
Let k be a term and P the set of its positions p s.t. $k|_p$ contains a variable.
Then the characteristic Partition $\mathcal{P}$ of k is induced by following equivalence relation

$$p_0 \sim p_1 \Leftrightarrow k|_{p_0} = k|_{p_1} \text{ where } p_0, p_1 \in P$$

Consider Definition 42 of a *characteristic Partition*, which describes a list of positions within the term, which refer to equal subterms. Let us consider following Example 10 to get a better view on the definition of a characteristic Partition.

Example 10 (Example for a characteristic Partition)
Consider the term $k = f(g(\alpha_1), g(\alpha_1), \alpha_2)$
Then $charPartition(k) = \{\{\epsilon\}, \{1,2\}, \{1.1, 2.1\}, \{3\}\}$
$\{\epsilon\}$ *represents the root*
$\{1,2\}$ *is induced by* $g(\alpha_1) = g(\alpha_1)$
$\{1.1, 2.1\}$ *is induced* $\alpha_1 = \alpha_1$
$\{3\}$ *is induced by the single occurrence of* α_2

Algorithm 3 Compute the set of all keys in normal form

function NORMFORM(L': set of terms of, n: number of non-terminals)
  $N = \{\}$ ▷ initialize output set
  k_{init} =GDV(L') ▷ compute a key via gdv
  $k = k_{init}[\overline{\alpha}\backslash\overline{\beta}]$ ▷ substitute all non-terminals
  $\mathcal{P}$ =CHARPARTITION(k)
  for all ordered lists $(P_1, \ldots, P_m)$, where $P_i \in \mathcal{P} \cup \{\emptyset\}$, $1 \leq m \leq n$ **do**
    $k_{new} = k$
    for $1 \leq i \leq n$ **do**
      for all positions $p \in P_i$ **do**
        if p is a position of k_{new} **then**
          $k_{new}|_p = \alpha_i$ ▷ Replace term at p with non-terminal
        end if
      end for
    end for
    if there exists no β_i in k_{new} ($i \in \mathbb{N}$) **then**
      $N = N \cup \{k_{new}\}$
    end if
  end for
  return N
end function

As we have summarized all necessary parts for computing all keys in normal form we proceed by defining Algorithm 3. It begins with initializing the output set N and continues by substituting all occurring non-terminals α_i by syntactically different non-terminals β_i. This is important for the next step, where we successively try to generate keys by replacing just those positions defined by the key's characteristic Partition. So for all ordered lists of size $1 \leq m \leq n$ of $\mathcal{P}$ we try to substitute again the occurring terms at positions $p \in P_i$ by non-terminals α_i. If and only if all previously introduced non-terminals β_i could be removed the term is called a key in normal form and will be added to the output set. This leads to a set of keys in normal form w.r.t. the input set L' of subsets of $st(L)$, i.e.

$$u \in \text{NORMFORM}(L) \Rightarrow u \text{ is in normal form rel. to } L$$

GDV(L') returns a key relative to L', where afterwards all β_i occurrences are replaced by corresponding α_i occurrences, which results in a key k. The characteristic partition of k is nothing less than all positions of subterms of k satisfying an equation, i.e. $\mathcal{P} = \{P_1, \ldots, P_l\}$ is a set of position sets of equal subterms of k. As we are looking at all permutations of $\mathcal{P}$ of size at most m, i.e. $(P_{i_1}, \ldots, P_{i_o})$, where $1 \leq o \leq m$, we generate a sequence of positions which we try to replace by corresponding non-terminals α_i. When we take a position set into account, we will replace the subterms at all positions (if possible) with a single α_i. Since the only subterms $q_0, q_1 \in st(k)$ which satisfy an equation are found in the same position set $k|_{p_0} = q_0$ and $k|_{p_1} = q_1$, we will not be able to satisfy them unless they denote the same subterms, i.e. $q_0 = q_1$.

Thus the resulting keys produced by NORMFORM(L') have to be in normal form relative to L'. The proof can be found in [3].

In Example 11 we illustrate the behavior of Algorithm 3 for a particular set of subterms of a language L.

Example 11 (Example computation of keys in normal form)
Let $n = 2$,
$L = \{f(a,a), f(b,b), f(g(a),g(a)), f(g(b),g(b)), f(g(a),g(b)), f(g(b),g(a))\}$ *a language and* $L' = \{f(g(a),g(a)), f(g(b),g(b)), f(g(a),g(b))\}$ *a subset of L,*
i.e. $L' \subseteq L$. *Then* $normform(L', n)$ *is computed as follows.*
$k_{init} = f(g(\alpha_1), g(\alpha_2))$ *and thus* $k = f(g(\beta_1), g(\beta_2))$
Next the characteristic partition of k *is generated*

$$\mathcal{P} = \{\{\epsilon\}, \{2.1\}, \{1\}, \{2\}, \{1.1\}\}$$

Then we generate a set $\mathbb{P}$ *of all ordered lists of elements in* $\mathcal{P}$ *of at most size* $n = 2$

$$\begin{aligned}\mathbb{P} = \{ \quad & (\epsilon), (2.1), (1), (2), (1.1), (2, 1.1), (1.1, 2), (1, 2), (2, 1), (1, 1.1), (1.1, 1), \\ & (2.1, 1), (1, 2.1), (2.1, 2), (2, 2.1), (2.1, 1.1), (1.1, 2.1), (\epsilon, 2.1), (2.1, \epsilon), \\ & (\epsilon, 1), (1, \epsilon), (\epsilon, 2), (2, \epsilon), (\epsilon, 1.1), (1.1, \epsilon) \qquad \}\end{aligned}$$

For the iteration over above set of permuted lists we will give three particular examples of lists, where further lists can be handled analogously.

- *Consider the case* $P_1 = (\epsilon)$, *where we perform the substitution on the whole key, s.t.* $k_{new} = \alpha_i$. *Since neither* β_1 *nor* β_2 *are contained in* k_{new} *any longer, we can add the term to the resultset* N.
- *Let us observe the case for* $(1.1, 2)$ *(i.e.* $P_1 = 1.1$ *and* $P_2 = 2$*), where two substitutions are performed on mentioned positions s.t.* $k_{new} = f(g(\alpha_1), \alpha_2)$. *As no* β_j *occurs in* k_{new} *anymore, we can add it to the resultset* N.
- *Finally consider following case of* $(2, 2.1)$ *(i.e.* $P_1 = 2$ *and* $P_2 = 2.1$*), where at first the substitution* $k|_2 = \alpha_1$ *is performed, s.t.* $k = f(g(\beta_1), \alpha_1)$. *Secondly we try to substitute* $k|_{2.1}$, *but since this position does not exist any longer, we skip this substitution. On the final check we observe that not all variable occurrences* β_j *have been substituted, i.e.* $k = f(g(\beta_1), \alpha_1)$. *Thus the key is not added to the output set.*

By applying above exemplary behavior to $\mathbb{P}$, *we successfully generated all normal forms of* $f(g(\beta_1), g(\beta_2))$, *namely*

$$\begin{aligned}N = [\quad & \alpha_1, f(g(\alpha_2), \alpha_1), f(g(\alpha_1), \alpha_2), f(\alpha_1, \alpha_2), f(\alpha_2, \alpha_1), f(\alpha_2, g(\alpha_1)), \\ & f(\alpha_1, g(\alpha_2)), f(g(\alpha_2), g(\alpha_1)), f(g(\alpha_1), g(\alpha_2)), \alpha_1, \alpha_2, \alpha_1, \alpha_2, \alpha_1, \alpha_2, \alpha_1, \alpha_2 \quad]\end{aligned}$$

Note: The set contains $f(g(\beta_1), g(\beta_2))$ *itself, namely* $f(g(\alpha_1), g(\alpha_2))$, *thus this key is already in normal form. which results in a set*

$$\begin{aligned}N' = \{ \quad & \alpha_1, f(g(\alpha_2), \alpha_1), f(g(\alpha_1), \alpha_2), f(\alpha_1, \alpha_2), f(\alpha_2, \alpha_1), f(\alpha_2, g(\alpha_1)), \\ & f(\alpha_1, g(\alpha_2)), f(g(\alpha_2), g(\alpha_1)), f(g(\alpha_1), g(\alpha_2)), \alpha_2 \qquad \}\end{aligned}$$

Note: N' is not necessarily a complete enumeration of keys for L, since we generated it merely for a particular subset $L' \subseteq L$. To retrieve an exhaustive set we have to refer to Algorithm 1, which generates the union of all normal forms of subsets $L' \subseteq L$ of size at most $n+1$.

As we can see in Algorithm 1, we are able to generate a sufficient set of keys of a particular term set L and proceed by generating a *MinCostSAT* formulation based on this very set. This formulation can be taken as a basis for a *Linear Programming* or *partial weighted MaxSAT* formulation and thus can be used to solve it via external tools such as an arbitrary MaxSAT solver or a Simplex solver.

Reduction to MinCostSAT

Definition 43 (i-rest)
Let $G = (S, N, \alpha_0, P)$ be a $trat$-n grammar and

$$\delta = \alpha_0, t_1, t_2, \ldots, t_m, q$$

be a derivation of a term $q \in L$, where $V(t_i) \subseteq \{\alpha_{i+1}, \ldots, \alpha_n\}$.
Let p be a position in t_{i-1} (for $i \geq 1$) s.t. $t_{i-1}|_p = \alpha_i$.
Then $t_i|_p$ is called an i-rest of the rigid derivation δ.

Consider Definition 43 of an i-rest, which describes the concept of substituting subterms by a variable within a derivation of a term.

For a better understanding consider Example 12 of an i-rest.

Example 12 (Example of an i-rest)
Let $G = (S, N, \alpha_0, P)$ be a $trat$-2-grammar and

$$\delta = \alpha_0, f(\alpha_1, \alpha_1), f(g(\alpha_2), g(\alpha_2)), f(g(a), g(a))$$

be a derivation of the term $f(g(a), g(a))$.
Then $g(\alpha_2)$ is a 1-rest and a is a 2-rest of δ.

We proceed by defining the essential structure of the MinCostSAT formulation by defining the two types of propositional variables occurring in the formulation in Definition 44.

Definition 44 (propositional Variables for MinCostSAT)
Let L be a language, $n \in \mathbb{N}$ be the given number of non-terminals (which is intentionally bound by $n \leq |L|$) and $S_{L,n}$ the sufficient set of keys of L. Then we consider for our MinCostSAT formulation following two types of propositional variables.

1. *$x_{i,k}$, which represents the potential production rule $\alpha_i \to k$, where $k \in S_{L,n}$ and $0 \leq i \leq n$*

2. *$x_{t,i,q}$, which states that $t \in st(L)$ is a potential i-rest of a derivation of $q \in L$ and $1 \leq i \leq n$*

For a MinCostSAT reduction we have to formalize essential parts of a grammar, i.e. how a decomposition is correctly built or which production rules imply the existence of other production rules. The following formulas will be necessary to ensure this. We start with Definition 45 of $R_{L,S}(q)$, which merely forbids grammars, where we have to derive more than one i-rest, i.e. non-rigid grammars.

Definition 45 ($R_{L,S}(q)$)
Let L be a term set and S the corresponding sufficient set of keys, containing at most the non-terminals $\alpha_1, \ldots, \alpha_n$ ($n > 0$). Assume that $q \in L$, then we have

$$R_{L,S}(q) \Leftrightarrow \bigwedge_{t_0,t_1 \in st(\{q\}), t_0 \neq t_1, 1 \leq i \leq n} \neg x_{t_0,i,q} \vee \neg x_{t_1,i,q}$$

Consider Example 13, where the formulation of $R_{L,S}(q)$ for a particular term is described.

Example 13 (Example of $R_{L,S}(q)$)
Let $q = f(s^2(0))$ be a term in a term set L, S be its sufficient set of keys and $n = 1$.
Then the subterms of q are

$$st(\{q\}) = \{0, s(0), s^2(0), f(s^2(0))\}$$

We generate the formula $R_{L,S}(q)$ according to Definition 45

$$\begin{aligned} R_{L,S}(f(s^2(0))) = & \\ (\neg x_{0,\alpha_1,f(s^2(0))} \vee \neg x_{s(0),\alpha_1,f(s^2(0))}) & \wedge(\neg x_{0,\alpha_1,f(s^2(0))} \vee \neg x_{s^2(0),\alpha_1,f(s^2(0))}) \wedge \\ (\neg x_{0,\alpha_1,f(s^2(0))} \vee \neg x_{f(s^2(0)),\alpha_1,f(s^2(0))}) & \wedge(\neg x_{s(0),\alpha_1,f(s^2(0))} \vee \neg x_{s^2(0),\alpha_1,f(s^2(0))}) \wedge \\ (\neg x_{s(0),\alpha_1,f(s^2(0))} \vee \neg x_{f(s^2(0)),\alpha_1,f(s^2(0))}) & \wedge(\neg x_{s^2(0),\alpha_1,f(s^2(0))} \vee \neg x_{f(s^2(0)),\alpha_1,f(s^2(0))}) \end{aligned}$$

Note: We merely have to consider unordered pairs of subterms.

In Definition 46 $D_{L,S}(t,l,q)$ we consider all possible rules x_{l,k_j}, i.e. $\alpha_l \to k_j$, treating keys k_j of t. Those keys may not contain non-terminals α_i, where $i \leq l$, but need not contain all non-terminals α_j for $j > l$. Thus we have to fix a notation, s.t. we can iterate through all non-terminals occurring in k_j, i.e. $\alpha_{i_1}, \ldots, \alpha_{i_m}$. The formula $D_{L,S}(t,l,q)$ itself is listing all possible rules applicable in a derivation, for deriving q from t, where $1 \leq l \leq n$. It ensures that in case a rule is chosen, its induced i-rests have to be fulfilled accordingly, leading to a formulation of a dependency graph. This dependencies merely ensure that every derivation of the wanted $trat\text{-}n$ grammar is terminating in a terminal term $t \in L'$, where $L' \subseteq L$.

Definition 46 ($D_{L,S}(t,l,q)$)
Let L be a language and S the corresponding sufficient set of keys containing at most the non-terminals $\alpha_1, \ldots, \alpha_n$ ($n > 0$). Assume $t \in st(L)$, $0 \leq l \leq n$ and $q \in L$. Furthermore assume that $k_1, \ldots, k_s \in S$, containing at most non-terminals α_i where $i > l$, are keys of t. Assume $1 \leq j \leq s$. Let $\alpha_{k_j}^{i_1}$ denote the non-terminal with the smallest index in k_j, $\alpha_{k_j}^{i_2}$ the one with

the second smallest index in k_j and so on. Let $\alpha_{k_j}^{i_m}$ be the non-terminal with the largest index occurring in k_j. Then following decomposition

$$k_j \circ_{\alpha_{k_j}^{i_1},\alpha_{k_j}^{i_2},\ldots\alpha_{k_j}^{i_m}} \begin{pmatrix} r_{k_j}^{i_1} \\ r_{k_j}^{i_2} \\ \vdots \\ r_{k_j}^{i_m} \end{pmatrix}$$

where $r_{k_j}^{i_1}, r_{k_j}^{i_2}, \ldots, r_{k_j}^{i_m}$ are the corresponding rests of the non-terminals in k_j, is represented as the following formula

$$D_{L,S}(t,l,q) \Leftrightarrow \bigvee_{1\leq j\leq s} x_{l,k_j} \wedge x_{r_{k_j}^{i_1},i_1,q} \wedge \cdots \wedge x_{r_{k_j}^{i_m},i_m,q} \tag{3.3}$$

$C_{L,S}(q)$ subsumes previously defined formulations, which we will consider step by step to get a closer look on its behavior.

Definition 47 ($C_{L,S}(q)$)
Let L be a term set and S the corresponding sufficient set of keys in normal form containing at most non-terminals $\alpha_1, \ldots, \alpha_n$ (for some $n > 0$). Furthermore let $q \in L$. Then

$$C_{L,S}(q) \Leftrightarrow \big(\bigwedge_{t\in st(\{q\}),1\leq i\leq n} x_{t,i,q} \rightarrow D_{L,S}(t,i,q)\big) \wedge D_{L,S}(q,0,q) \wedge R_{L,S}(q)$$

The first formula in the topmost conjunction, i.e. $\big(\bigwedge_{t\in st(\{q\}),1\leq i\leq n} x_{t,i,q} \rightarrow D_{L,S}(t,i,q)\big)$, states that every chosen i-rest of a term t deriving a term $q \in L$ appearing on the left hand-side of a $D_{L,S}$ implication, requires the underlying formula $D_{L,S}(t,i,q)$ to hold. This means that the remaining i-rest has to be further processed by choosing additional rules induced by $D_{L,S}(t,i,q)$ to eventually end up in a terminal term. The second part, namely $D_{L,S}(q,0,q)$, ensures that the term $q \in L$ has to be derived, whereas all non-terminals α_i ($i \geq 0$) are admitted. Finally in the third part $R_{L,S}(q)$ we ensure two different rests to be mutually exclusive within the grammar for a term $q \in L$, otherwise we would allow ambiguous grammars.

Definition 48 ($C_{L,S}$)
The formula $C_{L,S}$ is defined by the conjunction of formulas $C_{L,S}(q)$, for all $q \in L$.

$$C_{L,S} \Leftrightarrow \bigwedge_{q\in L} C_{L,S}(q)$$

The intended meaning of $C_{L,S}$ is that for every term $q \in L$ a derivation has to be generated according to the previously defined formulas, which induce a dependency structure representing a guideline for the grammar. The size of the formula $C_{L,S}$ is polynomially bounded by the input L, S (see [3]). For a MinCostSAT instance it remains to assign costs c_j to every propositional variable x_j. We simply assign all variables of the form $x_{i,k}$ a cost $c_j = 1$ and every other variable

$x_{t,i,q}$ a cost $c_j = 0$. This induces that the i-rest variables are only responsible for the correctness of a resulting decomposition and will be not affecting the overall costs. In contrast the costs of variables $x_{i,k}$ play an important role in the MinCostSAT formulation, due to the fact that their amount is minimized by definition, s.t. we obtain a minimal grammar, w.r.t. the amount of rules.

We will later on see in Section 4.2 how exactly this instance of MinCostSAT is transformed to an instance of *partial weighted MaxSAT* in order to solve it.

As we may not retrieve directly a grammar G, but an interpretation of $C_{L,S}$, we have to transform this into a grammar. This is achieved in a straight forward manner, namely by ignoring all propositional variables $x_{t,i,q}$ and simply creating rules $\alpha_i \to k$ for every variable $x_{i,k}$, where $E(x_{i,k}) = true$. This leads to a grammar G with a rule set

$$\begin{aligned} P_G = \{ \quad \alpha_0 \quad &\to k_{(0,0)} | \dots | k_{(0,m_0)} \\ \vdots \quad &\to \vdots \\ \alpha_n \quad &\to k_{(n,0)} | \dots | k_{(n,m_n)} \\ \} \end{aligned}$$

Since α_0 is the axiom or starting non-terminal, we can transform this grammar into a, for our purposes, more intuitive notation of a decomposition.

$$\{k_{(0,0)}, \dots, k_{(0,m_0)}\} \circ_{\alpha_1} \cdots \circ_{\alpha_n} \{k_{(n,0)}, \dots, k_{(n,m_n)}\}$$

Finally we can use this grammar G to generate a canonical substitution, which is then applied to an according *schematic extended Herbrand sequent* to obtain an *extended Herbrand sequent*. Thus we can us the latter to build a proof with several introduced cuts as previously described.

CHAPTER 4

Implementation

Before we start to discuss the implementation of the method in [3], we again need to fix certain notations and define some auxiliary functions and data structures which we are going to use throughout the implementation.

4.1 Preliminaries

First let us define a unification operator $\stackrel{\circ}{=}$ in Definition 49, which abbreviates the procedure of merging to sets or adding elements to a set.

Definition 49 (Unification operator for sets $\stackrel{\circ}{=}$)
Let S_1, S_2 be arbitrary sets, then we write

$$S_1 \stackrel{\circ}{=} S_2$$

as an abbreviation of

$$S_1 = S_1 \cup S_2$$

i.e. adding all elements $x \in S_2$ to the set S_1.

Another term needed for our implementation is that of a *rest fragment*, which is given in Definition 50. Since we are going to replace subterms by a variable at particular positions within a term, and those eventually contain other variables we have to define the terms in between as rest fragments.

Definition 50 (Rest fragment)
Let t be a term, p an arbitrary position in t and $t|_p = t'$ the corresponding subterm in t. If a replacement of t' by α_i is performed s.t. $t|_p = \alpha_i$, then t' is called a rest fragment of t w.r.t. α_i, possibly containing variables $\beta_1, \ldots, \beta_m$.

Example 14 (Example of a rest fragment)
Let $k = f(g(\beta_1), \beta_2)$ *be a key and*

$$\overline{r} = \left\{ \begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} c \\ d \end{pmatrix} \right\}$$

its rests abbreviating a term set

$$L = \{f(g(a), b), f(g(c), d)\}$$

For all terms $t \in L$ *consider following replacement* $t|_1 = \alpha_1$*, where* $g(\beta_1)$ *represents its rest fragment.*

Consider the following list of data structures in Figure 4.1, which were used in the implementation. Note that the algorithm was developed within a class and mentioned data structures are members of such, thus we can assume them to be available at any time.

1. *termMap* $\hat{=}$ Map storing for each (sub-)term its individual index
2. *termIndex* $\hat{=}$ a progressively incremented index for newly introduced terms
3. *keyList* $\hat{=}$ List representing a sufficient set of keys
4. *keyIndexMap* $\hat{=}$ Map storing for each key its index in keyList
5. *keyMap* $\hat{=}$ Map storing for each term a list of indexes of keys, which produce the particular term
6. *decompMap* $\hat{=}$ Map storing for each key a set of its produced terms
7. *propRules* $\hat{=}$ Map storing for each variable $x_{i,k}$ of the MinCostSAT formulation a tuple of the form (i, k)
8. *propRests* $\hat{=}$ Map storing for each variable $x_{t,i,q}$ of the MinCostSAT formulation a tuple of the form (t, i, q)

Figure 4.1: members of the *TreeGrammarDecomposition* class

Let us take a look on above data structures in detail, which are crucial to keep track of processed terms and keys. The MinCostSAT formulation described in Chapter 3 assumes the different propositional variables to be generated w.r.t. terms, keys and non-terminals. Since it is not a wise move to use the very terms, keys and non-terminals in the subscripts to identify a propositional variable, we will store them into corresponding data structures and assign each of them a unique index, which are used instead. Note that all data structures are assumed empty and all numeric variables are assumed to be 0. Let us start with $termMap$ (Item 1) which maps a term onto its unique index. To handle this data structure appropriately consider the function $addToTermMap$ in Algorithm 4, maintaining $termMap$ by possibly adding a new

term to it if it is not already contained. In case the new term t is already available in $termMap$, it merely returns the corresponding index, otherwise a fresh index is created (through incrementing $termIndex$; see Item 2), gets assigned to t in $termMap$ and is returned.

Algorithm 4 adding a term t to the $termMap$

```
1: function AddToTermMap(t: term)
2:     if t ∉ 𝔻(termMap) then
3:                                          ▷ where 𝔻(·) is the domain
4:         termMap(t) = termIndex           ▷ see Item 2
5:         termIndex = termindex + 1
6:     end if
7:     return termMap(t)
8: end function
```

Next let us deal with $keyList$ (Item 3) representing the prospective sufficient set of keys, where each key gets implicitly assigned an index based on its position in the list. For retrieving the index of a particular key within the $keyList$ we refer to $keyIndexMap$, which maps each key onto its corresponding index in $keyList$. In Item 5 we declare the $keyMap$, which stores for a term $t \in L$ a list of indexes of keys $k \in keyList$ producing it. Since we will need to access the data in the opposite direction, i.e. retrieving for a given key its produced terms we define $decompMap$ in Figure 4.1 Item 6. Items 7 and 8 will contain all propositional variables which were generated by the algorithm mentioned in Definition 44, whereas *rule* variables and *rest* variables will be stored separately. Furthermore we will need additional auxiliary functions defined on arbitrary sets. First consider Algorithm 5 which is able to generate from a given set S and an integer $n \geq 1$ all subsets of S of size n. In short the function will be called recursively with a decreasing set S', which differs from S by one element x and both cases x will be added to all generated subsets of the power set or not. The counter i merely keeps track of the amount of already taken elements to avoid violating the upper bound n.

Algorithm 5 generate all subsets of S of size n

```
1: function GENSETS(S: set, i: counter, n: size upper bound)
2:     switch S do
3:         case empty
4:             return {}
5:         case i + 1 < n
6:             let x ∈ S be an arbitrary element
7:             S' = S\{x}
8:             T = GENSETS(S', i + 1, n)
9:             T' = {}                                   ▷ init set T'
10:            for T in T do
11:                T' ≙ {{x} ∪ T}                        ▷ add x to all sets T ∈ T
12:            return GENSETS(S', i, n) ∪ T'
13:        case i + 1 ≥ n
14:            let x ∈ S be an arbitrary element
15:            S' = S\{x}
16:            return GENSETS(S', i + 1, n) ∪ {{x}}
17: end function
```

The function GENSETS is used solely in Algorithm 6 generating for a set S and an upper bound n all subsets of S of at most size n. This is achieved due to iterative calls of $genSets$ for $1 \leq i \leq n$.

Algorithm 6 generate all subsets of S of size $\leq n$

```
1: function BOUNDEDPOWER(S: set, n: size upper bound)
2:     P = {}                                            ▷ init bounded power set
3:     for i in 1, ..., n do
4:         P ≙ GENSETS(S, 0, i)
5:     return P
6: end function
```

Another important function for processing sets is $diagCross$ defined in Algorithm 7, which generates for a given set S all its subsets of size 2 and returns them as a set of tuples. This may seem artificial, but will be of use for our implementation. We achieve this, by combining every element x with all other elements left in $S' = S\backslash\{x\}$ and proceed by recursively call diagCross for the rest S'.

Algorithm 7 generate all subsets of S of size 2 as tuples

```
1: function DIAGCROSS(S: set)
2:     if S = {} then
3:         return {}
4:     else
5:         let x ∈ S be an arbitrary element
6:         S' = S\{x}
7:         R = {}
8:         for y in S' do
9:             R ≙ (x, y)
10:        return R ∪ DIAGCROSS(S')
11:    end if
12: end function
```

Another function used within the algorithm checks for a given term t if $\overline{r}$ is a rest w.r.t. k. We refer to Definition 38 describing a substitution of variables $\alpha_{i_1}, \ldots, \alpha_{i_m}$ with corresponding terms $r_{i_1}, \ldots, r_{i_m}$, s.t. we obtain

$$k[\alpha_{i_1} \backslash r_{i_1}] \ldots [\alpha_{i_m} \backslash r_{i_m}]$$

The function $isRest$ merely applies the above mentioned substitution where all variables $\alpha_{i_1}, \ldots, \alpha_{i_m}$ (ordered by their individual index) in k are substituted by their corresponding rests, which are abbreviated as $\overline{r}$. Finally we check if $t = k[\alpha_{i_1} \backslash r_{i_1}] \ldots [\alpha_{i_m} \backslash r_{i_m}]$ and return $true$ if they are equal, i.e.

$$k \circ_{\alpha_{i_1},\ldots,\alpha_{i_m}} \begin{pmatrix} r_{i_1} \\ \vdots \\ r_{i_m} \end{pmatrix} = t$$

or false otherwise.

Algorithm 8 check if $\overline{r}$ is a rest of t w.r.t. k

```
1: function ISREST(t: term, k: key, r̄: rest)
2:     let α_{i_1}, ..., α_{i_m} be non-terminals of k sorted by index
3:     let r_{i_1}, ..., r_{i_m} = r̄ be the potential corresponding rests
4:     return t ≙ k[α_{i_1}\r_{i_1}] ... [α_{i_m}\r_{i_m}]
5: end function
```

Consider Algorithm 9, which calculates for a given term t its *characteristic partition* as described in Definition 42. We start from a set of all positions in t and an empty characteristic partition. Next we iterate through P and collect for each position p_i, all positions $p_j \in P$, s.t. their denoted terms are syntactically identical, i.e. $t|_{p_i} = t|_{p_j}$. Afterwards we remove them from P and add a new partition containing the collected positions to the characteristic partition. Finally we return the characteristic partition $\mathcal{P}$ which encapsulates all position partitions denoting syntactically identical terms.

Algorithm 9 calculate characteristic Partition $\mathcal{P}$ for a term t

```
1: function CALCCHARPARTITION(t: term)
2:     P = {}                                   ▷ init characteristic partition
3:     let P be the set of all positions in t
4:     for p_i in P do
5:         S = {p_j | t|_{p_i} = t|_{p_j}, p_j ∈ P} ∪ {p_i}   ▷ add all corr. positions to partition
6:         P = P\S                              ▷ remove positions S from P
7:         P ≙ {S}                              ▷ add partition to characteristic partition
8:     return P
9: end function
```

4.2 The TreeGrammarDecomposition algorithm

As we have defined all necessary parts for describing the *TreeGrammarDecomposition* method [17] we proceed by successively describing all its intermediate steps for generating a minimal $trat$-n-grammar G out of a language L and an integer n.

Calculating a sufficient set of keys

Before we proceed by defining the computation of a sufficient set of keys, we have to define a method (Algorithm 10) which computes for a given term set L' and an integer $n \geq 1$ all keys in normal form. After initializing a set collecting the soon-to-be generated keys in Line 2, we call in Line 3 the generalized delta vector for L', which had already been implemented in *GAPT* and was described in Algorithm 2 (see [8]). The mentioned method returns a key k containing non-terminals $\alpha_0, \ldots, \alpha_m$ and all rests $\overline{r} = r_0, \ldots, r_m$, where $m < n$. Since the implementation of gdv in *GAPT* returns a key containing non-terminals starting with and index $i = 0$, we have to increment each index to retrieve a valid grammar where α_0 is its startsymbol. Simultaneously we will rename all variables s.t. we replace every occurring variable α_i in k by a syntactical different variable β_{i+1} in Line 4. Line 5 contains a call of Algorithm 9 where the characteristic partition $\mathcal{P}$ for k is computed. Algorithm 6 was also defined in the preliminaries of this chapter and is called in Line 6, where we obtain for our previously calculated characteristic partition $\mathcal{P}$ all of its permutations of size at most n. This bound n is crucial for the runtime and space bounds of the algorithm, namely operating in a polynomial order, rather than in an exponential one. Before we will see, why this boundary can be defined, let us take a look on the loop in Line 7, which merely iterates through all permutations of size at most n. In the next Lines 8 - 17 we try to replace successively for each position set $P_i \in \mathcal{P}'$ all positions $p_j \in P_i$ with a variable α_i and thus to eliminate all previously introduced variables β_l comprised in terms replaced by α_i.

Algorithm 10 Compute normal form of a term set

1: **function** NORMFORM(L': term set, n: Integer)
2: init $keys$ ▷ final list of keys
3: $(k, \overline{r}) = gdv(L')$ ▷ compute key via generalized delta vector
4: rename all α_i to β_{i+1} in k
5: $\mathcal{P}$ =CALCCHARPARTITION(k)
6: $\mathcal{P}'$ =BOUNDEDPOWER($\mathcal{P}$,n) ▷ permute all subsets of size $\leq n$ of $\mathcal{P}$
7: **for** $\mathcal{P}'_i$ **in** $\mathcal{P}'$ ▷ for each permutation **do**
8: $k' = k$
9: init empty map $rmap$ ▷ Map for storing rest fragments generated by substituting
10: **for** P_i **in** $\mathcal{P}'_i$ ▷ for each position set of $\mathcal{P}'_i$ **do**
11: $r = k|_{p_0}$, where $p_0 \in P_i$ ▷ backup an arbitrary rest
12: **for** p_j **in** P_i ▷ for each position in P_i **do**
13: **if** p_j exists in k' **then**
14: set $k'_{p_j} = \alpha_j$
15: **end if**
16: **if** all replacements on positions $p_j \in P_i$ were successfully executed **then**
17: $rmap(i) = r$
18: **end if**
19: **if** not a single β_l occurs in k' **then**
20: add k' to $keys$
21: $finalrests = \left\{ \begin{pmatrix} rmap(i_1)[\beta_{i_1} \backslash r_{i_1}], \\ \ldots, \\ rmap(i_m)[\beta_{i_m} \backslash r_{i_m}] \end{pmatrix} \right\}$ ▷ substitute rest fragments
22: $decompMap(k') \stackrel{\circ}{=} finalrests$
23: **end if**
24: **return** $keys$
25: **end function**

Example 15 (Example application of NORMFORM)
Let $L = \{f(a, g(b)), f(b, g(a)), f(a, g(c))\}$ be our (sub-)language from which we want to compute all keys in normal form, where $n = 2$.
We initialize keys *with an empty set in Line 2 and proceed by calling* GDV *(Line 3) for our language L.*
In further consequence we obtain a key

$$k = f(\alpha_0, g(\alpha_1))$$

and its corresponding rests

$$\overline{r} = \left\{ \begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} b \\ a \end{pmatrix}, \begin{pmatrix} a \\ c \end{pmatrix} \right\}$$

Since all $t \in L$ share a common leading function symbol on position ϵ, namely f, as well as on position 2, namely g, those function symbols remain in the generated key k.
Observe that

$$k \circ_{\alpha_0, \alpha_1} \overline{r} = L$$

In Line 4 we rename the non-terminals as described and obtain

$$k = f(\beta_1, g(\beta_2))$$

Afterwards we generate the characteristic partition of k in Line 5

$$\mathcal{P} = \{\{\epsilon\}, \{2.1\}, \{1\}, \{2\}\}$$

from which we generate all permuted subsets of size at most $n = 2$ denoted by sequences

$$\begin{aligned}\mathcal{P}' = \{ \quad & (\{\epsilon\}), (\{2.1\}), (\{1\}), (\{2\}), (\{1\}, \{2\}), \\ & (\{2\}, \{1\}), (\{2.1\}, \{1\}), (\{1\}, \{2.1\}), (\{2.1\}, \\ & \{2\}), (\{2\}, \{2.1\}), (\{\epsilon\}, \{2.1\}), (\{2.1\}, \{\epsilon\}), \\ & (\{\epsilon\}, \{1\}), (\{1\}, \{\epsilon\}), (\{\epsilon\}, \{2\}), (\{2\}, \{\epsilon\})\}\end{aligned}$$

In Line 7 we iterate over all previously generated permutations $\mathcal{P}'_i$ of $\mathcal{P}'$. We illustrate the following iterations on $\mathcal{P}'_2 = (\{1\})$ and $\mathcal{P}'_4 = (\{1\}, \{2\})$

For $\mathcal{P}'_2$ we first copy k and operate for this iteration merely on k'.
To keep track of our rest fragments we initialize $rmap$ as an empty map.
We proceed by iterating over all position sets within $\mathcal{P}'_2$, i.e. $\{1\}$.
The rest fragments of all positions in $\{1\}$ are denoted by $r = k'|_1 = \beta_1$.
Since the first position 1 exists in k' we replace the term at this position by α_1 as described in line 14.
As this is the only position in the set we continue by checking if all replacements on positions in $\mathcal{P}'_2$ were successfully executed, which in fact was the case. The intermediate computed key is denoted by $k' = f(\alpha_1, g(\beta_2))$ Therefore we store the calculated restfragment $r = \beta_1$ in $rmap$.
Next we have to evaluate if there occurs a β_l in k', which unfortunately is the case, as one can observe at $k|_{2.1} = \beta_2$. Therefore this iteration did not result in a key in normal form and we drop all previously computed restfragments consequently by reinitializing $rmap$ in the next iteration.

Consider following case of $\mathcal{P}'_4 = (\{1\}, \{2\})$ which eventually will result in a key in normal form. Again this iteration operates on a copy of k, namely k' and on a freshly initialized map $rmap$.
Iterating over all position sets of $\mathcal{P}'_4$, i.e. $\{1\}, \{2\}$ results in following computation steps.
The first iteration will happen analogously to previously described one, as our first position set equals $\{1\}$. Therefore we obtain $rmap(1) = r = k'|_1 = \beta_1$ and $k' = f(\alpha_1, g(\beta_2))$.
In the next iteration, applied on $\{2\}$, we generate the corresponding restfragment $r = g(\beta_2)$.
Since the position 2 exists in the intermediate key k', i.e. $k'|_2 = g(\beta_2)$ we proceed by replacing it with α_2 and consequently store the restfragment in $rmap$

$$rmap(2) = r = g(\beta_2)$$

Since all position sets were processed we proceed in Line 19 by checking if there still exists at least a β_l in $k' = f(\alpha_1, \alpha_2)$. Obviously this is not the case which implies that we found a new key in normal form w.r.t. L and $n = 2$.

Therefore we store k' in our list of keys in normal form $keys$ and calculate the $finalrests$ by means of our previously backed up restfragments in $rmap$, i.e.

$$finalrests = \left\{ \begin{pmatrix} \beta_1[\beta_1 \backslash a] \\ g(\beta_2)[\beta_2 \backslash b] \end{pmatrix}, \begin{pmatrix} \beta_1[\beta_1 \backslash b] \\ g(\beta_2)[\beta_2 \backslash a] \end{pmatrix}, \begin{pmatrix} \beta_1[\beta_1 \backslash a] \\ g(\beta_2)[\beta_2 \backslash c] \end{pmatrix} \right\}$$

which results in

$$finalrests = \left\{ \begin{pmatrix} a \\ g(b) \end{pmatrix}, \begin{pmatrix} b \\ g(a) \end{pmatrix}, \begin{pmatrix} a \\ g(c) \end{pmatrix} \right\}$$

Finally we store the finalrests associated with our key in normal form k' in decompMap *as shown in Line 22. In the end of the computation of* NORMFORM *we obtain the set of keys*

$$keys = \{\alpha_1, f(\alpha_1, \alpha_2), f(\alpha_2, \alpha_1), f(\alpha_2, g(\alpha_1)), f(\alpha_1, g(\alpha_2)), \alpha_2\}$$

and their corresponding finalrests stored in decompMap.

Recall Definition 50 of a rest fragment and an illustration of it in Example 14. We have to consider possible rest fragments in between α_i and β_l, while applying the mentioned replacements. After all β_l became conceivably successfully replaced by variables α_i, we perform substitutions for all β_l contained in the rest fragments by their individual rests (see Lines 19 to 22). The rests, denoted by their corresponding variables α_i, are added to a list associated to k' in $decompMap$. Note that the maximal amount of variables β_l is bounded by $m \leq n$, as previously mentioned, and thus the case of obtaining a new key k' by successfully replacing all β_l, leads to an introduction of at most n variables α_i. This fact allows us to restrict the size of a permutation of position sets to at most n, since for each permutation at most n variables α_i get introduced, which represents the previously mentioned crucial complexity bound. Finally the set of all keys in normal form is returned in Line 24.

Next we are going to generate the sufficient set of keys from a given language L. We start by generating all subterms of L and proceed by again calling $boundedPower$, but this time for the set of all subterms of L and a bound of $n+1$. Again this bound is crucial to avoid generating the whole power set of subterms of L, but only sets of size at most n. For each of those sets $normform$ will be called in Line 5, whose complexity is also bound as previously described by our omnipresent bound n. As we have successfully calculated the set of keys in normal form of L' and stored its real rests in $decompMap$, we now have to add the returned keys to our data structures, where all keys get associated with all terms $t \in L'$ and vice versa. This step is needed to store which key originated from which terms and which terms can be produced by which keys. Finally the sufficient set of keys can be found in $keyList$.

Algorithm 11 Compute a sufficient set of keys

```
1: function SUFFKEYS
2:     st = subterms(L)                          ▷ compute all subterms of a term set
3:     poweredSubsets =BOUNDEDPOWER(st,n + 1)    ▷ gen. subsets of st of size ≤ n + 1
4:     for L′ in poweredSubsets do
5:         keys =NORMFORM(L′)                    ▷ gen. keys for L′ in normal form
6:         add all keys to keyIndexMap and keyList
7:         let keyIndexes be the corresponding indexes of keys
8:         for t in L′ do
9:             add keyIndexes to keyMap(t)
10: end function
```

Generating a MinCostSAT formulation

As we already initialized all necessary data structures, i.e. our sufficient set of keys for a language L, we are able to obtain the corresponding MinCostSAT formulation as described in Section 3.4. The definition of the MinCostSAT formulation can be found in Paragraph 43 and is subdivided into analogous parts. Let us start with Algorithm 12 which generates for a subterm t of a term q and a non-terminal index l the formula $D_{L,S}(t, l, q)$. First we possibly add q to the $termMap$ via $addToTermMap$ and retrieve its index, before we proceed by declaring our resulting formula D. The formula itself represents a disjunction of formulas, thus we initialize said with $\bot$, as this has no effect whatsoever on the logical expression of our formulation. In Line 4 we iterate over all keys producing the particular subterm t of q. Note that we are going to encode every key, term, etc. by its index in their corresponding data structures. Therefore i_k represents the index of the key k in $keyList$. Next we initialize the propositional variable x_{l,i_k} representing the rule $\alpha_l \to k$ and assign it in the $propRules$ map to a tuple carrying all relevant indexes. This may seem redundant, because the indexes could also be retracted out of the propositional variable. Indeed the trade-off between explicitly storing additional tuples in the memory containing the indexes, and necessary object operations to retrieve said indexes from a propositional variable forced us to store the information separately. We collect all variables/non-terminals contained in k, after retrieving k from $keyList$ via its index i_k. Since we are at this point invariably interested in the particular indexes of the variables contained in k, we extract them in Line 9. As in the theoretical part already described we may proceed to handle the current key, if all variables contained in k have indexes i s.t. $i > l$. This condition ensures that our grammar is going to be acyclic and corresponds to Definition 30. If this rule is able to take place in our solution, regarding above condition, we check every possible rest (stored in $decompMap(k)$) if it is actually a valid rest of k w.r.t. t. Consider Algorithm 8 of $isRest$ deciding whether a particular key k, when performing according substitutions w.r.t. a rest vector $\overline{r}$ result in a particular term t. If so, we instantiate for each rest r_i a propositional variable $x_{i_{r_i},l,i_q}$ (where i_{r_i}, i_q are the indexes of r_i and q respectively) and conjugate them. This forms up a consequence, which claims that if the rule $\alpha_l \to k$ is chosen to be in the solution, then all of its rests, if there are any, have to be further decomposed by rules $\alpha_j \to k'$, where $j > l$. The disjunction of all of those conjunctions is returned as formula $D(t, l, q)$, which represents a list

of individual dependencies for all rules.

Algorithm 12 Compute $D(t, l, q)$ formula

function D(t: subterm, l: non-terminal index, q: term)
  $i_q = addToTermMap(q)$ ▷ get index of q
  $D = \bot$ ▷ init formula D
  for i_k **in** $keyMap(t)$ **do**
    init atom x_{l,i_k}
    $propRules(x_{l,i_k}) = (l, i_k)$ ▷ backup tuple
    $k = keyList(i_k)$ ▷ get key with index i_k
    let $A = \{\alpha_{i_1}, \ldots, \alpha_{i_m}\}$ be all non-terminals of k sorted by index
    let $I = \{i_1, \ldots, i_m\}$ be the indexes of $\alpha_i \in A$
    if for all $i \in I$ holds that $i > l$ **then**
      $D_k = x_{l,i_k}$ ▷ init subformula D_k
      for $\overline{r}$ **in** $decompMap(k)$ **do**
        if ISREST($t, k, \overline{r}$) **then**
          for r_i **in** $\overline{r}$ **do**
            $i_{r_i} = addToTermMap(r_i)$ ▷ get index of r_i
            init atom $x_{i_{r_i},l,i_q}$ ▷ let l be the index of a non-terminal replacing r_i
            $propRests(x_{i_{r_i},l,i_q}) = (i_{r_i}, l, i_q)$ ▷ backup triple
            $D_k = D_k \wedge x_{i_{r_i},l,i_q}$
        end if
      $D = D \vee D_k$
    end if
  return D
end function

We already know that, we have to make sure that our resulting grammar will be totally rigid. Consider Algorithm 13 which generates for a particular term q the formula $R(q)$ ensuring this condition. Recall Algorithm 7 which, for a given set S, provides a set of pairwise different elements $x, y \in S$, without an ordering. Through this function we generate from the set $subtermindexes$, a set of all pairs of indexes of subterms of q. For each of those pairs we will initialize propositional Variables x_{i_1,i,i_q} and x_{i_2,i,i_q} respectively, add them to our $propRests$ map and conjugate the formula $(\neg x_{i_1,i,i_q} \vee \neg x_{i_2,i,i_q})$ with already computed R, where initially $R = \top$. This formula allows us to represent the fact that all rests of term q have to be mutually exclusive, i.e. there can always exist at most one rest for a non-terminal within a derivation at a time. Finally the formula $R(q)$ is returned.

Algorithm 13 Compute $R(q)$ formula

1: **function** R(i_q: term index of q, $subtermIndexes$: set of subterm indexes of q)
2: $pairs =$DIAGCROSS($subtermIndexes$) ▷ pairs $i_1 \neq i_2 \in subtermIndexes$
3: $R = \top$ ▷ init formula R
4: **for** (i_1, i_2) **in** $pairs$ **do**
5: **for** i **in** $1, \ldots, n$ **do**
6: init atom x_{i_1,i,i_q}
7: init atom x_{i_2,i,i_q}
8: $propRests(x_{i_1,i,i_q}) = (i_1, i, i_q)$ ▷ backup triple
9: $propRests(x_{i_2,i,i_q}) = (i_2, i, i_q)$ ▷ backup triple
10: $R = R \wedge (\neg x_{i_1,i,i_q} \vee \neg x_{i_2,i,i_q})$
11: **return** R
12: **end function**

As we have already implemented a way to generate $D(t, l, q)$ and $R(q)$, provided corresponding parameters are provided, we continue by defining a way to compute $C(q)$ in Algorithm 14 as described in Definition 47. The function, given a term q, will first compute all subterms of q and add it to the $termMap$ to retrieve its corresponding index. As we will need all indexes of subterms of q for the corresponding call of $R(q)$, we initialize $subtermIndexes$ as an empty set for collecting them. Since $C(q)$ represents a conjunction, where its individual components are generated w.r.t. a subterm t and an integer i, it gets initialized with $\top$. We proceed by iterating over all subterms t of q, while again adding them to the $termMap$, and retrieve their indexes, which are added to our set of $subtermIndexes$. For $i \in \{1, \ldots, n\}$ we generate propositional variables x_{i_t,i,i_q}, where again i_t and i_q are t's and q's indexes respectively, and add them to our map $propRests$. In a moment we will generate the formula $D(t, i, q)$, but this does not cover the case of a *trivial rule*, i.e. a rule $\alpha_i \to k$ where k contains no α_j whatsoever. Those rules are necessary for our grammar to terminate, s.t. a point is reached were we cannot apply another rule, i.e. after a derivation $\delta = t_1 \to \cdots \to t_m$ where t_m contains no further α_j. To accomplish this we have to add another possible transition from a non-terminal α_i to the whole rest, which is currently processed. Consider Lines 12 to 14 where a rule with a trivial key t gets instantiated and stored in the corresponding map. After calling $D(t, i, q)$ we build a disjunction of said with the trivial rule. Finally C is extended iteratively. After partly formulating C by generating according subformulas for every $t \in st(\{q\})$ and $1 \leq i \leq n$, we generate $R(q)$ via Algorithm 13 and $D(q, 0, q)$ via Algorithm 12 respectively. Again we have to consider a special case of a trivial production rule, namely $\alpha_0 \to q$, which is not covered by above introduced trivial rules. Therefore we need a propositional variable $x_{0,i_{triv}}$, where i_{triv} is the key index of q. Note that in this case q is interpreted as a key, not a term, since we can only provide rules $\alpha_i \to k$, where k is a key. Thus we have to take its index as a key for the construction of a propositional variable. Remark: $D(t, i, q)$ or $D(q, 0, q)$ may be empty, i.e. there could be no key with according rests to further decompose t or q respectively, without violating one of previously defined conditions for a $trat$-n grammar. This would lead to D consisting only of a single atom, namely the propositional variable representing the trivial rule. The formula C is extended accordingly by above mentioned subformulas in Line 21 and returned afterwards.

Algorithm 14 Compute $C(q)$ for a term $q \in L$

function C(q: term)
  $st = subterms(q)$ ▷ compute subterms of q
  $i_q = addToTermMap(q)$ ▷ store term in $termMap$
  init set $subtermIndexes = \{\}$
  init formula $C = \top$ ▷ formula which is going to be returned
  for t **in** st **do**
    $i_t = addToTermMap(t)$ ▷ get index of subterm
    $subtermIndexes \stackrel{\circ}{=} i_t$ ▷ backup index of subterm
    for i **in** $1, \ldots, n$ **do**
      init atom x_{i_t,i,i_q} ▷ init atom $x_{t,\alpha_i,q}$
      $propRests(x_{i_t,i,i_q}) = (i_t, i, i_q)$ ▷ backup triple
      $i_{triv} = addKey(t)$ ▷ store t as trivial key
      init atom $x_{i,i_{triv}}$ ▷ init atom $x_{\alpha_i,t}$
      $propRules(x_{i,i_{triv}}) = (i, i_{triv})$ ▷ backup tuple
      $D_{t,i,q} =$D(t, i, q) ▷ generate $D_{L,S}(t, i, q)$
      $C = C \wedge (x_{i_t,i,i_q} \rightarrow (D_{t,i,q} \vee x_{i,i_{triv}}))$
  $R_{i_q,st} =$R$(i_q, subtermIndexes)$ ▷ generate $R_{L,S}(q)$
  $D_{q,0,q} =$D$(q, 0, q)$ ▷ generate $D_{L,S}(q, 0, q)$
  $i_{triv} = addKey(q)$ ▷ store q as trivial key
  init atom $x_{0,i_{triv}}$ ▷ init atom $x_{\alpha_0,q}$
  $C = C \wedge (D_{q,0,q} \vee x_{i,i_{triv}}) \wedge R_{i_q,st}$
  return C
end function

Finally we have to call the method described above for generating $C(q)$ for all $q \in L$. The final MinCostSAT formulation is then the conjunct of all $C(q)$ for $q \in L$.

Algorithm 15 Compute MinCostSAT formula

function MCS(L: term set/language)
  $F = \top$ ▷ init F
  for q **in** L **do**
    $F = F \wedge$C(q) ▷ iteratively build MinCostSAT formula F
  return F
end function

Transformation to partial weighted MaxSAT

Next we have to transform the MinCostSAT formulation in 4.2 into an instance of the partial weighted MaxSAT problem described in Definition 26 to a reasonable MaxSAT solver. Such a solver requires the instance to be in a certain format, i.e. a formula F which has to hold in every interpretation and a set of formulas G which can be violated, but are penalized with an individual

value for every one of them violated in a particular interpretation. Since we want to achieve our grammar to have a minimal number of rules, we generate G as described in Algorithm 16. We initialize G as an empty set of tuples, where each will contain a formula and its weight (or penalty). Iteratively for all rules $\alpha_i \rightarrow k$ the simple formula $\neg x_{i,i_k}$ is created by means of its corresponding propositional variable and gets assigned weight 1. This leads to the fact that the more variables x_{i,i_k} are true in an interpretation, the higher the costs of the interpretation will be, forcing the partial weighted MaxSAT Solver to minimize the number of rules needed while not violating any constraints in F.

Algorithm 16 generate soft constraints G for the partial weighted MaxSAT instance

```
1: function SOFTCONSTRAINTS
2:     G = {}                          ▷ init G as an empty set of tuples of formulas and weights
3:     for (x_{i,i_k}, (i, i_k)) in propRules do
4:         G ≙ {(¬x_{i,i_k}, 1)}
5:     return G
6: end function
```

Obtaining the grammar

As a partial weighted MaxSAT solver will return merely an interpretation fulfilling F with a minimal violation of G, it remains to transform the resulting interpretation to a set of rules to retrieve the grammar. Algorithm 17 returns, given an interpretation I, a set of rules $\alpha_i \rightarrow k$, where $I(x_{i,i_k}) = true$.

Algorithm 17 retrieve a set of rules from an interpretation

```
1: function GETRULES(I: interpretation)
2:     R = {}                                   ▷ init the set of rules R
3:     for (x_{i,i_k}, (i, i_k)) in propRules do
4:         if I(x_{i,i_k}) then
5:             k = keyList(i_k)
6:             R ≙ {α_i → k}
7:         end if
8:     return R
9: end function
```

By means of above generated set of rules, we can build a grammar as described in Algorithm 18.

The Algorithm

We already calculated all our needed keys and successfully transformed them, by means of our data structures, into a MinCostSAT formulation and by adding additional soft constraints to an instance of the partial weighted MaxSAT problem. In Algorithm 19, which comprises the calls

Algorithm 18 build a grammar from a set of rules

```
1: function GETGRAMMAR(R: set of rules)
2:     let smap be an empty map from integer to term set
3:     for α_i → k in R do
4:         if smap(i) exists then
5:             smap(i) ≐ {k}
6:         else
7:             smap(i) = {k}
8:         end if
9:     let U = smap(0)
10:    for all i let S_i = smap(i)
11:    return U ∘_{α_1} S_1 ∘_{α_2} ··· ∘_{α_n} S_n
12: end function
```

of recently described computations, we can now see the big picture of our implementation. In Line 2 we generate our sufficient set of keys and proceed in Line 3 by formulating our MinCost-SAT instance and compute in Line 4 the additional clauses necessary for our partial weighted MaxSAT instance. This instance will be given to a desired complete MaxSAT Solver in Line 6, which eventually returns an interpretation. If there is an interpretation, it is processed as described in Section 43 and Algorithms 17 18 in Lines 8 9 and finally the expected minimal grammar will be returned. Otherwise an empty one will be returned, which means that there is no grammar abbreviating L, i.e. F is *unsatisfiable*.

Algorithm 19 Compute a minimal grammar for a proof

```
1: function TREEGRAMMARDECOMPOSITION(L: term set, n: integer)
2:     SUFFKEYS                                  ▷ calculate sufficient set of keys
3:     F =MCS(L)                                 ▷ generate hard constraints
4:     G =SOFTCONSTRAINTS                        ▷ generate soft constraints
5:     solver = MaxSATSolver()                   ▷ instantiate desired MaxSAT solver
6:     I = solver.SOLVE(F,G)                     ▷ solve instance
7:     if ∃x_{i,i_k} I(x_{i,i_k}) then
8:         rules =GETRULES(interpretation)       ▷ get grammar rules
9:         grammar =GETGRAMMAR(rules)            ▷ get grammar
10:        return grammar
11:    else
12:        return empty grammar                  ▷ F is UNSAT
13:    end if
14: end function
```

Additional Interfaces and Features

Since there are many partial weighted MaxSAT solvers available and nearly all of them support the standardized *DIMACS* format *wcnf* [5], which is an abbreviation for weighted conjunctive normal form, a transformation method was developed to output our instance in this format. Since the *GAPT* system had neither been capable of handling wcnf instances nor supported a MaxSAT solver, an interface was implemented to provide those functionalities [16]. This was achieved by providing a transformation method for an instance to a file in wcnf format, as well as a possibility to call a MaxSAT solver within *GAPT* on a shell and obtain either unsatisfiability or an interpretation from it. As not all MaxSAT solvers perform equally well on every problem, the support for following (Figure 4.2) was implemented in *GAPT*.

- QMaxSAT [12] [13]
- ToySAT, ToySolver [15]
- MiniMaxSAT [11] [6]

Figure 4.2: Implemented support for MaxSAT solvers

During the implementation of all described algorithms, as well as the interfaces for supported MaxSAT solvers the focus was set on keeping the code flexible. The interface for the solvers mentioned above was developed in a modular way, s.t. it is possible to extend those easily. Furthermore the transformation of a sufficient set of keys to a MinCostSAT formula and thus to a partial weighted MaxSAT instance was as well implemented s.t. another transformation, e.g. to an *ILP* (*integer linear programming*) instance, could be integrated with a minimum amount of effort.

CHAPTER 5

Critical reflection

5.1 Comparison with related work

In the last chapters we covered the theoretical background as well as the practical implementation of an algorithm capable of reducing the problem of finding a minimal $trat$-grammar for a language to the MinCostSAT problem. Although the theoretical part focused on a proof's end-sequent of the form

$$\forall x F \rightarrow$$

the method described in [9] is capable of processing arbitrary Σ_1-sequents, which are of the following form

$$\forall \overline{x}_1 F_1 \ldots \forall \overline{x}_k F_k \rightarrow \exists \overline{y}_1 G_1 \ldots \exists \overline{y}_l G_l \tag{5.1}$$

as seen in Section 3.1 Definition 28.

Unfortunately the TreeGrammarDecomposition algorithm is only capable of producing particular grammars, namely those which translate to cuts having single quantified cut-formulas. To be precise it does not support at the moment cut-formulas containing blocks of quantifiers, e.g.

$$\forall x_1, \forall x_2, \ldots, \forall x_n F \tag{5.2}$$

Results

We compared the algorithm in [8] of computing a minimal grammar in Section 3.4 with the recently developed method in [3] described in Section 3.4 and the first one developed for this purpose, which was merely capable of introducing one cut via a single quantified formula. The former is capable of producing grammars, which support cut-formulas with blocks of quantifiers. It is even possible to handle underlying proofs in Equational Logic. In contrast the second one is capable of producing grammars translating into several cuts, but merely with single-quantified cut-formulas. The opportunity to introduce several quantified cuts into a sequent calculus proof

hopefully leads to new insights. Although the results of this thesis are very expressive, further experiments will be performed to provide more information about this topic.

The following figures will provide a comparison of the methods of cut-introduction confronted with different types of proofs (VeriT, TPTP, proof sequences), which they were executed on. The rows denote the different methods starting with the old method for introducing a single quantified cut, proceeding by the evolved method for introducing a multiple quantified cut and finally providing information for the method described in this thesis for introducing various single-quantified cuts. The last two rows are devoted to the mentioned method were in the first one we tried to introduce one single-quantified cut, whereas in the last one up to two single-quantified cuts were tried to be introduced, which was not even possible with the other methods described. The columns represent the different proofs which were the foundation for the experiments, starting with proof sequences which in fact are simply structured proofs like the one shown in 3.2, which were generated in various sizes. Those sequences were generated in incrementing sizes and tested with the various methods until a predefined timeout was reached. All sequences of larger sizes were then considered to be not compressible within this timeout and thus skipped. The second column represents a sample of various proofs found in the TPTP library, where their compression is considered to have a higher impact than the compression of the former ones. The third and last column denote the results gathered from a sample of VeriT proofs which are in contrast to the last two samples very large.

Figure 5.1 provides an overview of the executed experiments regarding their return status. Please consider the provided key for the possible range of statuses returned by the different methods. The most essential statuses necessary for describing the effect of the method of this thesis are the following.

- *ok* states the percentage of proofs, which were successfully compressed by the method
- *parsing_timeout* was returned if the system was not even possible to parse the provided proof within the timeout
- *delta_table_computation_timeout* is a status return when the Δ-table method was not capable of providing the Δ-table within the timeout
- *grammar_finding_timeout* was returned if the timeout was reached while calculating the minimal grammar of an already extracted language
- *cut_intro_uncompressible* were proofs which were not compressible by introducing the given number of cuts
- *prcons_timeout* states that the construction of the already compressed proof was not finished due to a timeout
- *sol_timeout* denotes the status returned when a timeout happened while calculating the canonical solution of the proof

As we can see in Figure 5.1 the newly introduced method is not as fast as the other two methods considering proof sequences. In fact the amount of timeouts in proof construction,

which for all methods is essentially the same, as well as the amount of successfully finished proofs slightly differs between the *single-cut* and the *many-cuts* method. This differences persist and even increase considering the Prover9 (TPTP) proofs, which are substantially harder to compress than the provided proof sequences. For VeriT instances we cannot even compute a minimal grammar in about 95% of the cases, which is not surprising when taking the size of those proofs into account. The general timeout for the latter proofs was even significantly raised to adapt to the mentioned circumstance.

In Figure 5.2 we compare the runtime for the different phases of the methods for all provided proofs. The following measured times are the most expressive ones for comparing the methods

- *time_grammar* denotes the runtime necessary for calculating a minimal grammar of an already extracted language/termset

- *time_minsol* represents the runtime of minimizing the found canonical solution of the proof

- *time_proof* is the relative runtime for constructing the proof via an extended Herbrand sequent

As we can see in the first column, the minimal grammars of the proof sequences were significantly faster calculated by the *many-cuts* method than by the other ones. Since the proof construction time is essentially the same for all methods, we can assume that for proof sequences the new method performs better, although those instances are not of practical significance. The differences in runtime go into reverse when considering more practical instances as TPTP and VeriT proofs. In contrast to the *one-cut* methods, where the runtime for computing a minimal grammar is relatively small, the *many-cuts* method takes significantly longer to return a grammar when compared to constructing the proof. Since the latter method is capable of introducing more than one cut at once, it is probably less efficient for introducing exactly one cut than the *one-cut* method, which is especially designed for this case. Be aware of the fact that the intermediary canonical solution is not improved by the *many-cuts* method yet. Therefore the results for time consumption should not be considered to be conclusive.

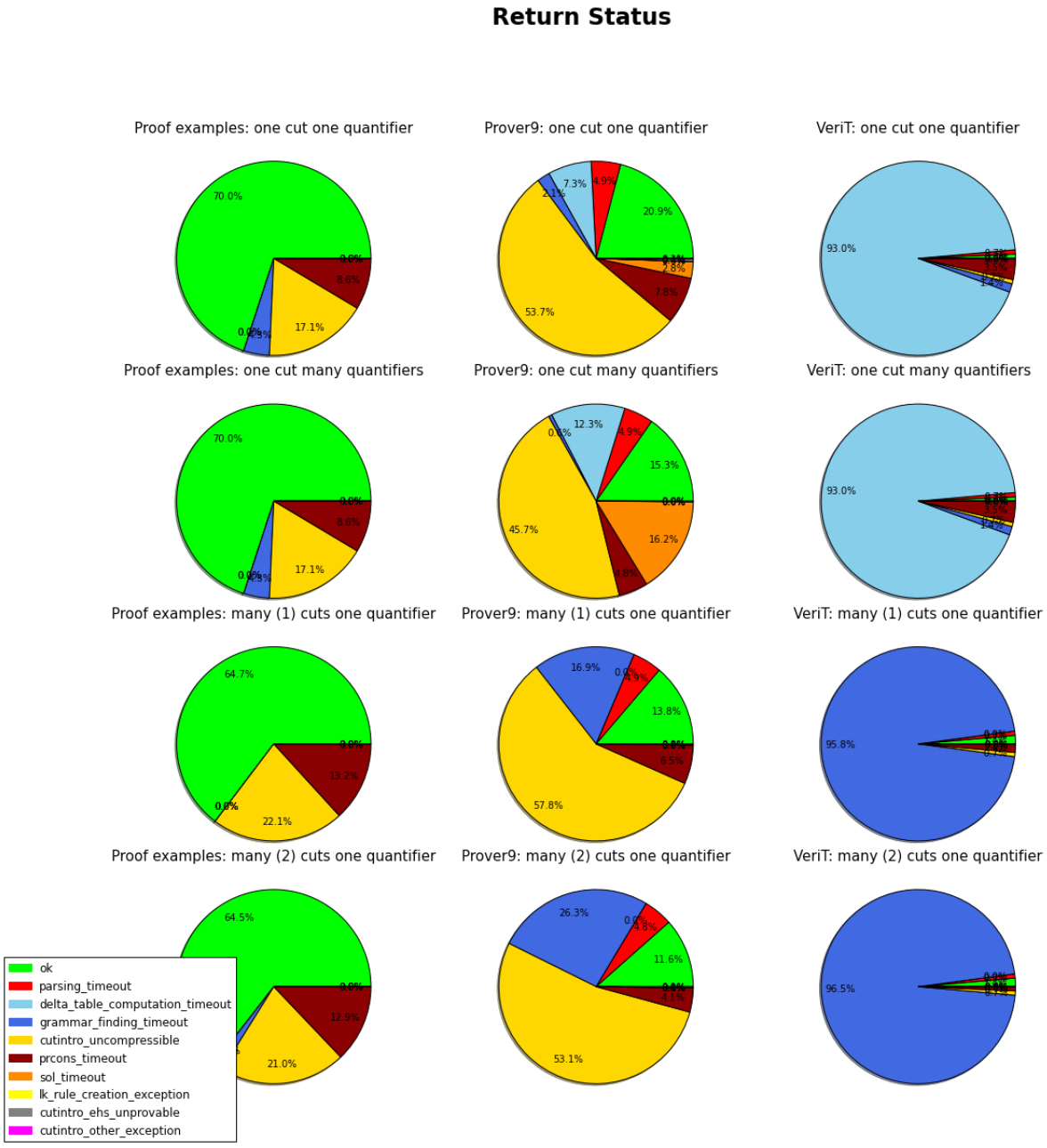

Figure 5.1: Return statuses of the experiments

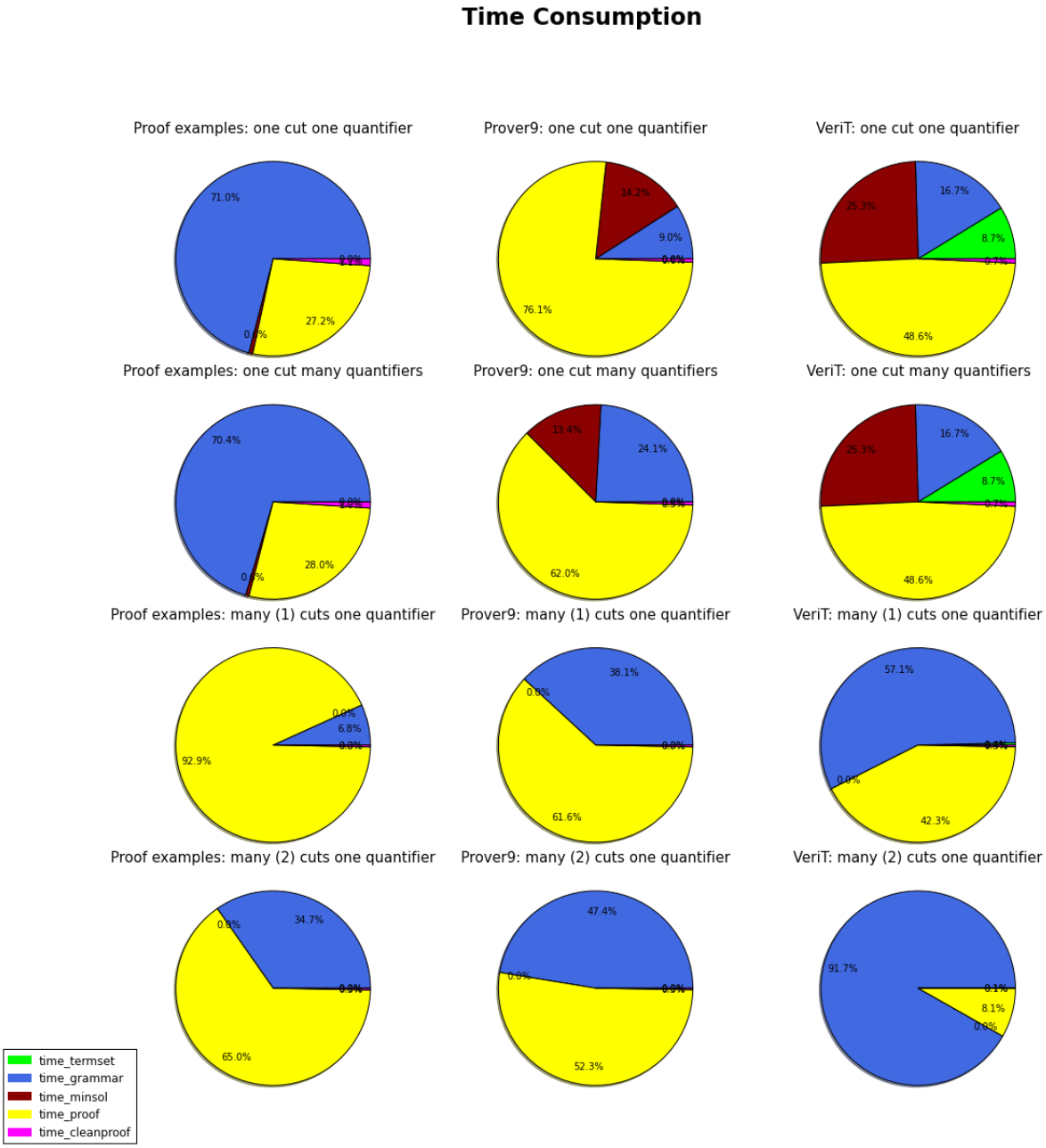

Figure 5.2: Time consumption for finished instances

In conclusion, although the *many-cuts* method does not significantly improve the efficiency of the previous methods, the possibilities of proof compression were extended by the possibility of introducing more than one single-quantified cut. This may lead to new insights into lemma construction in general, as well as for already known proofs, where new lemmas could

be discovered.

5.2 Complications and open issues

Performance issues

As expected the work on this thesis was not completely frictionless. During the first tests it turned out that the algorithm was performing worse than the last w.r.t. its runtime. After a slight profiling the reason for this lack of efficiency was found somewhere in between the solving of the partial weighted MaxSAT instance. In order to call a desired MaxSAT solver, it is necessary to transform the formulas F, G as described in Section 43 to the DIMACS *wcnf* format. This format represents an encoding of propositional formulas F, G in *conjunctive normal form*, i.e. a formula as shown in Definition 51.

Definition 51 (Conjunctive Normal Form)
Let C_i be formulas in propositional logic only containing disjunctions of literals, then

$$\bigwedge_{i=1}^{n} C_i$$

is a formula in conjunctive normal form. Note: C_i is called a clause of the CNF.

Since our MinCostSAT formulation is not given in CNF, we had to transform it to this normal form. In *GAPT* this was accomplished by a naive transformation, namely by transforming all logical connectives by the means of the *Double Negative Law*, *DeMorgan's Law*, the *Distributive Law* and the *replacement of the material implication*.

Those equivalences suffice to transform an arbitrary formula in propositional logic into conjunctive normal form. Unfortunately this is not the most efficient way to achieve a CNF, since the formula will grow exponentially, which should be of no relevant consequences for small formulas, but appears to be a problem for our MinCostSAT formulation. We solved this problem by implementing the transformation according to *Grigori Tseitin* [21], which is of linear complexity.

Ambiguous grammars

Another issue arose while testing the algorithm, when dealing with ambiguous grammars. Consider the proof π^* in Figure 3.2, which proves the end-sequent

$$P(0), \forall x P(x) \supset P(s(x)) \to P(s^8(0))$$

The term set in Example 16 of the Herbrand sequent of this proof can be compressed with more than one grammar, whereas all of them have the same number of production rules.

Example 16 (Term set of $H(\pi^*)$)

$$L = \{0, s(0), \ldots, s^7(0)\}$$

Consider Examples 17 and 18 of grammar decompositions of L and note that they differ in the way the eigenvariables, i.e. the non-terminals, are introduced, although they have an equal number of rules.

Example 17 (Grammar decomposition for term set of $H(\pi^*)$)

$$\{\alpha_1, s^3(\alpha_1), s^5(\alpha_1)\} \circ_{\alpha_1} \{(0), (s(0)), (s^2(0))\}$$

Example 18 (Another grammar decomposition for term set of $H(\pi^*)$)

$$\{\alpha_1, s^4(\alpha_1)\} \circ_{\alpha_1} \{(0), (s(0)), (s^2(0)), (s^3(0))\}$$

The grammar decomposition in Example 17 induces an extended Herbrand sequent, which can be transformed into a proof with one cut smaller than π^*. Whereas the decomposition in Example 18 can be transformed analogously and forms up a larger proof, with more applied rules. This issue is already known and was evaded in the first method by computing all minimal grammars and generating for each of them a proof, where merely this with the least number of rules was selected. Since the known MaxSAT solvers are not capable of finding all minimal grammars for a particular term set, this issue remains unsolved for the TreeGrammarDecomposition algorithm. Although the issue could be resolved in a naive way like follows. We can iteratively call a desired MaxSAT solver with the MinCostSAT formula, in combination with the introduced soft constraints while we forbid all previously discovered interpretations to be found. This will be done until a decomposition is found, which has a higher number of rules, than each of the previously found grammars. We then proceed as mentioned above by building all proofs according to the extended Herbrand sequents of the individual computed grammars and take the one with the minimal number of applied rules. For now the support for various MaxSAT solvers is assumed to fit the needs, as each of them possibly returns other interpretations in such cases. Although it is not guaranteed that we can possibly generate all different minimal grammars with the different solvers, the range of them allows us to potentially compute more than one.

CHAPTER 6

Summary and future work

We covered in this thesis the introduction of universal cuts in **LK** proofs, by giving an overview of the topic in Chapter 1 and providing necessary preliminary definitions in Chapter 2. In Chapter 3 we first described how extended Herbrand sequents can be obtained from proofs with cuts.

Subsequently the concept of a schematic extended Herbrand sequent was introduced, which seperates the needed instantiations of a proof from the number and the form of involved cuts. The necessary instantiations were then interpreted as a grammar decomposition, which produces a particular term set representing exactly those instantiations needed for the Herbrand-sequent of the cut-free proof.

Inverting this method of proof compression induced the problem of finding a minimal grammar decomposition for a language, which can be solved in two different ways. The first method, which was already implemented in *GAPT*, accomplished that by generating all minimal grammars and taking the one which represents a proof with a minimal number of rules [8]. The latter described in [3] achieved the same goal by taking advantage of prior knowledge of SAT solving. It reduces the problem of finding a minimal grammar for a language to the solution of a MinCostSAT formula, which is possible by introducing a specific bound on the number of used non-terminals.

In Chapter 4 the implementation of the second method in the *GAPT* system was described. Particular problems, a comparison with similar approaches as well as complications were covered in Chapter 5, where we faced a problem with ambiguous grammars of same size. The described method for avoiding this issue could be implemented in the future by possibly encoding it into the MinCostSAT formulation. Also the support for cut-formulas of higher complexity offers an incentive to dedicate the attention to the topic and remains a matter of future research.

Bibliography

[1] Matthias Baaz and Alexander Leitsch. On skolemization and proof complexity. *Fundamenta Informaticae*, 20(4):353–379, 1994.

[2] Matthias Baaz and Alexander Leitsch. *Methods of Cut-elimination*, volume 34. Springer, 2011.

[3] Sebastian Eberhard and Stefan Hetzl. Algorithmic compression of finite tree languages by rigid acyclic grammars. 2014.

[4] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(1):176–210, 1934.

[5] Federico Heras, Javier Larrosa, Simon De Givry, and Thomas Schiex. 2006 and 2007 max-sat evaluations: Contributed instances. *JSAT*, 4(2-4):239–250, 2008.

[6] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: A new weighted max-sat solver. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 41–55. Springer, 2007.

[7] Jacques Herbrand. Recherches sur la théorie de la démonstration. 1930.

[8] Stefan Hetzl, Alexander Leitsch, Giselle Reis, Janos Tapolczai, and Daniel Weller. Introducing quantified cuts in logic with equality. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 240–254. Springer International Publishing, 2014.

[9] Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. Algorithmic introduction of quantified cuts. *Theoretical Computer Science*, 549, 2014.

[10] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. Herbrand sequent extraction. In *Intelligent Computer Mathematics*, pages 462–477. Springer, 2008.

[11] Daniel Izquierdo. Github of minimaxsat. `https://github.com/izquierdo/tesis_postgrado/tree/master/src/MiniMaxSat`, 2010. [Online; accessed 06-November-2014].

[12] Miyuki Koshimura. Website of qmaxsat. `https://sites.google.com/site/qmaxsat/`, 2014. [Online; accessed 06-November-2014].

[13] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.

[14] Xiao Yu Li. Optimization algorithms for the minimum-cost satisability problem. 2004.

[15] Masahiro Sakai. Github of toysat and toysolver. `https://github.com/msakai/toysolver`, 2014. [Online; accessed 06-November-2014].

[16] Christoph Spoerk. Maxsat.scala. `https://github.com/gapt/gapt/blob/master/src/main/scala/at/logic/provers/maxsat/MaxSAT.scala`, 2014. [Online; accessed 22-March-2015].

[17] Christoph Spoerk. Treegrammardecomposition.scala. `https://github.com/gapt/gapt/blob/master/src/main/scala/at/logic/algorithms/cutIntroduction/TreeGrammarDecomposition.scala`, 2014. [Online; accessed 22-March-2015].

[18] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[19] Theory and Logic Group. Ceres - a cut-elimination system. `http://www.logic.at/ceres/`, 2004. [Online; accessed 22-March-2014].

[20] Theory and Logic Group. Gapt - generic architecture of proofs. `https://github.com/gapt/gapt`, 2009. [Online; accessed 22-March-2015].

[21] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[22] Bruno Woltzenlogel Paleo. Atomic cut introduction by resolution: Proof structuring and compression. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 463–480. Springer Berlin Heidelberg, 2010.