FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automation of cut-elimination in proof schemata

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Tsvetan Chavdarov Dunchev
Matrikelnummer 0827680

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.phil. Alexander Leitsch

Diese Dissertation haben begutachtet:

(Univ.Prof. Dr.phil. Alexander Leitsch) (Dr. Nicolas Peltier)

Wien, 15.11.2012

(Tsvetan Chavdarov Dunchev)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automation of cut-elimination in proof schemata

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Tsvetan Chavdarov Dunchev
Registration Number 0827680

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr.phil. Alexander Leitsch

The dissertation has been reviewed by:

(Univ.Prof. Dr.phil. Alexander Leitsch) (Dr. Nicolas Peltier)

Wien, 15.11.2012

(Tsvetan Chavdarov Dunchev)

Erklärung zur Verfassung der Arbeit

Tsvetan Dunchev
Favoritenstrasse 9-11, 1040 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)

Contents

Kurzfassung

Die Schnittelimination von Gentzen [Gen35] ist eine der bekanntesten Methoden zur Beweistransformation, die eine grundlagende Rolle in der automatischen Analyse mathematischer Beweise spielt. In einem Beweis haben die Schnittformeln die Bedeutung der Lemmata. Ein Beweis ohne die Schnittformeln entspricht der Elimination der Lemmata. Man sagt, dass ein Beweis ohne Lemmata eigentlich ein analytischer Beweis ist, wenn alle Formeln des Beweises inm Endsequent vorkommen. Eine Verbesserung der Methode von Gentzen ist die Methode *CERES* (Schnittelimination durch Resolution) [BL00]. Die ist eine Methode für Schnittelimination in der Logik erster Stufe, die die Teile des Beweises mit Schnittregeln analysiert. Diese Information ist besonderes wertvoll für die Konstruktion einer Menge von Klauseln, die immer unerfüllbar ist. Deren Widerlegung unterliegt theoretischen und praktischen Einschränkungen. Man kann zum Beispeil *CERES* nicht anwenden, wenn der Beweis eine Induktionsregel beinhaltet. Wenn das Beweis zu gross wird, dann finden die besten Theorembeweisern (derzeit Prover9 und Vampire) kein Ergebnis mehr. Eine Lösung des ersten Problems ist durch die Erweiterung von *CERES* auf $CERES_s$ gegeben. Die Methode $CERES_s$ ist eine Methode für Schnittelimination auf den Schemata erster Stufe. Schemata dienen dazu, unendliche Folgen von Beweisen darzustellen; diese Schemata bezeichnen wir als Beweisschemata. Die Methode $CERES_s$ ist nur teilweise automatisierbar, weil die Resolution einer Menge von schematischen Klauseln erster Stufe nicht entscheidbar ist.

In dieser Dissertation beschreiben wir die Methode $CERES_s$ und präsentieren die Algorithmen und Werkzeuge des **GAPT**-Systems [LWWP$^+$] für automatische Analyse schematischer Beweise erster Stufe.

Abstract

Cut-elimination introduced by Gentzen [Gen35] is the most prominent form of proof transformation in logic and plays a key role in automating the analysis of mathematical proofs. The removal of cuts corresponds to the elimination of intermediate statements (lemmas) used in proofs. The result is a proof which is analytic in the sense that all statements in the proof are subformulas of the result. An improvement of the Gentzen's method is the method *CERES* (Cut Elimination by Resolution) [BL00]. It is a method of cut-elimination in first-order logic which analyzes the parts of the proof used in cut inferences. This information is used to construct a set of clauses which is always unsatisfiable. The refutation of this clause-set serves as a skeleton of a proof with at most atomic cuts. The method *CERES* has some limitations from a theoretical and practical point of view. For example, *CERES* is not applicable in the presence of induction. Furthermore, for large proofs the clause-sets cannot be refuted because automated theorem provers (such as Prover9) fail. A solution for the first problem is obtained by extending *CERES* to $CERES_s$ - a method for cut-elimination in first-order schema. The concept of schemata allows us to represent an infinite sequence of proofs as a singe object called proof schemata. Nevertheless, the whole process is not fully automatable, because the problem of finding a resolution refutation for the set of first-order claus schemata is not semi-decidable.

In this thesis we describe the method $CERES_s$ and present algorithms and tools which are used by the **GAPT** framework [LWWP$^+$] for (semi)automated analysis of first-order proofs and proof schemata.

Acknowledgements

I would like to express my deep gratitude to Prof. Alexander Leitsch for giving me the opportunity to join one of his interesting and challenging international research projects. During the four years as his PhD and Master student I learned a lot from him both as a scientist and a person. I am very thankful to the rest of the members of Alex's project: Daniel Weller, Mikheil Rukhaia and David Cerna as well as to our partners from the University of Grenoble: Nicolas Peltier (coordinator and my second adviser), Vincent Aravantinos (who joined our team for one year), Abdelkader Kersani and Thierry Boy de la Tour. The cooperation and the regular joint project meetings in Vienna and Grenoble played a significant role for finding solutions to crucial problems. I would like also to say many thanks to Tomer Libal who gave me much software-engineering advise and contributed a lot to the implementation of the **GAPT** framework. Many thanks also to Martin Riener and Stoiko Ivanov who revealed to me many of the secrets of *GNU/Linux*. I also would like to thank the other current and former PhD students of Alex: Martin Riener, Mikheil Rukhaia, David Cerna, Giselle Machado N. Reis, Bruno Woltzenlogel Paleo, Daniel Weller and Stefan Hetzl. With each of them I had many valuable discussions related with my work. Many thanks to David Cerna. Being that his native language is english, he was able to edit many parts of my thesis. Last but not least, I would like to thank my colleagues and friends from the *Theory and Logic Group* of *TU Wien*: Paolo Baldi (who taught me italian and showed me south Italy), Lara Spendier, Christoph Roschger, Eugen Jiresch and Florian Schweikert. From each of them I learned something and had many nice moments during my stay in Vienna.

Chapter 1

Introduction

An important method in the development of mathematics is the analysis of proofs. Indeed many mathematical concepts such as the notion of group or the notion of probability were introduced by analyzing existing arguments. In some sense the analysis and synthesis of proofs form the very core of mathematical progress. Cut-elimination introduced by Gentzen [Gen35] is the most prominent form of proof transformation in logic and plays a key role in automating the analysis of mathematical proofs. The removal of cuts corresponds to the elimination of intermediate statements (lemmas) from proofs resulting in a proof which is analytic in the sense that all statements in the proof are subformulas of the result. Therefore, the proof of a combinatorial statement is converted into a purely combinatorial proof. The development of the method *CERES* (cut-elimination by resolution) [BL00] was inspired by the idea to fully automate cut-elimination on real mathematical proofs, with the aim of obtaining new interesting elementary proofs. While a fully automated treatment proved successful for mathematical proofs of moderate complexity (e.g. the "tape proof" [BHL$^+$06] [AZ99] and the "lattice proof" [HLWWP08b]), more complex mathematical proofs required an interactive use of *CERES*. This way, Fürstenberg's proof of the infinitude of primes (using topological arguments) was successfully analyzed [BHL$^+$08]. The resulting cut-free proof contains Euclid's construction of primes. Fürstenberg's proof was formalized as an infinite sequence of **LK**-proofs. Even in its interactive use *CERES* proved to be superior to reductive cut-elimination due to additional structural information provided by the characteristic clause set. Like the reductive cut-elimination method, *CERES* is also not applicable in the presence of induction. This was one of the motivating reasons *CERES* to be extended to $CERES_s$, a cut elimination method for schematic proofs. A schematic proof is a way of finitely representing an infinite sequence of proofs. Such a concept is capable of doing cut elimination in presence of

implicit induction. Furthermore, the analysis of the Fürstenberg's proof of the infinitude of prime numbers revealed that the most famous automated theorem provers, such as Prover9 and Vampire, fail to refute the clause set of the proof for more than three primes. This was also a motivation for searching another approach and developing $CERES_s$.

CERES is a cut-elimination method based on resolution. The method roughly works as follows: from the input proof φ of a sequent S a clause term is extracted and evaluated to an unsatisfiable set of clauses $\mathrm{CL}(\varphi)$, the characteristic clause set. A resolution refutation γ of $\mathrm{CL}(\varphi)$, which is obtained using a first-order theorem prover, serves as a skeleton for an (atomic cut normal form) *ACNF* - a proof of S which contains at most atomic cuts. $CERES_s$ is based on a similar concept. The advantage of the latter in this respect is that it extracts a schematic clause set of constant length. In contrast, if for a concrete instance of the proof we apply *CERES* we get clause sets with different length (even with an exponential growth). The computation of the *ACNF* in $CERES_s$ is done semi-automatically because of the undecidability of the unification problem for the schematic first order logic. The method $CERES_s$ has been implemented in the **GAPT** framework [LWWP$^+$] and is a useful tool in automated proof mining, thus contributing to an experimental culture of computer-aided proof analysis in mathematics. The thesis is organized as follows. The second chapter contains the preliminaries. The language for first-order schemata is defined and the calculus **LKS** for first-order schemata is presented. Then, the formal description of the method *CERES* is explained. The third chapter is devoted to the method $CERES_s$. The fourth chapter is devoted to the **GAPT** framework and, more specifically, to the subset which is related with the implementation of the method $CERES_s$. Also a detailed description of the general architecture and the data-structures is given. In the fifth chapter the grammar of the proof schemata input language *SHLK* is defined and an algorithm for generating propositional proof schemata. The main contribution of this thesis is presented in chapter 6. Several key algorithms are described. The outcome of the algorithms is presented in chapter 7 where we show the whole automation process in detail.

Chapter 2

Preliminaries

Before going deeply into the method $CERES_s$ and the algorithms used for its automation, we give some basic definitions and notations.

2.1 Language for propositional schemata

The aim of this section is to introduce a notion of schemata for propositional formula [ACP09], a key notion needed to introduce propositional proof schemata. We start by first introducing the syntax. The main goal is to define formally a language that can specify an infinite sequence of first-order formulas by a finite term. In order to make clear the difference, we partition the universe of objects by using a 3-sorted logic. The sorts are ω, ι and o. They are also called basic types. Types are objects of a syntactic nature that are assigned to lambda terms. We create types as usual : if τ_1, τ_2 are types, then $\tau_1 \to \tau_2$ is a type. We use left associativity, i.e. $\tau_1 \to \tau_2 \to \cdots \to \tau_n$ means $\tau_1 \to (\tau_2 \to (\cdots \to \tau_n)))$. Objects of type o are called boolean individuals, objects of type ω are called arithmetical individuals and objects of type ι are called first-order individuals.

We assume countably infinite and pairwise disjoint sets V_a of *index variables* of type ω (intended to be interpreted over $\mathbb{N}$), V_1 of *first-order variables* of type ι, V_2 of *second-order variables* of type $\omega \to \iota$ and a countably infinite set of *propositional symbols* of type $\omega \to o$. As we will see later in this section, index variables can be free or bound. Free index variables are called *parameters*. For the sake of readability we will use k, m, n as parameters, i, j as bound index variables, the greek letters α, β as natural numbers and $a, b, \ldots$ as linear arithmetic expressions.

Definition (arithmetic expression) An *arithmetic expression* is a term of

finite type ω built as usual on the signature $\{0, s, +\} \cup V_a$, where s is interpreted as a *successor function* which has type $\omega \to \omega$ and there is at most one index variable in the term. If an arithmetic expression a does not contain variables it is called *ground*. The binary symbol "+" is interpreted as a usual arithmetical *addition* which has a type: $\omega \to \omega \to \omega$. The *numerals* $\bar{0}, \bar{1}, \bar{2}, \ldots$ denote the terms $0, s(0), s(s(0)), \ldots$ respectively and they represent the natural numbers.

Definition (regular arithmetic expression) Let t is a linear arithmetic expression. If t is equivalent (Presburger arithmetic) to an expression of the form $s^{\alpha}(k)$, where $k \in V_a \cup \{0\}$, $\alpha \in \mathbb{N}$, then t is called a regular arithmetic expression.

Example: $k + \alpha$ is a regular arithmetic expressions, but $n + i$, $2.k + \alpha$ and $s(n) + s(s(i))$ are not considered as regular arithmetic expressions.

We make use of the standard term-rewriting system for arithmetic expressions:

- $a + 0 \to a$
- $a + s(b) \to s(a + b)$

It is known to be confluent and terminating for arithmetic expressions. For simplicity we will identify such expressions with their *normal forms* $s^{\alpha}(k)$, where k is a parameter or 0.

Definition (Indexed proposition) An expression of the form P_a, where a is an arithmetic expression of type ω and P a propositional symbol of type $\omega \to o$, is called a *(linear) indexed proposition*. If a is a regular arithmetic expression, then P_a is called a regular *indexed proposition*

Example: $P_{n+\bar{3}}$ is a linear indexed proposition, but P_{n+k} is not, where $n, k \in V_a$

Definition (propositional formula schemata) We define formula schemata inductively in the following way:

- An indexed proposition is a formula schema.

- If ϕ_1 and ϕ_2 are formula schemata, then so are $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$ and $\neg\phi_1$.
- If ϕ is a formula schema, a, b are arithmetic expressions and i is an index variable not bound in ϕ, then $\bigwedge_{i=a}^{b} \phi$ and $\bigvee_{i=a}^{b} \phi$ are formula schemata such that i is bound in both formula schemata.

The binary connectives $\vee, \wedge, \rightarrow$ are are interpreted as a constants of type $o \rightarrow o \rightarrow o$. The $\neg$ is of type $o \rightarrow o$, the iterations $\bigwedge$ and $\bigvee$ are of type $\omega \rightarrow \omega \rightarrow (\omega \rightarrow o) \rightarrow o$. We denote formula schemata by $A, B, \ldots$. The notation $A(k)$ is used to indicate a parameter k in A. Then $A(a)$ denotes $A\{k \leftarrow a\}$.

Definition (Iterated schemata) A formula schema which has as bounds regular arithmetic expressions over the same parameter k is called an *iterated schema*.

Remark: *From now on when we say a formula schema we mean an iterated formula schema.*

Example: $P_0 \vee \bigwedge_{i=0}^{k+1}(P_i \rightarrow P_{i+1})$ is a formula schema with a parameter k, but $\bigwedge_{i=1}^{k+n} P_i$ is not (because the upper bound is not a regular arithmetic term).

Definition (Semantics) An interpretation of the schema language is a function I mapping an element of V_a to value in $\mathbb{N}$ and each indexed proposition to $\{true, false\}$. I is extended as usual into a function mapping all arithmetic expressions to natural numbers. The semantics $[\varphi]_I$ of a formula schema φ under the interpretation I is defined inductively:

- $[P_k]_I = P_{I(k)}$
- $\neg[\varphi]_I = true$ iff $[\varphi]_I = false$
- $[\varphi_1]_I \vee [\varphi_2]_I = true$ iff $[\varphi_1]_I = true$ or $[\varphi_2]_I = true$
- $[\varphi_1]_I \wedge [\varphi_2]_I = true$ iff $[\varphi_1]_I = true$ and $[\varphi_2]_I = true$
- $[\bigvee_{i=a}^{b} \varphi]_I = true$ iff there is an integer α, s.t. $I(a) \leq \alpha \leq I(b)$ and $[\varphi]_{I\{i/\alpha\}} = true$
- $[\bigwedge_{i=a}^{b} \varphi]_I = true$ iff for every integer α , s.t. $I(a) \leq \alpha \leq I(b)$ and $[\varphi]_{I\{i/\alpha\}} = true$

Definition A *substitution* σ is a function mapping every free index variable to an arithmetic expression. We write $[a_1/i_1, \ldots, a_k/i_k]$ for the substitution mapping respectively $i_1, \ldots, i_k$ to $a_1, \ldots, a_k$. The application of a substitution σ to a schema (or arithmetic expression) a is defined as usual and denoted by $a\sigma$. Notice that if a is an arithmetic expression and σ a substitution mapping every variable in a to a ground term (i.e. a term with no index variable) then $a\sigma$ is a ground numeral.

Definition (Satisfiable) A formula schema φ is satisfiable iff there exists and interpretation I, s.t. $[\varphi]_I = true$.

The condition of having at most one index variable in the linear arithmetic expressions is very restrictive. As we will see later, it plays a key role of the definition of schematic proofs. On other hand, if we want to guarantee a decidability of the satisfiability problem for the propositional fragment of formula schemata we need stronger restrictions, namely we need the notion of *regular schemata* [ACP11]. The undecidability of the satisfiability problem for non-regular schemata can be shown by a tricky reduction [Coo04] to the Post correspondence problem. On the other hand, analyzing interesting mathematical schematic proofs requires more indices in the propositional case or at least first-order logic with at least one index. In this thesis we will not consider regular schemata because it is not essential for the definition of the method $CERES_s$.

Definition (Sequent) Let Γ and Δ are multisets of propositional formula schemata. Then $\Gamma \vdash \Delta$ is called *sequent* and has the meaning of the formula $A_1 \wedge \cdots \wedge A_n \rightarrow B_1 \vee \cdots \vee B_m$, for $A_i \in \Gamma$ and $B_j \in \Delta$, where $i = 1, \ldots, n$ and $j = 1, \ldots, m$. A sequent is called *atomic* if all elements of Γ and Δ are indexed propositions (or atoms in the case of first-order logic).

Definition (Clause) Let $\Gamma \vdash \Delta$ is a sequent. If Γ and Δ are multisets of indexed propositions , then $\Gamma \vdash \Delta$ is called a *clause*.

Definition (p-Sequent) Let S be a sequent such that $n\colon \omega, x_1\colon \iota, \ldots, x_\alpha\colon \iota$ are the only free variables which occur in it, for $\alpha \geq 0$. Then S is called a *p-sequent* and it is denoted by $S(n, x_1, \ldots, x_\alpha)$.

Definition (Simple sequent) Let S be a sequent such that $n\colon \omega$ is the only

free-variables which occur in it. Then S is called a *simple sequent* and it is denoted by $S(n)$.

Definition (Composition) Let $C = \Gamma_1 \vdash \Delta_1$ and $D = \Gamma_2 \vdash \Delta_2$ are clauses. Then $C \circ D = \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$.

2.2 Language for first-order schemata

In order to investigate interesting and non-trivial sequences of uniform mathematical proofs, we have to extend the current formalism to first-order logic, but we still restrict it to a schema with one free index variable (parameter). This restriction comes also from another reason, namely the extraction of a schematic clause set (a set of clauses which contain specific information for the cut formulas) from a proof derivation. It is not yet clear how its extraction is performed in the case of more than one parameter. As in the propositional case, its main purpose is to allow a specification of an infinite set of first-order formulas by finite terms.

Let V_1 be a set of first-order variables of type ι, V_2 be a set of second-order variables of type $\omega \to \iota$, $\mathcal{F}^n$ a set of n-ary function symbols. Let $\mathcal{F}_a^n$ be a set of $(n+1)$-ary defined function symbols. We denote $\mathcal{F}_a = \bigcup \mathcal{F}_a^n$. Over the defined symbols we assume a strict ordering $<$ in order to guarantee termination. For every defined symbol $f \in \mathcal{F}_a^n$ we assume that its type is $\omega \to \tau_1 \to \cdots \to \tau_n \to \iota$.

Definition (s-term) The *s-terms* are defined inductively as follows:

- $\bar{0}$ is a term (of type ω)
- variables from V_1 are terms (of type ι)
- index variables from V_a are terms (of type ω)
- if t is a term of type ω, then $s(t)$ is a term (often written $t+1$) of type ω, where s is of type $\omega \to \omega$
- if $g \in \mathcal{F}^n$ and t_i are terms, for $i = 1, \ldots, n$, then $g(t_1, \ldots, t_n)$ is a term
- if $f \in \mathcal{F}_a^{n+1}$ is a defined symbol, t_i are terms, for $i = 1, \ldots, n$, and a is an arithmetic expression, then $f(a, t_1, \ldots, t_n)$ is a term. Always we associate such f with a term-rewriting system:

- $f(\bar{0}, x_1, \ldots, x_n) \to s$
- $f(k+1, x_1, \ldots, x_n) \to t[f(k, x_1, \ldots, x_n)]$,

  where s is a term with variables in $\{x_1, \ldots, x_n\}$ and t is a term with variables in $\{k, x_1, \ldots, x_n\}$ and for any defined term $g(a, \vec{t})$ occurring in t, either $g = f, a = k$ and $\vec{t} = x_1, \ldots, x_n$, or $g > f$.

To denote that an expression t rewrites to an expression s in finitely many steps we write $t \twoheadrightarrow s$. Since this definition has a primitive recursive nature, each term has a unique normal form.

Definition (first-order formula schemata) *First-order schemata* are built inductively over the set of s-terms using a set of predicate symbols, the standard logical connectives $\neg, \vee, \wedge$ and $\to$, the quantifiers $\forall$ and $\exists$ and the iterative connectives $\bigvee_{i=a}^{b}$ and $\bigwedge_{i=a}^{b}$.

Remark: *We do not allow quantification over variables of type* ω.

The *semantics* of the first-order schemata is defined in the following way: interpretation I is defined as usual, with the additional constraint that the terms of type ω are interpreted as natural numbers, and extended to schemata of formulas as in the propositional case.

Having defined symbols in our language increases its expressivity because we can easily formalize primitive recursive functions. On other hand this results in some limitations such that we loose the decidability of the term unification problem. The rewriting rules which define the s-terms give us a language for the primitive recursive functions. The programs over this language are called $LOOP2$ programs [BL74]. It is well known that such a programs always halt.

Theorem. The unification problem of s-terms is undecidable.

Proof: We reduce the unification problem of s-terms to the Halting problem [Tur36]. The functions computable with $LOOP2$ programs for which there exists a primitive recursive enumeration $\psi : \mathbb{N} \to \mathbb{N}$ are called elementary. That means that:

$$\psi(n, m) = \text{ result of LOOP-2 program nr. } n \text{ on input } m.$$

By E_k^n we are denoting the elementary functions $\mathbb{N}^n \to \mathbb{N}^k$. It can be shown that there is a universal Turing machine such that its halting predicate

$T(n,m,k)$ is elementary [BL74]. $T(n,m,k) = true$ iff the program number n stops on input m in less than k steps. Let $f \in E^2_1$. Then there exists a primitive recursive function h s.t.

$$\psi(h(n),m) = f(n,m) \text{ for all } n,m \in \mathbb{N}.$$

Now we define a function $g\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ by:

$$\begin{aligned} g(n,k) &= k+1 \text{ if } \neg T(n,n,k) \\ &= 0 \text{ otherwise.} \end{aligned}$$

Hence, $g \in E^2_1$. By definition of g we obtain:

$$n \in \bar{K} \leftrightarrow \forall k.\neg T(n,n,k) \leftrightarrow \forall k.g(n,k) \neq 0$$

where K is the halting problem, i.e. the set:

$$\{(n,n)| \text{ the program nr. } n \text{ will stop on input } n\}$$

As ψ is a primitive recursive and effective enumeration of E^1_1 there exists a primitive recursive function h s.t. $\psi(h(n),k) = g(n,k)$ for all $n,k \in \mathbb{N}$, such that:

$$n \in \bar{K} \leftrightarrow \forall k.\psi(h(n),k) \neq \bar{0}.$$

Now let $f_\psi \in F^1_s$ the representation of ψ, $f_h \in F^0_s$ that of h. Then deciding the unification problems $\exists y.\psi(\bar{m},y) = \bar{0}$ for number constants $\bar{m}$ and $n \in \mathbb{N}$ is equivalent to the equivalence:

$$\exists y.f_\psi(f_h(\bar{n}),y) = \bar{0} \leftrightarrow n \in K$$

This way we obtained a decision procedure for K, which obviously does not exists, because K is undecidable. □

2.3 Sequent calculus LKS for first-order proof schemata

After defining the notions of schematic terms and formulas, now we are ready to define the notion of proof schemata, so that we can interpret the schematic first-order language and state a soundness result. The notion of schematic

proof is based on the usual classical sequent calculus **LK** : it is a finite derivation tree of sequents which are formed according to the rules of the proof system **LKS**. The sequent calculus for first-order proof schemata extends the Gentzen's sequent calculus **LK** in two ways. First, rules that iterate over formula schemata were defined. Still the length of the sequent in a schematic proof remains constant. This is essential for keeping track on the cut-ancestors as we will see later. Second, we have a rule which operates in a term level within a formula - it applies finitely many reductions to all s-terms according to the given confluent and terminating term-rewriting systems. The axioms may be sequents which contain atoms or indexed predicates only. We assume that the lower and upper bounds of the iterations $\bigwedge$ and $\bigvee$ are always 0 and $n + \alpha$, respectively. The inference rules are:

1.Logical rules:

- $\wedge$ introduction:

$$\frac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \wedge : l1 \qquad \frac{B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \wedge : l2$$

$$\frac{\Gamma \vdash \Delta, A \qquad \Pi \vdash \Lambda, B}{\Gamma, \Pi \vdash \Delta, \Lambda, A \wedge B} \wedge : r$$

- $\vee$ introduction:

$$\frac{A, \Gamma \vdash \Delta \qquad B, \Pi \vdash \Lambda}{A \vee B, \Gamma, \Pi \vdash \Delta, \Lambda} \vee : l$$

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B} \vee : r1 \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B} \vee : r2$$

- Equivalences: $A_0 \equiv \bigvee_{i=0}^{0} A_i$ and $(\bigvee_{i=0}^{n} A_i) \vee A_{n+1} \equiv \bigvee_{i=0}^{n+1} A_i$. Analogously $\bigwedge$ is defined.

- $\neg$ introduction:

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta} \neg : l \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A} \neg : r$$

2. Structural rules:

- Weakening :

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

- Contraction:

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

3. Cut rule:

$$\frac{\Gamma \vdash \Delta, A \qquad A, \Pi \vdash \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\, cut$$

4. Quantifier rules:

- existential:

$$\frac{A\{x \leftarrow \alpha\}, \Gamma \vdash \Delta}{(\exists x)A, \Gamma \vdash \Delta}\, \exists\colon l \qquad \frac{\Gamma \vdash \Delta, A\{x \leftarrow t\}}{\Gamma \vdash \Delta, (\exists x)A}\, \exists\colon r$$

- universal:

$$\frac{A\{x \leftarrow t\}, \Gamma \vdash \Delta}{(\forall x)A, \Gamma \vdash \Delta}\, \forall\colon l \qquad \frac{\Gamma \vdash \Delta, A\{x \leftarrow \alpha\}}{\Gamma \vdash \Delta, (\forall x)A}\, \forall\colon r$$

6. Definition rule:

$$\frac{S[t]}{S[t']} \twoheadrightarrow$$

where $t \twoheadrightarrow t'$.

The first-order variable α of sort ι in the $\forall$: r and $\exists$: l rules is called *eigen variable*. It should be free and it should not appear in Γ and Δ. We do not allow quantification over parameters, i.e. over variables of sort ω. The term t in $\forall$: l and $\exists$: r rules is an arbitrary term containing only free first-order variables. The formulas A and B in the upper sequent(s) of an inference are called *auxiliary* formulas. The formulas in the end-sequent of the inference are called main formulas (Γ and Δ are set of formulas, not formulas in this sense).

Now we will define the concepts of a proof derivation (or simply proof). We assume that there are infinitely many countable symbols for denoting pairs of **LKS**-proofs $\varphi, \psi, \chi, \phi, \dots$ which are called *proof symbols* and they come with a pair of **LKS**-proofs. The first element of such a pair will be called a base-case proof (denoted by $\pi_1, \pi_2, \dots$), the second element - a step-case proof (denoted by $\nu_1(k+1), \nu_2(k+1), \dots$). We assume that $<$ is a given strict order between the proof symbols.

Definition (proof-link) Let t is a arithmetic expression and $S(n, x_1, \dots, x_\alpha)$ be a p-sequent. Then the expression: $\dfrac{\varphi(t, a_1, \dots, a_\alpha)}{S(t, a_1, \dots, a_\alpha)}$ is called a *proof-link*, where a_i is a free first-order variable of type ι, for $i = 1, \dots, \alpha$ and t is an arithmetical term.

Definition (schematic proof) A binary tree is called a *proof* if each node is a sequent obtained by applying **LKS**-rules from the leafs to the root. The leafs must contain either atomic sequents (the sequent contains only atoms/indexed predicates) or proof-links.

Definition (proof schemata) Let ψ be a proof symbol and k is an integer variable. Then a *proof schema pair* for ψ_i is a pair of proofs $\langle \psi_i(0), \psi_i(k+1) \rangle$, where $\psi_i(0)$ is a ground **LKS**-proof (called base-case) with end-sequent $S_i(0)$ and $\psi_i(k+1)$ is an **LKS**-proof (called step-case) with end-sequent $S_i(k+1)$. The proof $\psi_i(k+1)$ may contain proof-links of the form:

- $\dfrac{\psi_i(k, a_1, \dots, a_\alpha)}{S_i(k, a_1, \dots, a_\alpha)}$
- $\dfrac{\psi_j(t, a_1, \dots, a_\alpha)}{S_j(t, a_1, \dots, a_\alpha)}$, where $i < j$ and t is an integer term with a free indexed variable k.

Intuitively speaking, the proof-link can be thought of as a subproof. Indeed, for a given evaluation I, the term $I(t)$ is a ground term and the proof $\varphi(I(t))$, where k is a free indexed variable in t, is a ground (sub)proof with end-sequent $[S]_I$. Note that a ground proof may contain the iterations $\bigwedge$ and $\bigvee$ which lower and upper bounds are ground.

As the reader has already noticed, the first kind of proof-links refers to the proof itself with an index which is always smaller then the current one. Whereas the second proof-link refers to another proof whose proof symbol is strictly bigger than ψ_i with respect to the proof symbol ordering $<$. Nevertheless, in the second case we do not require a restriction over the parameter of the called proof. The conditions in the above definition are crucial for defining the notion of the proof schema as a primitive recursion of schematic proofs. Furthermore, they guarantee that no looping is possible, i.e. the mutual recursion does not take place.

For a proof schemata ψ we assume an identification between formula occurrences in the end-sequents of $\psi(0)$ and $\psi(k+1)$ so that we can speak of occurrences in the end-sequent of ψ. This will allow us to define some crucial for the method *CERES*$_s$ relations between the formula occurrences of the proof-tree such as the ancestor relation which is formally defined in the next chapter.

Example: Let $\Psi = \langle \psi_1, \psi_2 \rangle$ is a propositional proof schema, where ψ_i is the pair $(\pi_i, \nu_i(k+1))$ for $i = 1, 2$, where the pair for ψ_2 is defined as follows:

- π_2:

$$\dfrac{\dfrac{p_0 \vdash p_0}{\bigvee_{i=0}^{0} p_i \vdash p_0} \equiv : l}{\bigvee_{i=0}^{0} p_i \vdash \bigvee_{i=0}^{0} p_i} \equiv : r$$

- $\nu_2(k+1)$:

$$\dfrac{\dfrac{\dfrac{\overset{(\nu_2(k))}{\bigvee_{i=0}^{k} p_i \vdash \bigvee_{i=0}^{k} p_i} \qquad p_{k+1} \vdash p_{k+1}}{\bigvee_{i=0}^{k} p_i \vee p_{k+1} \vdash \bigvee_{i=0}^{k} p_i, p_{k+1}} \vee : l}{\bigvee_{i=0}^{k} p_i \vee p_{k+1} \vdash \bigvee_{i=0}^{k+1} p_i} \equiv : r}{\bigvee_{i=0}^{k+1} p_i \vdash \bigvee_{i=0}^{k+1} p_i} \equiv : l$$

and the pair for ψ_1 is defined as follows:

- π_1:

$$\dfrac{\dfrac{p_0 \vdash p_0}{\neg p_0, p_0 \vdash}\,\neg\colon l \qquad p_1 \vdash p_1}{p_0, \neg p_0 \vee p_1 \vdash p_1}\,\vee\colon l$$

- $\nu_1(k+1)$:

$$\dfrac{\dfrac{\overset{(\nu_1(k))}{p_0, \bigwedge_{i=0}^{k}(\neg p_i \vee p_{i+1}) \vdash \bigvee_{i=0}^{k+1} p_i} \qquad \dfrac{\dfrac{\overset{(\nu_2(k+1))}{\bigvee_{i=0}^{k+1} p_i \vdash \bigvee_{i=0}^{k+1} p_i}\, w\colon l}{\neg p_{k+1}, \bigvee_{i=0}^{k+1} p_i \vdash \bigvee_{i=0}^{k+1} p_i} \qquad p_{k+2} \vdash p_{k+2}}{\bigvee_{i=0}^{k+1} p_i, \neg p_{k+1} \vee p_{k+2} \vdash p_{k+2}, \bigvee_{i=0}^{k+1} p_i}\,\vee\colon l}{p_0, \bigwedge_{i=0}^{k}(\neg p_i \vee p_{i+1}), \neg p_{k+1} \vee p_{k+2} \vdash \bigvee_{i=0}^{k+1} p_i, p_{k+2}}\,cut$$
$$\dfrac{p_0, \bigwedge_{i=0}^{k}(\neg p_i \vee p_{i+1}), \neg p_{k+1} \vee p_{k+2} \vdash \bigvee_{i=0}^{k+1} p_i, p_{k+2}}{p_0, \bigwedge_{i=0}^{k+1}(\neg p_i \vee p_{i+1}) \vdash \bigvee_{i=0}^{k+1} p_i, p_{k+2}}\,\wedge\colon l$$
$$\dfrac{p_0, \bigwedge_{i=0}^{k+1}(\neg p_i \vee p_{i+1}) \vdash \bigvee_{i=0}^{k+1} p_i, p_{k+2}}{p_0, \bigwedge_{i=0}^{k+1}(\neg p_i \vee p_{i+1}) \vdash \bigvee_{i=0}^{k+2} p_i}\,\equiv\colon r$$

Remark: *The unary inference rules $\equiv_l$ and $\equiv_r$ which are used above are not formally in the calculus. We define them as macro-rules in order to improve the readability. For an indexed proposition C_i and arithmetical terms a, b where $a < b$, the rules are defined as follows, respectively:*

$$\dfrac{\dfrac{\dfrac{C_a, \bigwedge_{i=a+1}^{b} C_i, \Gamma \vdash \Delta}{C_a \wedge \bigwedge_{i=a+1}^{b} C_i, \bigwedge_{i=a+1}^{b} C_i, \Gamma \vdash \Delta}\,\wedge\colon l1}{C_a \wedge \bigwedge_{i=a+1}^{b} C_i, C_a \wedge \bigwedge_{i=a+1}^{b} C_i, \Gamma \vdash \Delta}\,\wedge\colon l2}{\bigwedge_{i=a}^{b} C_i, \Gamma \vdash \Delta}\,c\colon l$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \Delta, C_a, \bigvee_{i=a+1}^{b} C_i}{\Gamma \vdash \Delta, C_a \vee \bigvee_{i=a+1}^{b} C_i, \bigvee_{i=a+1}^{b} C_i}\,\vee\colon r1}{\Gamma \vdash \Delta, C_a \vee \bigvee_{i=a+1}^{b} C_i, C_a \vee \bigvee_{i=a+1}^{b} C_i}\,\vee\colon r2}{\Gamma \vdash \Delta, \bigvee_{i=a}^{b} C_i}\,c\colon r$$

In the example above we have $\psi_1 < \psi_2$. In the step-case of ψ_1 we see the two kinds of proof-links : the one which refers to $\nu_1(k)$, i.e. to itself, and the one which refers to $\nu_2(k+1)$. The formulas in red are cut-ancestors. None of them comes from the weakening rules and therefore, according to the subformula property of **LK** it follows that these formulas are propagated somehow in the proof-links. How exactly this is done is a central topic of the next chapter. As it is also seen from the example we do not have positions of the formula occurrences in the sequent, i.e. the antecedent and the succedent parts of the sequent are multisets of formula occurrences. However, we have a positions in the proof derivation:

Definition (ancestor relation) Let φ be a proof and ζ is a node in the proof. Then $\varphi.\zeta$ denotes the subproof of φ with root ζ. A proof node μ is called *immediate successor* of ζ if μ is the occurrence of a premise of an inference

rule ρ in φ with conclusion ζ. Let ρ_1 and ρ_2 be arbitrary unary and binary inference in an **LKS** proof:

$$\frac{\Pi_1, \Gamma_1 \vdash \Delta_1, \Lambda_1}{\Pi, \Gamma_1 \vdash \Delta_1, \Lambda}\,\rho_1 \qquad \frac{\Pi_1, \Gamma_1 \vdash \Delta_1, \Lambda_1 \qquad \Pi_2, \Gamma_2 \vdash \Delta_2, \Lambda_2}{\Gamma_1, \Gamma_2, \Pi \vdash \Delta_1, \Delta_2, \Lambda}\,\rho_2$$

where Π_i, Δ_i (Π, Δ) denote the auxiliary formulas in the premise (principal formulas) in the conclusion. We define a relation R such that $v \in \Gamma_i$, $w \in \Delta_i$ are positions of a formula f in the premise and $v' \in \Gamma_i$, $w' \in \Delta_i$ are positions of the same formula in the conclusion of the inferences. Then we define vRv', wRw'. Let $v'_1 \in \Pi$ be the principal formula of ρ_1 (ρ_2) and v_1 (v_1, v_2) is (are) the auxiliary formula(s) of ρ_1 (ρ_2). Then $v_1 R v'_1$ $(v_1 R v'_1, v_2 R v'_1)$. $\mathcal{R}$ does not have other pairs of formula occurrences other than those mentioned. The reflexive and transitive closure of R is called *ancestor relation* in φ.

Remark: *We can think of a formula occurrence as an object which is a formula in the antecedent or succedent part of the sequent together with its ancestors. This difference will be exploited in the next chapters where we describe the algorithms and the datastructures in the system CERES$_s$. When we talk about a formula in a proof, we usually mean a formula occurrence.*

2.4 Resolution calculus for first-order proof schemata

In this section we define a very strong calculus for resolution schemata which is capable to deal with a schematic description of a sequence of clauses. In the previous section we introduced the set V_2. The need to use such variables appears, because we need to construct somehow an enumeration for the first-order index variables which plays a role when the parameter is instantiated with a concrete instance.

Definition (clause) Let $p_1, \ldots, p_\alpha$ and $q_1, \ldots, q_\beta$ be schematic atomic formulas; then $p_1, \ldots, p_\alpha \vdash q_1, \ldots, q_\beta$ is called a *clause*. A clause is called *arithmetically ground* (shorthand:*a*-ground) if it does not contain arithmetic variables. An arithmetically ground clause is in *normal form* if it is irreducible under the defining rewrite rules. The set of clause schemata is denoted by CS.

We introduce *clause symbols* and denote them by $c, c', c_1, c_2, \ldots$ for defining clause schemata. Each clause symbol is associated with a unique arity. The type of the first argument is always an arithmetic expression. Clause variables are denoted by $X, Y, X_1, Y_1, \ldots$ and the set of all clause variables is denoted by V_c.

Definition (clause schema)

- Clauses and clause variables are clause schemata.
- If C_1 and C_2 are clause schemata then $C_1 \circ C_2$ is a clause schema.
- Furthermore, let c be a clause symbol of arity $\beta+\gamma+1$, a an arithmetic term, $x_1, \ldots, x_\beta \in V_2$ and $X_1, \ldots, X_\gamma \in V_c$. Then $c(a, x_1, \ldots, x_\beta, X_1, \ldots, X_\gamma)$ is a clause schema w.r.t. the rewrite system $\mathcal{R}(c)$, where $\mathcal{R}(c)$ is of the form:

$$\begin{array}{rcl} c(\bar{0}, x_1, \ldots, x_\beta, X_1, \ldots, X_\gamma) & \rightarrow & C \\ c(k+1, x_1, \ldots, x_\beta, X_1, \ldots, X_\gamma) & \rightarrow & c(k, x_1, \ldots, x_\beta, X_1, \ldots, X_\gamma) \circ D \end{array}$$

  where C is an arithmetically ground clause schema such that $V(C) \subseteq \{x_1, \ldots, x_\beta, X_1, \ldots, X_\gamma\}$ and D is a clause with $V(D) \subseteq \{x_1, \ldots, x_\beta, k\}$.

Example: Let $\sigma \in F^3_s, g \in F^1, x \in V_2,\ l \in V_a, X \in V_c$ with the corresponding rewrite rules $\mathcal{R}(\sigma)$:

$$\begin{array}{rcl} \sigma(\bar{0}, x, l) & \rightarrow & x(l) \\ \sigma(k+1, x, l) & \rightarrow & g(\sigma(k, x, l)) \end{array}$$

and let $c(n, x, X)$ be a clause schema for $\mathcal{R}(c)$ consisting of the rules:

$$\begin{array}{rcl} c(0, x, X) & \rightarrow & X \circ (\ \vdash P(\sigma(\bar{0}, x, \bar{0}))) \\ c(k+1, x, X) & \rightarrow & c(k, x, X) \circ (\ \vdash P(\sigma(k+1, x, k+1))) \end{array}$$

The normal forms of $c(n, x, X)$ for $\{n \leftarrow \alpha\}$ are just the clause schemata:

$$X \vdash P(x(0)), P(g(x(1))), \ldots, P(g^\alpha(x(\alpha)))$$

Definition (c-substitution) A *c-substitution* is a mapping with a finite domain which maps variables from V_c to clauses in CS.

Definition (semantics of clause schema) Let C be a clause schema. Let ϑ be an arithmetically ground s-substitution with $V_a(C) \cup V_1(C) \subseteq dom(\vartheta)$ and λ be a c-substitution without clause variables in the range and $V_c(C) \subseteq dom(\lambda)$. We define the interpretation of C under (ϑ, λ) as:

$$v_{\mathrm{c}}(\vartheta, \lambda, C) = ((C\lambda)\vartheta)\downarrow$$

where $\downarrow$ means normalization.

Example: let $c(n, x, X)$ be the clause schema from the previous example, $\vartheta = \{n \leftarrow \alpha\}$ and $\lambda = \{X \leftarrow Q(x(n))\}$. Then:

$$v_{\mathrm{c}}(\vartheta, \lambda, C) = Q(x(\alpha)) \vdash P(x(0)), P(g(x(1))), \ldots, P(g^{\alpha}(x(\alpha)))$$

The notion of the clause schema makes it possible to describe a sequence of clause sets $CL(\varphi_1), \ldots, CL(\varphi_n)$ for a given instance of the parameter n. In order to guarantee the growing length of this sequence (i.e. adding new clauses to a set), we will introduce clause-set variables $\xi_1, \xi_2, \ldots$ over finite sets of clauses. The set of all clause-set variables is denoted by V_{clset}. The set CST of clause-set terms is defined inductively as follows:

Definition (clause-set term)

- if $\xi \in V_{\mathrm{clset}}$, then $\xi \in \mathrm{CST}$
- if $C \in CS$, then $[C] \in \mathrm{CST}$
- if $t_1, t_2 \in \mathrm{CST}$, then $t_1 \oplus t_2 \in \mathrm{CST}$ and $t_1 \otimes t_2 \in \mathrm{CST}$

Definition (clause-set term evaluation) Let t be a clause-set term s.t. $V_a(t) \cup V_c(t) \cup V_{\mathrm{clset}}(t) = \emptyset$. Then we define the evaluation of t to a set of clauses in the standard way, where $\cup$ and $\times$ are the set-theoretical union and cartesian product:

- If $t = [C]$, then $|[C]| = \{C\}$

- If $t = t_1 \oplus t_2$, then $|t| = |t_1| \cup |t_2|$
- If $t = t_1 \otimes t_2$, then $|t| = |t_1| \times |t_2|$

As we will see later, the characteristic clause term - a structure which is the skeleton of the computation of the characteristic clause set, corresponds to the clause-set term . The evaluation of the clause term is exactly the clause set as one can notice this connection by looking at its semantics:

Definition (semantics of clause-set terms) Let t be a clause-set term with $V_c(t) = \{X_1, \ldots, X_\alpha\}$, $V_{\text{clset}}(t) = \{\xi_1, \ldots, \xi_\beta\}$ and $V_a(t) = \{n\}$. Let $C_1, \ldots, C_\alpha$ are clauses, $\vartheta = \{n \leftarrow \gamma\}$, $\lambda = \{X_1 \leftarrow C_1, \ldots, X_\alpha \leftarrow C_\alpha\}$ and $\mu = \{\xi_1 \leftarrow s_1, \ldots, \xi_\beta \leftarrow s_\beta\}$ (for clause-set terms $s_1, \ldots, s_\beta$ not containing clause-set variables). Then we define a semantic function v_{cst} by:

$$v_{\text{cst}}(\vartheta, \lambda, \mu, t) = |(t\mu\lambda\vartheta) \downarrow |$$

where $|.|$ is from the previous definition.

Example: Let c be the clause symbol from the previous example. Then

$$t\colon\ ([c(n, x, X)] \otimes [\vdash P(x(n))]) \oplus \xi$$

is a clause-set term. Let $\vartheta = \{n \leftarrow \alpha\}$, $\lambda = \{X \leftarrow \vdash\}$ and $\mu = \{\xi \leftarrow [P(\sigma(n, x, n)) \vdash]\}$. Then the evaluation $v_{\text{cst}}(\vartheta, \lambda, \mu, t)$ is:

$$\{P(g^\alpha(x(\alpha))) \vdash\} \cup \{\vdash P(x(0)), P(g(x(1))), \ldots, P(g^\alpha(x(\alpha))), P(x(\alpha))\}$$

Definition Let t be a clause-set term, $\xi_1, \ldots, \xi_\alpha$ in V_{clset}, and $s_1, \ldots, s_\alpha$ objects of appropriate type. Then $t\{\xi_1 \leftarrow s_1, \ldots, \xi_\alpha \leftarrow s_\alpha\}$ is called a clause-set term over $\{s_1, \ldots, s_\alpha\}$ (note that every ordinary clause set term is also a clause set term over any set $\{s_1, \ldots, s_\alpha\}$).

Example: Let t be the clause-set term:

$$t\colon\ ([c(n, x, X)] \otimes [\vdash P(x(n))]) \oplus \xi$$

from the previous example and s be an object of the type of CST. Then

$$t'\colon\ ([c(n, x, X)] \otimes [\vdash P(x(n))]) \oplus s$$

is a clause-set term over $\{s\}$.

Having defined the clause-set term, we can now define the clause-set schemata. As we will see later, this is essential for the definition of our notion of schematic resolution deduction.

Definition (clause-set schema) The symbols $d_0, d_1, \ldots$ will be reserved for denoting clause-set schemata. A *clause-set schema* is a tuple $\Delta\colon (d_1, \ldots, d_\alpha)$ together with sets of rewrite rules $\mathcal{R}(d_1), \ldots, \mathcal{R}(d_\alpha)$ such that for all $i = 1, \ldots, \alpha$ we define $\mathcal{R}(d_i)$ as:

$$\begin{aligned} d_i(\bar{0}, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma) &\rightarrow t_i^b \\ d_i(k+1, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma) &\rightarrow t_i^s \end{aligned}$$

where t_i^b is clause-set term over $\{d_j(r_j) \mid i < j \leq \alpha\}$ for some $r_j \in \mathrm{CST}$ and t_i^s is a clause-set term over $\{d_j(s_j) \mid i < j \leq \alpha\} \cup \{d_i(k)\}$ for some $s_j \in \mathrm{CST}$ and:

- $V(t_i^b) \subseteq \{x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma\}$
- $V(t_i^s) \subseteq \{x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma, k\}$

Definition (semantics of clause-set schemata) We extend v_{cst} to a function v_{cst}^*. Let $\Delta\colon (d_1, \ldots, d_\alpha)$ be a clause set schema, ϑ a substitution on $V_a(\Delta)$, λ a substitution on $V_c(\Delta)$ and ξ be a substitution on $V_{\mathrm{clset}}(\Delta)$. We define:

$$\begin{aligned} v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, d_\alpha(\bar{0}, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma)) &= v_{\mathrm{cst}}(\vartheta, \lambda, \xi, t_\alpha^b) \\ v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, d_\alpha(k+1, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma)) &= v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, t_\alpha^s) \end{aligned}$$

Note that t_α^b is a clause-set term. For $1 \leq i < \alpha$ we define:

$$\begin{aligned} v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, d_i(\bar{0}, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma)) &= v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, t_i^b) \\ v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, d_i(k+1, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma)) &= v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, t_i^s) \end{aligned}$$

The clause set schema defined by Δ w.r.t. $(\vartheta, \lambda, \mu)$ for $dom(\vartheta) = \{n\}$ is then defined as:

$$v_{\mathrm{cst}}^*(\vartheta, \lambda, \xi, d_1(n, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma))$$

A clause-set schema is called *unsatisfiable* if there exist λ and ξ such that for all α and $\vartheta_\alpha\colon \{n \leftarrow \alpha\}$ the clause set:

$$v^*_{\mathrm{cst}}(\vartheta_\alpha, \lambda, \xi, d_1(n, x_1, \ldots, x_\delta, X_1, \ldots, X_\beta, \xi_1, \ldots, \xi_\gamma))$$

is unsatisfiable.

Example: Let σ be defined by:

$$\begin{array}{rcl} \sigma(\bar{0}, x, l) & \to & x(l) \\ \sigma(Sk, x, l) & \to & g(\sigma(k, x, l)) \end{array}$$

where $c(n, x, X)$ is the previous example and $\sigma' \in F^1_s$ with the rewrite rules:

$$\begin{array}{rcl} \sigma'(\bar{0}) & \to & a \\ \sigma'(Sk) & \to & g(\sigma'(k)) \end{array}$$

Note that $\sigma'(n)\downarrow_\alpha$ evaluates to $g^\alpha(a)$. Furthermore we define the clause set schema $\Delta = (d_1, d_2)$ by:

- $\mathcal{R}(d_1)$:

$$\begin{array}{rcl} d_1(\bar{0}, x, X) & \to & (d_2(\bar{0}, x, X) \oplus \xi) \\ d_1(k+1, x, X) & \to & d_2(k+1, x, X) \oplus [c(k+1, x, X)] \end{array}$$

- $\mathcal{R}(d_2)$:

$$\begin{array}{rcl} d_2(\bar{0}, x, X) & \to & [P(a) \vdash\ ] \\ d_2(k+1, x, X) & \to & (d_2(k, x, X) \oplus [P(\sigma'(k+1)) \vdash\ ] \end{array}$$

Let $\vartheta = \{n \leftarrow \alpha\}$, $\lambda = \{X \leftarrow \vdash\}$ and $\mu = \{\xi \leftarrow [c(\bar{0}, x, X)]\}$. Then we have:

$$v^*_{\mathrm{cst}}(\vartheta, \lambda, \mu, d_1(n, x, X)) = \{\vdash P(x(0)), \ldots, P(g^\alpha(x(\alpha)));\ P(a) \vdash; \ldots; P(g^\alpha(a)) \vdash\}$$

Remark: *On contrary to the usual convention, as a separator between the elements of a set of clauses we use ";" instead of "," in order to avoid a*

confusion, because "," is already used to separate the formulas in the antecedent/succedent part of a sequent.

So far we have defined the notion of a schema clause and a clause-set schema which describes a sequence of set of clauses. The next step is to find a deductive formalism which can handle the clause-set schemata and to produce a resolution refutation out of it under some conditions. In general this problem is not trivial, because the set of values for the parameter is infinite and verifying that for each value of the parameter the set of leafs of the deduction tree is a subset of the schematic clause set for the same instance of the parameter, is a very hard task without having the notion of induction.

We continue with the description of a schematic resolution proof which is built over the notion of resolution term.

Definition (resolution term)

- clause schemata are resolution terms.
- Let s_1 and s_2 be resolution terms w.r.t. $\mathcal{R}_1$ and $\mathcal{R}_2$, and P be an indexed atom. Then $r(s_1; s_2; P)$ is a resolution term w.r.t. $\mathcal{R}_1 \cup \mathcal{R}_2$

Resolution terms define resolution deductions only if appropriate substitutions are applied to the clauses unifying atoms in clauses.

Definition (V_2-substitution schema) Let $x_1, \ldots, x_\alpha \in V_2$ and $t_1, \ldots, t_\alpha$ be s-terms such that $V_a(t_i) \subseteq \{n, k\}$ for $i = 1, \ldots, \alpha$ then:

$$\theta \colon \{x_1 \leftarrow \lambda k.t_1, \ldots, x_\alpha \leftarrow \lambda k.t_\alpha\}$$

is called a V_2-substitution schema (note that the terms t_i may contain arbitrary variables in V_2).

Every V_2-substitution schema evaluates to sequences of "ordinary" second order substitutions under an assignment for the parameter n. Indeed, if $\vartheta = \{n \leftarrow \beta\}$, then:

$$\theta_\beta = \theta\vartheta = \{x_1 \leftarrow \lambda k.(t_1)\downarrow_\beta, \ldots, x_\alpha \leftarrow \lambda k.(t_\alpha)\downarrow_\beta\}$$

Note that the $(t_i)\downarrow_\beta$ contain only k as arithmetic variable.

Definition (resolvent) Let $C \colon C_1 \vdash C_2,\ D \colon D_1 \vdash D_2$ be clauses with $V_a(\{C, D\}) = \emptyset$ and $V_c(\{C, D\}) = \emptyset$; let P be an atom. Then:

$$res(C, D, P) = C_1, D_1 \setminus P \vdash C_2 \setminus P, D_2$$

where $C \setminus P$ denotes the multiset of atoms in C after removal of all occurrences of P. The clause $res(C, D, P)$ is called a *resolvent* of C_1 and C_2 on P. In case P does not occur in C_2 and D_1 then $res(C, D, P)$ is called a pseudo-resolvent (note that inferring $res(C, D, P)$ from C and D is sound in any case).

Definition (resolution deduction) If C is a clause then C is a resolution deduction and $ES(C) = C$, where ES is the end-sequent of the deduction. If γ_1 and γ_2 are resolution deductions and $ES(\gamma_1) = D_1$, $ES(\gamma_2) = D_2$ and $res(D_1, D_2, P) = D$, where $res(D_1, D_2, P)$ is a resolvent, then $r(\gamma_1, \gamma_2, P)$ is a resolution deduction and $ES(r(\gamma_1, \gamma_2, P)) = D$.

Definition (refutation) Let t be a resolution deduction and $\mathcal{C}$ be the set of all clauses occurring in t. Then t is called a *resolution refutation* of $\mathcal{C}$ if $ES(t) = \vdash$.

A resolution deduction can be straightforwardly transformed to a tree:

Definition (to tree transformation) Any resolution deduction can easily be transformed into a resolution tree by the following transformation T:

- If $\gamma = C$ for a clause C then $T(\gamma) = C$.
- If $\gamma = r(\gamma_1, \gamma_2, P)$, $\varphi_1 = T(\gamma_1)$, $\varphi_2 = T(\gamma_2)$, $ES(\varphi_1) = C_1$, $ES(\varphi_2) = C_2$ and $res(C_1, C_2, P) = C$, then $T(\gamma)$ is:

$$\dfrac{\dfrac{(\varphi_1)}{C_1} \quad \dfrac{(\varphi_2)}{C_2}}{C}$$

The length of $T(\gamma)$ is polynomial in the length of γ as can be proved easily.

Example: Let γ is the resolution term:
$r(r(\vdash Q_0(x), P_0(x), P_1(x); P_1(x) \vdash; P_1(x)); Q_0(x), Q_1(x) \vdash; Q_0(x))$

$T(\gamma)$ is the resolution tree:

$$\dfrac{\dfrac{\vdash Q_0(x), P_0(x), P_1(x) \qquad P_1(x) \vdash}{\vdash Q_0(x), P_0(x)} \qquad Q_0(x), Q_1(x) \vdash}{\vdash P_0(x), Q_1(x)}$$

We define a notion of resolution proof schema in the spirit of **LK**-proof schemata.

Definition Let t be a resolution term, $X_1, \ldots, X_\alpha$ in V_c, and $s_1, \ldots, s_\alpha$ objects of appropriate type. Then $t\{X_1 \leftarrow s_1, \ldots, X_\alpha \leftarrow s_\alpha\}$ is called a resolution term over $\{s_1, \ldots, s_\alpha\}$.

Definition (resolution proof schema) A resolution proof schema over the variables $x_1, \ldots, x_\alpha \in V_2$ and $X_1, \ldots, X_\beta \in V_{\text{clset}}$ is a structure $((\rho_1, \ldots, \rho_\gamma), \mathcal{R})$ with $\mathcal{R}\colon \mathcal{R}_1 \cup \ldots \cup \mathcal{R}_\gamma$, where the $\mathcal{R}_i$ (for $0 \leq i \leq \gamma$) are defined as follows:

$$\begin{array}{rcl} \rho_i(0, x_1, \ldots, x_\alpha, X_1, \ldots, X_\beta) & \rightarrow & t_i^b \\ \rho_i(k+1, x_1, \ldots, x_\alpha, X_1, \ldots, X_\beta) & \rightarrow & t_i^s \end{array}$$

where:

- t_i^b is a resolution term over terms of the form $\rho_j(a_j, s_1, \ldots, s_\alpha, C_1, \ldots, C_\beta)$ for $1 \leq i < j$
- t_i^s is a resolution term over terms of the form $\rho_j(a_j, s_1, \ldots, s_\alpha, C_1, \ldots, C_\beta)$ and $\rho_i(k, s'_1, \ldots, s'_\alpha, C'_1, \ldots, C'_\beta)$ for $1 \leq i < j$

Definition (semantics of resolution proof schemata) A resolution proof schema R is called a *resolution deduction schema* from a clause-set schema Δ if there exist substitutions λ for V_c and μ for V_{clset} and a V_2-substitution schema θ s.t. for every ϑ_β of the form $\{n \leftarrow \beta\}$ $(\rho_1(n, \bar{x}, X_1, \ldots, X_\alpha)\lambda\theta\vartheta_\beta) \downarrow$ is a resolution deduction t_β from $v^*_{\text{cst}}(\vartheta_\beta, \lambda, \mu, d_1(n, \bar{x}, Y_1, \ldots, Y_\gamma, \xi_1, \ldots, \xi_\delta))$. If for all β $ES(\rho_1(\beta, \bar{x}, X_1, \ldots, X_\alpha)) = \vdash$, then we call R a resolution refutation of Δ.

Example: Let Δ be the clause-set schema defining the sequence of clauses:

$$\mathcal{C}_\alpha = \{ \vdash P(x_0), \ldots, P(g^\alpha(x_\alpha));\ P(a) \vdash\ ; \ldots ; P(g^\alpha(a)) \vdash\ \}$$

Let $(\rho, \mathcal{R})$ be a proof schema with clause variable X defined by the following rewrite system $\mathcal{R}$:

$$\begin{array}{rcl} \rho(0, x, X) & \rightarrow & r(\ \vdash P(\sigma(\bar{0}, x, \bar{0})) \circ X; P(\sigma'(\bar{0})) \vdash; P(\sigma(\bar{0}, x, \bar{0}))) \\ \rho(k+1, x, X) & \rightarrow & \end{array}$$

$$r(\rho(k, x, \vdash P(\sigma(k{+}1, x, k{+}1)) \circ X); P(\sigma'(k{+}1)) \vdash; P(\sigma(k{+}1, x, k{+}1)))$$

Then $(\rho, \mathcal{R})$ is a refutation schema for Δ; indeed, for $\lambda = \{X \leftarrow \vdash\}$, $\mu = \{\xi \leftarrow [c(\bar{0}, x, X)]\}$ and $\theta = \{x \leftarrow \lambda k.a\}$, we get for all $\vartheta_\alpha = \{n \leftarrow \alpha\}$ that:

$$(\rho(n, x, X)\lambda\theta\vartheta_\alpha) \downarrow$$

is a resolution refutation of $v^*_{\text{cst}}(\vartheta_\alpha, \lambda, \mu, d_1(n, x, X, \xi))$, which is just $\mathcal{C}_\alpha$.

2.5 The method *CERES*

Before introducing the method *CERES*$_s$ we conclude the chapter with a brief presentation and example of the first-order method *CERES*. Cut-elimination is a transformation which was first introduced by Gentzen [Gen35] in 1935 in his famous 'Hauptsatz'. Its main goal is the elimination of the cut-rule in an **LK** proof. Removing the cut rules, i.e. the cut formulas (which can be thought of as lemmas), results in a proof which is analytic in the sense that all statements in the proof are analytic due to the subformula property, i.e. all formulas except the cut-ancestors are presented as formulas or subformulas in the end-sequent of the proof. The process of cut-elimination by Gentzen is constructive. It consists of the following steps:

- selection of an uppermost cut
- reduction of the rank and the grade
- decomposition of the cut formulas

The Gentzen's reductive method of cut elimination has two main disadvantages, which makes it hard to be used in the practice : it is very slow and weak in detecting redundancy. The method *CERES* [BL00] is superior to the Gentzen's method. It extracts and analyzes a proof structure which is called characteristic clause set which contains objects from the axioms only and is always unsatisfiable. The resolution refutation of the clause set serves as a skeleton for the analytic variant of the proof. It is obtained by mapping each element of the clause set to a proof called proof projection [BL00]. It is an object obtained from the original proof by skipping all inferences which operate on cut-ancestors. Finally, inserting the ground projections into the resolution refutation , we get a proof with atomic cuts which we call *ACNF*

- atomic cut normal form of the original proof. Here is an example:

Example: Let φ is the **LK** proof with a non-atomic cut formula $B \vee C$:

$$\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{\neg A, A \vdash}\neg:l \qquad B \vdash B}{A, \neg A \vee B \vdash B}\vee:l}{A, \neg A \vee B \vdash B \vee C}\vee:r1 \qquad \dfrac{B \vdash B \qquad C \vdash C}{B \vee C \vdash B, C}\vee:l}{A, \neg A \vee B \vdash B, C}\,cut$$

Let ν be an inference rule in φ. The characteristic clause set is extracted inductively according to the following rules.

- if ν is an axiom S, then $CL(\nu) = \{S'\}$, where $S' \subseteq S$ contains cut-ancestors only
- if ν is a unary rule, then $CL(\nu) = CL(\nu')$, where ν' is the upper inference rule.
- if ν is a binary rule with premises ν_1 and ν_2, then:
  - if the auxiliary formulas of ν are cut-ancestors, then: $CL(\nu) = CL(\nu_1) \cup CL(\nu_2)$
  - otherwise: $CL(\nu) = CL(\nu_1) \times CL(\nu_2)$

If ν_0 is the root inference of a proof φ, then $CL(\nu_0)$ is the clause set of φ. In our example $CL(\varphi) = \{\vdash B\ ;\ B \vdash\ ;\ C \vdash\}$, which has a resolution refutation $\mathcal{R}$:

$$\dfrac{\vdash B \qquad B \vdash}{\vdash}\,cut$$

The proof projections are computed in the following way. Again let ν is an inference rule in φ.

- if ν is an axiom S, then:
  - if all formula occurrences in S are cut-ancestors, then $PR_\nu(\varphi) = \emptyset$
  - $PR_\nu(\varphi) = \{S\}$, otherwise

- if ν is a unary rule in φ with *immediate predecessor* ν' and $PR_{\nu'}(\varphi) = \{\psi_1, \ldots, \psi_n\}$, then:
  - if ν' operates on cut-ancestors, then $PR_\nu(\varphi) = PR_{\nu'}(\varphi)$
  - $PR_\nu(\varphi) = \{\nu(\psi_1), \ldots, \nu(\psi_n)\}$, otherwise
- if ν is a binary rule with premises ν_1 and ν_2, then:
  - if the auxiliary formulas of ν are cut-ancestors, then: $PR_\nu(\varphi) = PR_{\nu_1}(\varphi)^{\Gamma_2 \vdash \Delta_2} \cup PR_{\nu_2}(\varphi)^{\Gamma_1 \vdash \Delta_1}$
  - otherwise: $PR_\nu(\varphi) = PR_{\nu_1}(\varphi) \times_\nu PR_{\nu_2}(\varphi)$,

where $\Gamma_i \vdash \Delta_i$, for $i = 1, 2$ is the subsequent of the conclusion of the inference ν_i which contains non-cut-ancestors. If Q is a set of **LK** proofs, then $Q^{\Gamma \vdash \Delta}$ is the set $\{\psi^{\Gamma \vdash \Delta} | \psi \in Q\}$, where $\psi^{\Gamma \vdash \Delta}$ is ψ followed by weakening rules such that the conclusion of the root of the new derivation is $S \circ (\Gamma \vdash \Delta)$, where S is the end-sequent of ψ. The meaning of $\times_\nu$ is the following: if P, Q are **LK** proofs, then $P \times_\nu Q = \{\nu(\psi, \chi) | \psi \in P, \chi \in Q\}$. If ν_0 is the root inference of an **LK** proof φ, then $PR_{\nu_0}(\varphi)$ is the set of projections for φ.

Let for each clause C in the clause set $\varphi[C]$ be the proof projection for C. In our example the projections $\varphi[\ \vdash B]$ and $\varphi[B \vdash\ ]$ are respectively:

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

In this example the refutation did not provide a non-empty substitution and we do not have to ground the projections because they are already grounded. The *ACNF* of φ is:

$$\dfrac{\dfrac{PR(\ \vdash B) \qquad PR(B \vdash\ )}{A, \neg A \vee B \vdash B, C, B \qquad A, \neg A \vee B, B \vdash B, C}\,cut}{A, \neg A \vee B, A, \neg A \vee B \vdash B, C, B, C}\,c^*\colon l, c^*\colon r$$

Chapter 3

The method $CERES_s$

Cut-elimination was first introduced by Gentzen [Gen35] and it turned out to be crucial for the analysis of mathematical proofs. The cut formulas (shortly cuts) correspond to the lemmas in a proof. Removing the cuts from a proof results in a proof without intermediate statements. Hence, due to the subformula property, we obtain a proof which is purely analytic, i.e. all of the axioms occur in the end-sequent as (sub)formulas. Gentzen's method of cut-elimination is not the only way cut-elimination can be performed. Gentzen developed a reductive cut-elimination method which is essentially a proof-transformation. On the other hand, the *CERES* method performs cut-elimination using resolution [BHL$^+$06]. It is well known that cut-elimination has non-elementary complexity [Ore82] and in spite of these results $CERES$ is complexity-wise superior to Gentzen's reductive cut-elimination method in many cases. Due to the complexity of its resolution refutation, and the fact that *CERES* reduces all cuts simultaneously to atomic cuts, it has non-elementary speed-up in its performance compared to the Gentzen method. In the case of proof schemata the *CERES* method is still applicable. Given an instance of the parameter, one can unfold the proof schemata and obtain a usual **LK** proof, then the *CERES* method can be applied. On the other hand, doing this for each instance is very expensive and possible only for a few instances. For example, experiments on the prime proof have shown that even for the instance with only three primes, the transformation with $CERES$ was not applicable in practice due to the failure of the resolution provers such as Prover9 [McC10] to refute the extracted characteristic clause set. Furthermore, without an automated proof assistant the analysis of the clause set by humans is a very hard task and it may take weeks until a mathematician finds a refutation. This and many other obstacles, as we will see later, can be overcome by the method $CERES_s$, because it gives a *schematic* description of all cut-free proofs for a given parameter n. The

method $CERES_s$ is an extension of the method *CERES* [BL06] which is a method for cut-elimination in first-order logic. Given an **LKS** proof the *CERES* method is not applicable because it is not clear how the cuts will appear through the proof-links. The $CERES_s$ method gives a solution to these problems. Now we can discuss the details starting with the main features of *CERES* which are essential for $CERES_s$.

Here we list the main components of *CERES* which will be extended to $CERES_s$:

- *skolemization* of the proof
- extraction of the *characteristic clause set*
- computation of the *proof projections*
- constructing the *atomic cut normal form* ($ACNF$)

The *skolemization* is a transformation which removes the strong quantifiers in a sequent. A proof derivation which has a skolemized end-sequent is called a skolemized proof. The skolemization is essential for the method *CERES* because if a proof is not skolemized we may get unsound proof projection. This is possible, because the proof projections are constructed such that some inferences in the original proof are skipped. In advance we will say that for the $CERES_s$ method we assume that the **LKS**-proof corresponding to the smallest proof symbol w.r.t. the order $<$ (see page 13) is skolemized.

The *characteristic clause set* (*ccs*) encodes in some sense the structure of the proof, namely the essence of the cut formulas used in the proof. It is obtained by analyzing which axioms go into a cut and which go into the end-sequent. The *ccs* is always unsatisfiable [BL00]. Its elements in the case of *CERES* are clauses. In the case of $CERES_s$ the *ccs* for a proof schema is computed via the $ccs's$ for the schematic proofs in this proof schema. Hence, it is a more complex object because its extraction depends on the proof-links in the step cases of the schematic proofs. We do not know in advance the structure of the clause set in the proof-links of a schematic proof, so we use a special *clause symbol* to "encode" the information such as proof name, index and cut-formulas in the end-sequent. This will be explained in details later in this chapter.

A *proof projection* is a cut-free **LK(S)** proof which is computed for each clause $C \in ccs$ by omitting all inferences with auxiliary formulas which go

into a cut. The same idea for defining proof projections is used also in the definition of $CERES_s$. Like in the extraction of the *ccs*, we again introduce a special symbols, called *projection symbols*, which encodes the projections in the proof-links. Furthermore, we introduce a projection term - an object which is more convenient to work with rather then the set. The projection term contains projection symbols and can be unfolded to a ground projection term (without projection symbols) for a given instance of the parameter which represents a set of ground **LK(S)** proofs.

The *ACNF* (atomic cut normal form) of the *CERES* method is obtained by mapping the resolution refutation of the *ccs* to the proof projections and creating a proof with at most atomic cuts out of it. In the $CERES_s$ method the approach is similar : given a schematic clause set we define a resolution proof schemata and verify the unsatisfiability of the clause set. Then the resolution proof is mapped to the proof projections for the same instance of the parameter. Finally, we combine the projections in a resolution proof which looks like an **LK** proof with (at most atomic) cuts and contractions only. The resulting derivation is the *ACNF*.

So far we have shown the basic differences between the methods *CERES* and $CERES_s$. Still one may ask, are there another differences. Apart from the proof-links, there is an important difference in a technical aspect - $CERES_s$ is applicable in the presence of implicit induction whereas the *CERES* and the Gentzen's methods are not. Hence, $CERES_s$ is more powerful than they are, because first an **LKS** proof represents an infinite sequence of **LK** proofs and second, if we consider Peano arithmetic (PA) and add to the calculus the induction axiom:

$$\frac{\Gamma, A(\alpha) \vdash \Delta, A(s(\alpha))}{A(\bar{0}), \Gamma \vdash \Delta, A(t)}\, ind$$

where α is an eigenvariable not occurring in $A(\bar{0}), \Gamma, \Delta$, we see that $CERES_s$ is superior over reductive cut-elimination. This can be seen in the following example. Consider the sequent S:

$$Def(\hat{f}), \forall x(P(x) \to P(f(x))) \vdash (\forall n)(\forall x)((P(\hat{f}(n,x)) \to P(g(n,x))) \to (P(x) \to P(g(n,x))))$$

where $g \in F^2$, $f \in F^1$, $\hat{f} \in F^2_s$ such that $Def(\hat{f})$ is:

$$\begin{aligned} (\forall x)\hat{f}(\bar{0}, x) &= x \\ (\forall n)(\forall x)\hat{f}(s(n), x) &= f(\hat{f}(n, x)) \end{aligned}$$

Obviously, S can not be proven without induction. The reason is that if we delete the $\forall\colon r$ inference in ψ, then we can not eliminate the formula $\forall x(P(x) \to P(\hat{f}(\beta, x)))$ because we can not cross the induction rule. In fact we need an inductive lemma, for instance:

$$Def(\hat{f}), \forall x(P(x) \to P(f(x))) \vdash (\forall n)(\forall x)(P(x) \to P(\hat{f}(n, x)))$$

A proof ψ of this inductive lemma could be:

$$\dfrac{\dfrac{(\forall x)\hat{f}(\bar{0}, x) = x \vdash \forall x(P(x) \to P(\hat{f}(\bar{0}, x)))\ (\psi_1) \qquad \dfrac{\dfrac{\Gamma, \forall x(P(x) \to P(\hat{f}(\alpha, x))) \vdash \forall x(P(x) \to P(\hat{f}(s(\alpha), x)))\ (\psi_2)}{\Gamma, \forall x(P(x) \to P(\hat{f}(\bar{0}, x))) \vdash \forall x(P(x) \to P(\hat{f}(\gamma, x)))}\, ind}{\Gamma, \forall x(P(x) \to P(\hat{f}(\bar{0}, x))) \vdash (\forall n)(\forall x)(P(x) \to P(\hat{f}(n, x)))}\, \forall\colon r}{Def(\hat{f}), \forall x(P(x) \to P(f(x))) \vdash (\forall n)(\forall x)(P(x) \to P(\hat{f}(n, x)))}\, cut$$

where Γ is the context $(\forall n)(\forall x)\hat{f}(s(n), x) = f(\hat{f}(n, x)), (\forall x)(P(x) \to P(f(x)))$

The proofs ψ_1 and ψ_2 are easily defined, but some simple equational reasoning is required. Thus, to avoid the use of equality rules, we admit an atomic equality axiom on the leaves. The axiom has the following form: $P(s), t = s \vdash P(t)$ and $\vdash s = s$. Then ψ_1 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{P(u), \hat{f}(\bar{0}, u) = u \vdash P(\hat{f}(\bar{0}, u))}{\hat{f}(\bar{0}, u) = u \vdash P(u) \to P(\hat{f}(\bar{0}, u))}\, \to\colon r}{(\forall x)\hat{f}(\bar{0}, x) = x \vdash P(u) \to P(\hat{f}(\bar{0}, u))}\, \forall\colon l}{(\forall x)\hat{f}(\bar{0}, x) = x \vdash \forall x(P(x) \to P(\hat{f}(\bar{0}, x)))}\, \forall\colon r$$

and ψ_2 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(\hat{f}(\alpha, u)) \vdash P(\hat{f}(\alpha, u))}{P(\hat{f}(\alpha, u)) \to P(f(\hat{f}(\alpha, u))), \hat{f}(s(\alpha), u) = f(\hat{f}(\alpha, u)), P(\hat{f}(\alpha, u)) \vdash P(\hat{f}(s(\alpha), u))}\, \to\colon l}{P(f(\hat{f}(\alpha, u))), \hat{f}(s(\alpha), u) = f(\hat{f}(\alpha, u)) \vdash P(\hat{f}(s(\alpha), u))}\, \forall\colon l*}{\Gamma, P(\hat{f}(\alpha, u)) \vdash P(\hat{f}(s(\alpha), u))}\, \to\colon l}{P(u), \Gamma, P(u) \to P(\hat{f}(\alpha, u)) \vdash P(\hat{f}(s(\alpha), u))}\, \to\colon r}{\Gamma, P(u) \to P(\hat{f}(\alpha, u)) \vdash P(u) \to P(\hat{f}(s(\alpha), u))}\, \forall\colon l}{\Gamma, \forall x(P(x) \to P(\hat{f}(\alpha, x))) \vdash P(u) \to P(\hat{f}(s(\alpha), u)))}\, \forall\colon r}{\Gamma, \forall x(P(x) \to P(\hat{f}(\alpha, x))) \vdash \forall x(P(x) \to P(\hat{f}(s(\alpha), x)))}\, \forall\colon r}{P(u) \vdash P(u)}\, \to\colon l$$

Finally, we define φ as (to save some space, the cut-formula: $(\forall n)(\forall x)(P(x) \to P(\hat{f}(n, x)))$ is denoted with C):

$$\dfrac{\begin{array}{cc}(\psi) & (\chi)\\ Def(\hat{f}),\forall x(P(x)\to P(f(x)))\vdash C & C\vdash(\forall n)(\forall x)((P(\hat{f}(n,x))\to P(g(n,x)))\to(P(x)\to P(g(n,x))))\end{array}}{Def(\hat{f}),\forall x(P(x)\to P(f(x)))\vdash(\forall n)(\forall x)((P(\hat{f}(n,x))\to P(g(n,x)))\to(P(x)\to P(g(n,x))))}\,cut$$

where χ is an induction-free proof of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(\hat{f}(\beta,u))\vdash P(\hat{f}(\beta,u))}\quad\dfrac{}{P(g(\beta,u))\vdash P(g(\beta,u))}}{P(\hat{f}(\beta,u))\to P(g(\beta,u)),P(\hat{f}(\beta,u))\vdash P(g(\beta,u))}\to:l}{P(u)\vdash P(u)\qquad P(\hat{f}(\beta,u))\to P(g(\beta,u)),P(\hat{f}(\beta,u))\vdash P(g(\beta,u))}\to:l}{P(u),P(\hat{f}(\beta,u))\to P(g(\beta,u)),P(u)\to P(\hat{f}(\beta,u))\vdash P(g(\beta,u))}\to:r}{P(\hat{f}(\beta,u))\to P(g(\beta,u)),P(u)\to P(\hat{f}(\beta,u))\vdash P(u)\to P(g(\beta,u))}\to:r}{P(u)\to P(\hat{f}(\beta,u))\vdash(P(\hat{f}(\beta,u))\to P(g(\beta,u)))\to(P(u)\to P(g(\beta,u)))}\forall:l*}{(\forall n)(\forall x)(P(x)\to P(\hat{f}(n,x)))\vdash(P(\hat{f}(\beta,u))\to P(g(\beta,u)))\to(P(u)\to P(g(\beta,u)))}\forall:r*}{(\forall n)(\forall x)(P(x)\to P(\hat{f}(n,x)))\vdash(\forall n)(\forall x)((P(\hat{f}(n,x))\to P(g(n,x)))\to(P(x)\to P(g(n,x))))}$$

Performing Gentzen's reductive cut-elimination, we locate the place in the proof, where $(\forall n)$ is introduced. In χ, $(\forall n)(\forall x)(P(x)\to P(\hat{f}(\bar{n},x)))$ is obtained via $\forall x(P(x)\to P(\hat{f}(\beta,x)))$ by $\forall\colon l$. In the proof ψ we may delete the $\forall\colon r$ inference yielding the cut-formula and replace γ by β. But in the attempt to eliminate $\forall x(P(x)\to P(\hat{f}(\beta,x)))$ in ψ we get stuck, as we cannot "cross" the *ind* rule. The *ind* rule can not be eliminated, because of the variable β. In fact, if we instead had $\forall x(P(x)\to P(\hat{f}(t,x)))$ for a closed term t over $\{\bar{0},s,+,*\}$ we could prove $\mathrm{PA}\vdash t=\bar{n}$ and also:

$$Def(\hat{f}),\forall x(P(x)\to P(f(x))),\forall x(P(x)\to P(\hat{f}(\bar{0},x)))\vdash\forall x(P(x)\to P(\hat{f}(\bar{n},x)))$$

without induction (by iterated cuts) and cut-elimination would proceed. This problem, however, is neither rooted in the specific form of ψ, nor in the *ind* rule. Even if we had used binary induction rule:

$$\dfrac{\Gamma\vdash A(\bar{0})\qquad\Delta,A(\alpha)\vdash A(s(\alpha))}{\Gamma,\Delta\vdash A(t)}\,ind$$

the result would be the same. In fact, there exists no proof of S with only atomic cuts – even if *ind* is used. In particular, induction on the formula $(\forall n)(\forall x)((P(\hat{f}(n,x))\to P(g(n,x)))\to(P(x)\to P(g(n,x))))$ fails. In order to prove the end-sequent an inductive lemma is needed (something which implies $(\forall n)(\forall x)(P(x)\to P(\hat{f}(n,x)))$) and can not be eliminated.

While there are no proof of S in PA with only atomic cuts, the sequents S_n:

$$Def(\hat{f}),\forall x(P(x)\to P(f(x)))\vdash\forall x((P(\hat{f}(n,x))\to P(g(n,x)))\to(P(x)\to P(g(n,x))))$$

do have such proofs for all n. They can be proved even without induction. Instead of a unique proof φ of S we get an infinite sequence of proofs φ_n

of S_n, which have cut-free versions φ'_n (henceforth "cut-free" means that atomic cuts are admitted). This kind of "infinitary" cut-elimination only makes sense if there exists a uniform representation of the sequence of proofs φ'_n. We will illustrate below that the method *CERES* has the potential of producing such a uniform representation, thus paving the way for cut-elimination in the presence of induction.

Let S_n be the sequent:

$$\forall x(P(x) \to P(f(x))) \vdash \forall x((P(\hat{f}(\bar{n}, x)) \to P(g(\bar{n}, x))) \to (P(x) \to P(g(\bar{n}, x))))$$

where n is a number variable, henceforth called a parameter. For the symbol $\hat{f}$ we add the reduction rules:

$$\begin{array}{rcl} \hat{f}(\bar{0}, x) & \to & x \\ \hat{f}(s(k), x) & \to & f(\hat{f}(k, x)) \end{array}$$

First we define a proof schema ψ_n playing the role of ψ in the inductive proof above. Let ψ_0 be:

$$\dfrac{\dfrac{\dfrac{\dfrac{P(\hat{f}(\bar{0}, x_0)) \vdash P(\hat{f}(\bar{0}, x_0))}{P(x_0) \vdash P(\hat{f}(\bar{0}, x_0))} \twoheadrightarrow}{\vdash P(x_0) \to P(\hat{f}(\bar{0}, x_0))} \to\colon r}{\vdash (\forall x)(P(x) \to P(\hat{f}(\bar{0}, x)))} \forall\colon r}{\forall x(P(x) \to P(f(x))) \vdash (\forall x)(P(x) \to P(\hat{f}(\bar{0}, x)))} w\colon l$$

and ψ_{k+1} be:

$$\dfrac{\dfrac{\overset{(\psi_k)}{\overbrace{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}}}{\forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(\hat{f}(k, x)))} \quad (1)}{\forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(\hat{f}(s(k), x)))} \; cut, c\colon l$$

where (1) is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(x_{k+1}) \vdash P(x_{k+1})}{P(x_{k+1}), P(x_{k+1}) \to P(\hat{f}(k, x_{k+1})), \forall x(P(x) \to P(f(x))) \to P(\hat{f}(s(k), x_{k+1}))} \to\colon r}{P(x_{k+1}) \to P(\hat{f}(k, x_{k+1})), \forall x(P(x) \to P(f(x))) \vdash P(x_{k+1}) \to P(\hat{f}(s(k), x_{k+1}))} \forall\colon l}{\forall x(P(x) \to P(\hat{f}(k, x))), \forall x(P(x) \to P(f(x))) \vdash P(x_{k+1}) \to P(\hat{f}(s(k), x_{k+1}))} \forall\colon r}{\forall x(P(x) \to P(\hat{f}(k, x))), \forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(\hat{f}(s(k), x)))}$$

$$\dfrac{\dfrac{P(\hat{f}(k, x_{k+1})) \vdash P(\hat{f}(k, x_{k+1}))}{P(\hat{f}(k, x_{k+1})), P(\hat{f}(k, x_{k+1})) \to P(f(\hat{f}(k, x_{k+1}))) \vdash P(\hat{f}(s(k), x_{k+1}))} \to\colon l}{P(\hat{f}(\bar{k}, x_{k+1})), \forall x(P(x) \to P(f(x))) \vdash P(\hat{f}(s(k), x_{k+1}))} \forall\colon l$$

$$\dfrac{P(\hat{f}(s(k), x_{k+1})) \vdash P(\hat{f}(s(k), x_{k+1}))}{P(f(\hat{f}(k, x_{k+1}))) \vdash P(\hat{f}(s(k), x_{k+1}))} \twoheadrightarrow$$

Instead of S_n we construct the skolemized version S'_n:[1]

$$\forall x(P(x) \to P(f(x))) \vdash (P(\hat{f}(n,c)) \to P(g(n,c))) \to (P(c) \to P(g(n,c)))$$

and define φ_n:

$$\dfrac{\dfrac{}{\forall x(P(x) \to P(f(x))) \vdash C}\,(\psi_n) \qquad \dfrac{}{C \vdash (P(\hat{f}(n,c)) \to P(g(n,c))) \to (P(c) \to P(g(n,c)))}\,(\chi_n)}{\forall x(P(x) \to P(f(x))) \vdash (P(\hat{f}(n,c)) \to P(g(n,c))) \to (P(c) \to P(g(n,c)))}\,cut$$

where $C = \forall x(P(x) \to P(\hat{f}(n,x)))$ and (χ_n) is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(\hat{f}(n,c)) \vdash P(\hat{f}(n,c))} \quad \dfrac{}{P(g(n,c)) \vdash P(g(n,c))}}{P(\hat{f}(n,c)) \to P(g(n,c)), P(\hat{f}(n,c)) \vdash P(g(n,c))}\,{\to}:l \quad \dfrac{}{P(c) \vdash P(c)}}{P(c), P(\hat{f}(n,c)) \to P(g(n,c)), P(c) \to P(\hat{f}(n,c)) \vdash P(g(n,c))}\,{\to}:l}{P(\hat{f}(n,c)) \to P(g(n,c)), P(c) \to P(\hat{f}(n,c)) \vdash P(c) \to P(g(n,c))}\,{\to}:r}{P(c) \to P(\hat{f}(n,c)) \vdash (P(\hat{f}(\bar{n},c)) \to P(g(n,c))) \to (P(c) \to P(g(n,c)))}\,{\to}:r}{\forall x(P(x) \to P(\hat{f}(n,x))) \vdash (P(\hat{f}(n,c)) \to P(g(n,c))) \to (P(c) \to P(g(n,c)))}\,\forall:l$$

As a next step we define characteristic clause set schema of φ_n, $\mathrm{CL}(\varphi_n)$, inductively: $\mathrm{CL}(\varphi_0)$ is $\{ \vdash P(c);\ P(\hat{f}(\bar{0},c)) \vdash \}$ and $\mathrm{CL}(\varphi_{n+1})$ is:

$$\{P(\hat{f}(\bar{0},x_1)) \vdash P(\hat{f}(s(\bar{0}),x_1));\ \cdots;\ P(\hat{f}(\bar{n},x_{n+1})) \vdash P(\hat{f}(s(\bar{n}),x_{n+1}));$$
$$\vdash P(c)\ ;\ P(\hat{f}(\bar{n}+1,c)) \vdash \}$$

Now, if we apply the reduction rules for $\hat{f}$ to the clause $P(\hat{f}(\bar{0},x_1)) \vdash P(\hat{f}(s(\bar{0}),x_1))$, we get $P(x_1) \vdash P(f(x_1))$ and the clause set boils down – via term-rewriting followed by subsumption to:

$$\mathrm{CL}(\varphi_n)' = \{P(x_1) \vdash P(f(x_1));\ \vdash P(c);\ P(\hat{f}(\bar{n},c)) \vdash \}$$

A sequence of resolution refutations of $\mathrm{CL}(\varphi_n)'$ is given by δ_n:

$$\dfrac{\dfrac{}{\vdash P(\hat{f}(n,c))}\,(\eta_n) \qquad \dfrac{}{P(\hat{f}(n,c)) \vdash}}{\vdash}$$

where η_0 is $\vdash P(c)$ and η_{k+1} is:

[1] Skolemization is vital for *CERES* but the situation in the inductive proof above remains the same – we get c instead of β and again the same argument applies that the cut cannot be eliminated.

$$\dfrac{\begin{array}{c}(\eta_k)\\ \vdash P(\hat{f}(k,c)) \qquad P(x_{k+1}) \vdash P(f(x_{k+1}))\end{array}}{\vdash P(\hat{f}(k+1,c))}\, x_{k+1} \leftarrow \hat{f}(k,c)$$

Here $P(\hat{f}(k+1,c))$ rewrites to $P(f(\hat{f}(k,c)))$, where $\vdash P(f(\hat{f}(k,c)))$ is the end-sequent (clause) of η_{k+1}.

Now we can start with the formalization of the method $CERES_s$ for first-order schemata. Our goal is to give a schematic description of all cut-free proofs for a parameter k. We stress again that by Gentzen's method this is impossible because of the presence of proof-links, i.e. moving the cuts through the proof-links is not clear. With $CERES_s$ we overcome this problem - we analyze globally the proof schema and provide the desired description of sequences of cut-free proofs. First, let us consider the following example which will be the running one until the end of this chapter: a schema $\Psi = \langle(\pi_1, \nu_1(k+1)), (\pi_2, \nu_2(k+1))\rangle$, where σ and φ are schematic proof symbols corresponding respectively to the pairs of **LKS**-proofs $(\pi_1, \nu_1(k+1))$ and $(\pi_2, \nu_2(k+1))$, such that $\sigma < \varphi$. We fix a term-rewriting system, where $g \in F^1_a$, $f \in F^1$, $k \in V_a$ and $x \in V_1$:

$$\begin{array}{rcl} g(\bar{0}, x) & \rightarrow & x \\ g(k+1, x) & \rightarrow & f(g(k, x)) \end{array}$$

The two base-case proofs π_1 and π_2, and the two step-case proofs $\nu_1(k+1)$ and $\nu_2(k+1)$ in Ψ are defined as follows:

The proof π_1 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(a) \vdash P(a) \qquad P(g(\bar{0},a)) \vdash P(g(\bar{0},a))}{P(a), P(a) \rightarrow P(g(\bar{0},a)) \vdash P(g(\bar{0},a))}\rightarrow\!: l \qquad P(h(\bar{0},a)) \vdash P(h(\bar{0},a))}{P(a), P(a) \rightarrow P(g(\bar{0},a)), P(g(\bar{0},a)) \rightarrow P(h(\bar{0},a)) \vdash P(h(\bar{0},a))}\rightarrow\!: l}{P(a) \rightarrow P(g(\bar{0},a)), P(g(\bar{0},a)) \rightarrow P(h(\bar{0},a)) \vdash P(a) \rightarrow P(h(\bar{0},a))}\rightarrow\!: r}{P(g(\bar{0},a)) \rightarrow P(h(\bar{0},a)), \forall x(P(x) \rightarrow P(g(\bar{0},x))) \vdash P(a) \rightarrow P(h(\bar{0},a))}\forall\!: l}{\forall x(P(x) \rightarrow P(g(\bar{0},x))), \forall x(P(g(\bar{0},x)) \rightarrow P(h(\bar{0},x))) \vdash P(a) \rightarrow P(h(\bar{0},a))}\forall\!: l$$

The proof χ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(a) \vdash P(a) \qquad \dfrac{P(g(k+1,a)) \vdash P(g(k+1,a)) \qquad P(h(k+1,a)) \vdash P(h(k+1,a))}{P(g(k+1,a)), P(g(k+1,a)) \rightarrow P(h(k+1,a)) \vdash P(h(k+1,a))}\rightarrow\!: l}{P(a), (P(g(k+1,a)) \rightarrow P(h(k+1,a))), P(a) \rightarrow P(g(k+1,a)) \vdash P(h(k+1,a))}\rightarrow\!: l}{P(g(k+1,a)) \rightarrow P(h(k+1,a)), P(a) \rightarrow P(g(k+1,a)) \vdash P(a) \rightarrow P(h(k+1,a))}\rightarrow\!: r}{P(a) \rightarrow P(g(k+1,a)), \forall x(P(g(k+1,x)) \rightarrow P(h(k+1,x))) \vdash P(a) \rightarrow P(h(k+1,a))}\forall\!: l}{\forall x(P(x) \rightarrow P(g(k+1,x))), \forall x(P(g(k+1,x)) \rightarrow P(h(k+1,x))) \vdash P(a) \rightarrow P(h(k+1,a))}\forall\!: l$$

The proof $\nu_1(k+1)$ is:

$$\dfrac{\dfrac{}{\forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k,x))))}(\nu_2(k)) \qquad \dfrac{}{\forall x(P(x)\to P(f(g(k,x)))), \forall x(P(g(k+1,x))\to P(h(k+1,x)))\vdash P(a)\to P(h(k+1,a))}\chi}{\forall x(P(x)\to P(f(x))), \forall x(P(g(k+1,x))\to P(h(k+1,x)))\vdash P(a)\to P(h(k+1,a))}\twoheadrightarrow\ cut$$

The proof π_2 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(a)\vdash P(a)} \qquad \dfrac{}{P(f(g(\bar{0},a)))\vdash P(f(g(\bar{0},a)))}}{P(f(a))\vdash P(f(g(\bar{0},a)))}\twoheadrightarrow}{P(a), P(a)\to P(f(a))\vdash P(f(g(\bar{0},a)))}\to: l}{P(a)\to P(f(a))\vdash P(a)\to P(f(g(\bar{0},a)))}\to: r}{\forall x(P(x)\to P(f(x)))\vdash P(a)\to P(f(g(\bar{0},a)))}\forall: l}{\forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(\bar{0},x))))}\forall: r$$

The proof τ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(z_{k+1})\vdash P(z_{k+1})} \qquad \dfrac{\dfrac{}{P(g(k+1,z_{k+1}))\vdash P(g(k+1,z_{k+1}))}}{P(f(g(k,z_{k+1})))\vdash P(g(k+1,z_{k+1}))}\twoheadrightarrow \qquad \dfrac{}{P(f(g(k+1,z_{k+1})))\vdash P(f(g(k+1,z_{k+1})))}}{P(f(g(k,z_{k+1}))), P(g(k+1,z_{k+1}))\to P(f(g(k+1,z_{k+1})))\vdash P(f(g(k+1,z_{k+1})))}\to: l}{P(z_{k+1}), P(g(k+1,z_{k+1}))\to P(f(g(k+1,z_{k+1}))), P(z_{k+1})\to P(f(g(k,z_{k+1})))\vdash P(f(g(k+1,z_{k+1})))}\to: l}{P(g(k+1,z_{k+1}))\to P(f(g(k+1,z_{k+1}))), P(z_{k+1})\to P(f(g(k,z_{k+1})))\vdash P(z_{k+1})\to P(f(g(k+1,z_{k+1})))}\to: r}{P(z_{k+1})\to P(f(g(k,z_{k+1}))), \forall x(P(x)\to P(f(x)))\vdash P(z_{k+1})\to P(f(g(k+1,z_{k+1})))}\forall: l}{\forall x(P(x)\to P(f(g(k,x)))), \forall x(P(x)\to P(f(x)))\vdash P(z_{k+1})\to P(f(g(k+1,z_{k+1})))}\forall: l$$

The proof $\nu_2(k+1)$ is:

$$\dfrac{\dfrac{\dfrac{}{\forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k,x))))}(\nu_2(k)) \qquad \dfrac{}{\forall x(P(x)\to P(f(g(k,x)))), \forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k+1,x))))}\tau}{\forall x(P(x)\to P(f(x))), \forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k+1,x))))}\forall: r\ cut}{\forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k+1,x))))}c: l$$

Remark: *Note that the proofs χ and τ are not elements of a pair in the proof schemata Ψ , although they are* **LKS** *proofs. Neither of them have proof links, nor they are ground. For the sake of readability only, the step cases of σ and φ were divided into two parts. This does not affect the extraction of the ccs and the projections. In fact for the method CERES$_s$ the proofs χ and τ are "invisible".*

3.1 Schematic characteristic clause-set

At the heart of the *CERES* method lies the *characteristic clause set*, which describes the cuts in an **LK** proof. It consists of the subsequents of the axioms whose formula occurrences go into a cut. The connection between the cut-elimination and the characteristic clause set is that any resolution refutation of the characteristic clause set can be used as a skeleton of a proof containing only atomic cuts.

The characteristic clause set can either be defined directly [BL00] or it can be obtained via a transformation from a *characteristic clause term* [BL06]. We will use the second approach here. The reason is that the clause term is closed under the rewriting rules (defined below) that we have given for the clause term symbols, while the notion of the clause set is not closed. Therefore, it is much easier to work with a single object (the schematic clause term). In order to obtain the schematic clause set from the schematic clause term we apply a transformation called *clausification* which is described later.

Our main aim is to extend the usual inductive definition of the characteristic clause term to the case of proof links. This will give rise to a notion of *schematic characteristic clause-set term*. As usual, a *clause-set term* is a term built inductively from clauses and the binary symbols $\otimes, \oplus$. The usual definition of the characteristic clause term depends on the occurrences of the cuts in a proof (i.e. whether a given formula occurrence is a cut-ancestor or not). In the method $CERES_s$ the extraction of the clause set changes due to the presence of proof-links. Therefore, cut-ancestor in the proof-link can generate new cut-ancestors when the proof schema is unfolded for a concrete instance of the parameter. That means that in a schematic proof some of the formula occurrences in the end-sequent are potential candidates for cut-ancestors. Therefore, we need some machinery to track the cut-status of the formula occurrences through the proof-links. We call a set Ω of formula occurrences from the end-sequent of an **LKS**-proof π a *configuration* for π.

We will represent the characteristic clause term of a proof link in our object language: For all proof symbols ψ and configurations Ω we assume a unique indexed proposition symbol $\mathrm{cl}^{\Omega,\psi}$ called *clause term symbol*. The intended semantics of $\mathrm{cl}_a^{\Omega,\psi}$ is "the characteristic clause set of $\psi(a)$, with the configuration Ω".

In the example above all cut-ancestors of the **LKS** proofs $\nu_1(k+1)$ and $\nu_2(k+1)$ are marked with red color. The formula occurrences which after an

instantiation of the parameter and unfolding the proof became cut-ancestors are marked with blue color. More precisely speaking, consider the proof $\nu_2(k+1)$. It has only one cut-inference and the cut-formula (in red) is:

$$\forall x(P(x) \to P(f(g(k,x))))$$

The formula occurrence corresponding to that formula is in the succedent part of the end-sequent of the left part of the inference. The left part of the inference contains only a nullary rule - a proof-link to $\nu_2(k)$ (informally we can think of axioms and proof-links as nullary rules). That means that the formula occurrence $\forall x(P(x) \to P(f(g(k,x))))$ and all its ancestors in the proof $\nu_2(k)$ will became cut-ancestors after the instantiation of the parameter k. Therefore, the red cut-ancestors are propagated upwards in the derivation and possibly new configurations appear. Finally, the propagation of the cut-ancestors goes through the base-case of the schematic proof.

Let ψ be a proof symbol corresponding to the pair of **LKS**-proofs $\langle \pi, \nu(k{+}1) \rangle$. A *(cut-)configuration* Ω for ψ will be denoted as $(A_1, \ldots, A_n \mid B_1, \ldots, B_m)$, where $m, n \in \mathbb{N}$ and A_i, for $i = 1 \ldots n$, are formula occurrences in the antecedent part of the end-sequent of the step-case proof $\nu(k+1)$ and B_i, for $i = 1 \ldots m$, are formula occurrences in the succedent part of the end-sequent of $\nu(k+1)$. To be absolutely precise the configuration has formally to be defined as a coding of the formula occurrences in a sequent which depends on the parameter n. This is possible, because the uniqueness of the ancestor relation. However, we will not do it here, because the correspondence between a formula and a formula occurrence according to the notation above is clear enough for the reader. In our running example we have the following cut configurations:

- for σ:
  - $\Omega_1 = \emptyset$
- for φ:
  - $\Omega_2 = (\quad \mid \forall x(P(x) \to P(f(g(k,x))))$

Note that in the proof $\varphi(k+1)$ the blue formula occurrences, i.e. the propagated cut-ancestors, do not generate new configurations. In general this is not the case as we will see some interesting examples in the last chapter.

Definition (characteristic clause-set term) Let π be an **LKS**-proof and Ω a

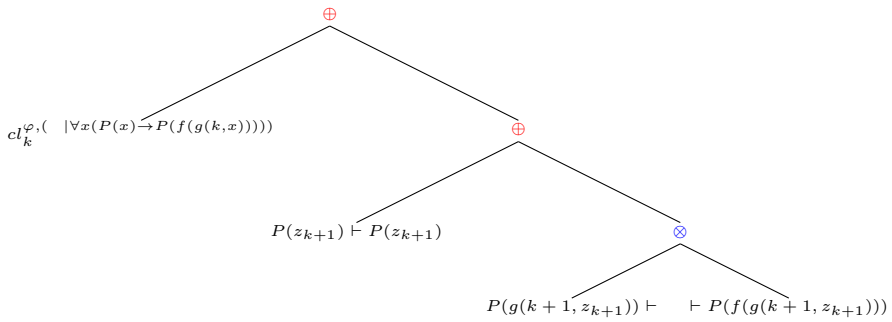
configuration. In the following, by $\Gamma_\Omega, \Delta_\Omega$ and Γ_C, Δ_C we will denote multi-sets of formulas of Ω- and cut-ancestors respectively. Let ρ be an inference in π. We define a clause-set term $\Theta_\rho(\pi, \Omega)$ inductively:

- if ρ is an axiom of the form $\Gamma_\Omega, \Gamma_C, \Gamma \vdash \Delta_\Omega, \Delta_C, \Delta$, then $\Theta_\rho(\pi, \Omega) = [\Gamma_\Omega, \Gamma_C \vdash \Delta_\Omega, \Delta_C]$, where Δ contains no Ω- or cut-ancestors.
- if ρ is a proof-link of the form $\dfrac{(\psi(a, x_1, \ldots, x_\alpha))}{\Gamma_\Omega, \Gamma_C, \Gamma \vdash \Delta_\Omega, \Delta_C, \Delta}$ then define Ω' as the set of formula occurrences from $\Gamma_\Omega, \Gamma_C \vdash \Delta_\Omega, \Delta_C$ and $\Theta_\rho(\pi, \Omega) = \mathrm{cl}^{\Omega', \psi}_{a, x_1, \ldots, x_\alpha}$
- if ρ is an unary rule with immediate predecessor ρ', then $\Theta_\rho(\pi, \Omega) = \Theta_{\rho'}(\pi, \Omega)$.
- if ρ is a binary rule with immediate predecessors ρ_1, ρ_2, then
  - if the auxiliary formulas of ρ are Ω- or cut-ancestors, then $\Theta_\rho(\pi, \Omega) = \Theta_{\rho_1}(\pi, \Omega) \oplus \Theta_{\rho_2}(\pi, \Omega)$,
  - otherwise $\Theta_\rho(\pi, \Omega) = \Theta_{\rho_1}(\pi, \Omega) \otimes \Theta_{\rho_2}(\pi, \Omega)$.

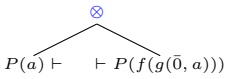
Finally, we define $\Theta(\pi, \Omega) = \Theta_{\rho_0}(\pi, \Omega)$, where ρ_0 is the last inference of π, and $\Theta(\pi) = \Theta(\pi, \emptyset)$.

A characteristic term in our running example (remember $\sigma < \varphi$) has a binary tree representation:

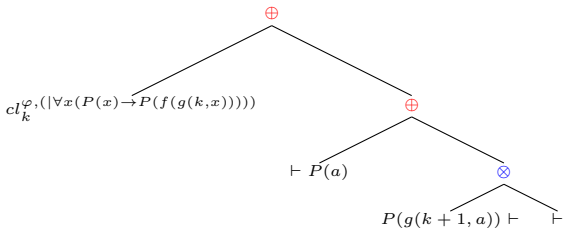
- $\Theta(\nu_1(k+1), (\ |\forall x(P(x) \to P(f(g(k, x))))))$



- $\Theta(\pi_1, (\ |\forall x(P(x) \to P(f(g(k, x))))))$:



- $\Theta(\nu_2(k+1), (\ |\ ))$:



- $\Theta(\pi_2, (\ |\ ))$:

$$\vdash$$

We say that a clause term is *ground* if it does not contain index variables and clause term symbols. Analogously to proof schemata, we define a notion of evaluation of characteristic clause terms:

Definition (evaluation) We define the rewrite rules for clause term symbols for all proof symbols ψ_β and configurations Ω:

$$\begin{aligned} \mathrm{cl}^{\Omega,\psi_\beta}_{0,x_1,\ldots,x_\alpha} &\to \Theta(\pi_\beta, \Omega) \\ \mathrm{cl}^{\Omega,\psi_\beta}_{k+1,x_1,\ldots,x_\alpha} &\to \Theta(\nu_\beta(k+1), \Omega) \end{aligned}$$

for all $1 \leq \beta \leq \alpha$. Next, let $\gamma \in \mathbb{N}$ and let $\mathrm{cl}^{\Omega,\psi_\beta}\downarrow_\gamma$ be the normal form of $\mathrm{cl}^{\Omega,\psi_\beta}_{\gamma,x_1,\ldots,x_\alpha}$ under the above rewriting system. Then define $\Theta(\psi_\beta, \Omega) = \mathrm{cl}^{\Omega,\psi_\beta}$ and $\Theta(\Psi, \Omega) = \Theta(\psi_1, \Omega)$. Finally, we define the *schematic characteristic clause term* $\Theta(\Psi) = \Theta(\Psi, \emptyset)$.

Let us compute $\Theta(\Psi)$ for instance $k = 1$:

$$\Theta(\Psi)\downarrow_2 = \Theta(\Psi\downarrow_2, \emptyset) = \Theta(\sigma(2), \emptyset) =$$

$$= \vdash cl_1^{\varphi,(\,|\forall x(P(x)\to P(f(g(\bar{1},x)))))} \oplus (\vdash P(a) \oplus (P(g(\bar{2}, a)) \vdash \otimes \vdash)) =$$

$$= \vdash cl_1^{\varphi,(\,|\forall x(P(x)\to P(f(g(\bar{1},x)))))} \oplus \vdash P(a) \oplus P(f(f(a))) \vdash$$

$$cl_1^{\varphi,(\,|\forall x(P(x)\to P(f(g(\bar{1},x)))))} \to \Theta(\varphi(1), (\,|\forall x(P(x) \to P(f(g(\bar{1},x))))))$$

$$\Theta(\varphi(1), (\,|\forall x(P(x) \to P(f(g(\bar{1},x))))))) =$$

$$= \vdash cl_0^{\varphi,(\,|\forall x(P(x)\to P(f(g(\bar{0},x)))))} \oplus P(z_1) \vdash P(z_1) \oplus (P(g(\bar{0},z_1)) \vdash \otimes \vdash P(f(g(\bar{1},z_1))) \twoheadrightarrow$$

$$\twoheadrightarrow \vdash cl_0^{\varphi,(\,|\forall x(P(x)\to P(f(g(\bar{0},x)))))} \oplus P(z_1) \vdash P(z_1) \oplus P(z_1) \vdash P(f(f(z_1)))$$

$$cl_0^{\varphi,(\,|\forall x(P(x)\to P(f(g(\bar{0},x)))))} \to \Theta(\varphi(0), (\,|\forall x(P(x) \to P(f(g(\bar{0},x))))))) =$$

$$= P(a) \vdash \otimes \vdash P(f(g(\bar{0},a))) \twoheadrightarrow P(a) \vdash \otimes \vdash P(f(a))$$

Remark: *Here* $\twoheadrightarrow$ *means term-rewriting and should not be confused with the* **LKS***-rule* $\twoheadrightarrow$ *defined in chapter 2.*

Hence, $\Theta(\Psi) \downarrow_2 = (P(a) \vdash \otimes \vdash P(f(a))) \oplus P(z_1) \vdash P(z_1) \oplus P(z_1) \vdash P(f(f(z_1))) \oplus \vdash P(a) \oplus P(f(f(a))) \vdash$

The next step is to transform the clause-set term to sets:

Definition (cartesian product of sequents) Let $\Gamma \vdash \Delta$ and $\Pi \vdash \Lambda$ be arbitrary sequents, then we define $\Gamma \vdash \Delta \times \Pi \vdash \Lambda = \Gamma, \Pi \vdash \Delta, \Lambda$. We extend this relation to sets of sequents P, Q in a natural way: $P \times Q = \{S_P \times S_Q \mid S_P \in P, S_Q \in Q\}$.

Definition (characteristic clause sets) Let Θ be a clause term. Then we define a clause set $|\Theta|$ in the following way:

- $|[\Gamma \vdash \Delta]| = \{\Gamma \vdash \Delta\}$
- $|\Theta_1 \otimes \Theta_2| = |\Theta_1| \times |\Theta_2|$
- $|\Theta_1 \oplus \Theta_2| = |\Theta_1| \cup |\Theta_2|$.

For an **LKS**-proof π and configuration Ω, $\mathrm{CL}(\pi, \Omega) = |\Theta(\pi, \Omega)|$. We define the *standard characteristic clause set* $\mathrm{CL}(\pi) = \mathrm{CL}(\pi, \emptyset)$ and the *schematic characteristic clause-set* $\mathrm{CL}(\Psi, \Omega) = |\Theta(\Psi, \Omega)|$ and $\mathrm{CL}(\Psi) = \mathrm{CL}(\psi_1, \emptyset)$.

Going back to our running example we have:

$CL(\Psi \downarrow_2) = |\Theta(\Psi) \downarrow_2| =$

$= (\{P(a) \vdash\} \times \{\vdash P(f(a))\}) \cup \{P(z_1) \vdash P(z_1)\} \cup \{P(z_1) \vdash P(f(f(z_1)))\} \cup$
$\{\vdash P(a)\} \cup \{P(f(f(a))) \vdash\} =$

$= \{P(a) \vdash P(f(a))\} \cup \{P(z_1) \vdash P(z_1)\} \cup \{P(z_1) \vdash P(f(f(z_1)))\} \cup$
$\{\vdash P(a)\} \cup \{P(f(f(a))) \vdash\} =$

$= \{P(a) \vdash P(f(a)) \ ; \ P(z_1) \vdash P(z_1) \ ; \ P(z_1) \vdash P(f(f(z_1))) \ ;$
$\vdash P(a) \ ; \ P(f(f(a))) \vdash \ \}$

In order to build a bridge to the next section, we give the resolution refutation $\mathcal{R}$ of $CL(\Psi \downarrow_2)$ which will be a skeleton for building the proof projections:

$$\dfrac{\dfrac{P(z_1) \vdash P(f(f(z_1))) \qquad \vdash P(a)}{\vdash P(f(f(a)))}\, cut, \{z_1 \leftarrow a\} \qquad P(f(f(a))) \vdash}{\vdash}\, cut$$

We conclude this section with the following results showing that the notion of the characteristic clause term is well defined and the commutativity between the clause term and the clause set:

Lemma. Let $\gamma \in \mathbb{N}$ and Ω be a configuration, then $\Theta(\psi_\beta, \Omega) \downarrow_\gamma$ is a ground clause term for all $1 \leq \beta \leq \alpha$. Hence $\Theta(\Psi) \downarrow_\gamma$ is a ground clause term.

Proof: by induction in γ. Follows from the soundness of the **LKS** calculus [Ruk12].

Lemma. Let Ω be a configuration and $\gamma \in \mathbb{N}$. Then $\Theta(\Psi \downarrow_\gamma, \Omega) = \Theta(\Psi, \Omega) \downarrow_\gamma$.

Proof: by double induction on γ and on the number α of proof-symbols

in Ψ [Ruk12].

Lemma. Let π be a ground **LKS**-proof. Then $\mathrm{CL}(\pi)$ is unsatisfiable.

Proof: By the identification of ground **LKS**-proofs with propositional **LK**-proofs [BL00].

3.2 Schematic proof projections

The next step in the schematization of the *CERES* method consists of the definition of schematic proof projections. The aim is, in analogy with the preceding section, to construct a *schematic projection term* that can be evaluated to a set of ground **LKS**-proofs. As before, we introduce formal symbols representing sets of proofs, and again the notion of **LKS**-proof is not closed under the rewrite rules for these symbols, which is the reason for introducing the notion of projection term.

For our term notation we assume for every **LKS**-rule ρ a corresponding *rule symbol* that, by abuse of notation, we also denote by ρ. Given a unary rule ρ and an **LKS**-proof π, there are different ways to apply ρ to the end-sequent of π, because the choice of auxiliary formulas is free. Formally, the projection terms we construct will include this information so that evaluation is always well-defined, but we will suppress it in the notation since the choice of the auxiliary formulas will always be clear from the context.

For every proof symbol ψ and a configuration Ω, we assume a unique proof symbol $\mathrm{pr}^{\Omega,\psi}$. A *projection term* is a term built inductively from sequents and terms $\mathrm{pr}^{\Omega,\psi}(a)$, for some arithmetic expression a, using unary rule symbols, unary symbols $w^{\Gamma\vdash\Delta}$ for all sequents $\Gamma \vdash \Delta$ and binary symbols $\oplus, \otimes_\sigma$ for all binary rules σ. The symbols $\mathrm{pr}^{\Omega,\psi}$ are called *projection symbols*. The intended interpretation of $\mathrm{pr}^{\Omega,\psi}(a)$ is "the set of characteristic projections of $\psi(a)$, with the configuration Ω".

Definition (characteristic projection term) Let π be an **LKS**-proof and Ω an arbitrary configuration for π. Let $\Gamma_\Omega, \Delta_\Omega$ and Γ_C, Δ_C be multisets of formulas corresponding to Ω- and cut-ancestors respectively. We define a *projection term* $\Xi_\rho(\pi, \Omega)$ inductively:

- If ρ corresponds to an initial sequent S, then we define $\Xi_\rho(\pi, \Omega) = S$.

- If ρ is a proof-link in π of the form: $\dfrac{(\psi(a, x_1, \ldots, x_\alpha))}{\Gamma_\Omega, \Gamma_C, \Gamma \vdash \Delta_\Omega, \Delta_C, \Delta}$ then, letting Ω' be the set of formula occurrences from $\Gamma_\Omega, \Gamma_C \vdash \Delta_\Omega, \Delta_C$, define $\Xi_\rho(\pi, \Omega) = \mathrm{pr}^{\psi,\Omega'}(a, x_1, \ldots, x_\alpha)$

- If ρ is a unary inference with immediate predecessor ρ', then:

  - if ρ is a rewrite rule $\twoheadrightarrow$ or the auxiliary formula(s) of ρ are Ω- or cut-ancestors, then $\Xi_\rho(\pi, \Omega) = \Xi_{\rho'}(\pi, \Omega)$

  - $\Xi_\rho(\pi, \Omega) = \rho(\Xi_{\rho'}(\pi, \Omega))$, otherwise

- If σ is a binary inference with immediate predecessors ρ_1 and ρ_2, then:

  - if the auxiliary formulas of σ are Ω- or cut-ancestors, let $\Gamma_i \vdash \Delta_i$ be the ancestors of the end-sequent in the conclusion of ρ_i, for $i = 1, 2$, and define: $\Xi_\sigma(\pi, \Omega) = w^{\Gamma_2 \vdash \Delta_2}(\Xi_{\rho_1}(\pi, \Omega)) \oplus w^{\Gamma_1 \vdash \Delta_1}(\Xi_{\rho_2}(\pi, \Omega))$

  - $\Xi_\sigma(\pi, \Omega) = \Xi_{\rho_1}(\pi, \Omega) \otimes_\sigma \Xi_{\rho_2}(\pi, \Omega)$, otherwise

Define $\Xi(\pi, \Omega) = \Xi_{\rho_0}(\pi, \Omega)$, where ρ_0 is the root inference of π. We say that a projection term is *ground* if it does not contain index variables and projection symbols.

Example: Consider again our running example above. Like in the creation of the clause term, for the two proof-links and the two configurations Ω_1 and Ω_2, we have one projection symbol : $\mathrm{pr}_k^{\varphi,(|\forall x(P(x)\to P(f(g(\bar{0},x)))))}$ and four projection terms - (two configurations and two proof symbols):

$$\Xi(\nu_2(k+1), \Omega_2) = w^{S_1}(c_l(\mathrm{pr}_k^{\varphi,\Omega_2})) \oplus w^{S_2}(\forall_l(P(z_{k+1}) \vdash P(z_{k+1}) \oplus (\twoheadrightarrow_l (P(g(k+1, z_{k+1})) \vdash P(g(k+1, z_{k+1}))) \otimes_{\to_l} (P(f(g(k+1, z_{k+1}))) \vdash P(f(g(k+1, z_{k+1})))))))$$

$$\Xi(\pi_2, \Omega_2) = \forall_l(w^{S_3}(P(a) \vdash P(a) \oplus w^{S_4}(\twoheadrightarrow_l (P(f(g(\bar{0}, a))) \vdash P(f(g(\bar{0}, a)))))))$$

$$\Xi(\nu_1(k+1), \Omega_1) = w^{S_5}(\mathrm{pr}_k^{\varphi,\Omega_2}) \oplus w^{S_6}(\forall_l(\to_r (w^{S_7}(P(a) \vdash P(a)) \oplus w^{S_8}((P(g(k+1, a)) \vdash P(g(k+1, a))) \otimes_{\to_l} (P(h(k+1, a)) \vdash P(h(k+1, a))))))))$$

$\Xi(\sigma(\bar{0}), \Omega_1)$ is equal to the term representing $\sigma(0)$.

The sequents $S_1, \ldots, S_8$ in the weakening (sub)terms above are:
$S_1 = \forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(f(g(k+1, x))))$
$S_2 = S_6 = \forall x(P(x) \to P(f(x)))$
$S_3 = P(f(a)) \vdash$
$S_4 = \vdash P(a)$
$S_5 = \forall x(P(g(k+1, x)) \to P(h(k+1, x))) \vdash P(a) \to P(h(k+1, x))$
$S_7 = P(a) \vdash$
$S_8 = P(g(k+1, a)) \to P(h(k+1, a)) \vdash P(h(k+1, a))$

We now define the projection-set schema, which describes how the projection symbols are to be replaced in a projection term:

Definition (projection-set schema) We define the rewrite rules for projection term symbols for all proof symbols ψ_β and configurations Ω:

$$\begin{aligned} \mathrm{pr}^{\psi_\beta,\Omega}(0, x_1, \ldots, x_\alpha) &\to \Xi(\pi_\beta, \Omega) \\ \mathrm{pr}^{\psi_\beta,\Omega}(k+1, x_1, \ldots, x_\alpha) &\to \Xi(\nu_\beta(k+1), \Omega) \end{aligned}$$

for all $1 \leq \beta \leq \alpha$. Let $\gamma \in \mathbb{N}$ and let $\mathrm{pr}^{\psi_\beta,\Omega}\downarrow_\gamma$ be the normal form of $\mathrm{pr}^{\psi_\beta,\Omega}(\gamma, x_1, \ldots, x_\alpha)$ under the rewrite system just given extended with the term-rewriting rules for defined function and predicate symbols. Then, define $\Xi(\psi_\beta, \Omega) = \mathrm{pr}^{\psi_\beta,\Omega}$ and $\Xi(\Psi, \Omega) = \Xi(\psi_1, \Omega)$ and finally the *schematic projection term* $\Xi(\Psi) = \Xi(\Psi, \emptyset)$.

On the contrary to the clause term which unfolds to a set, the projection term unfolds to a set of proofs, for a given instance of the parameter. We use the following term-to-set transformation:

Definition (ground projection term) A projection term is called *ground* if there are no parameters and projection symbols in it.

Definition (projection term evaluation) Let Ξ be a ground projection term. Then we define a set of ground **LKS**-proofs $|\Xi|$ in the following way:

- $|A \vdash A| = \{A \vdash A\}$
- $|\rho(\Xi)| = \rho(|\Xi|)$ for unary rule symbols ρ
- $|w^{\Gamma\vdash\Delta}(\Xi)| = |\Xi|^{\Gamma\vdash\Delta}$

- $|\Xi_1 \oplus \Xi_2| = |\Xi_1| \cup |\Xi_2|$
- $|\Xi_1 \otimes_\sigma \Xi_2| = |\Xi_1| \times_\sigma |\Xi_2|$, a for binary rule symbol σ

For a ground **LKS**-proof π and configuration Ω we define $\mathrm{PR}(\pi, \Omega) = |\Xi(\pi, \Omega)|$ and the *standard projection set* $\mathrm{PR}(\pi) = \mathrm{PR}(\pi, \emptyset)$. For $\gamma \in \mathbb{N}$ we define $\mathrm{PR}(\Psi) \downarrow_\gamma = |\Xi(\Psi) \downarrow_\gamma|$.

Let us compute $\mathrm{PR}(\Psi) \downarrow_1$, i.e. for $k = 0$:

$$\mathrm{PR}(\Psi) \downarrow_1 = | \Xi(\Psi) \downarrow_1 | =$$
$$| w(\mathrm{pr}_0^{\varphi,\Omega_2}) \oplus w(\forall_l(\to_r (w(P(a) \vdash P(a)) \oplus w((P(g(\bar{1}, a)) \vdash P(g(\bar{1}, a))) \otimes_{\to_l} (P(h(\bar{1}, a)) \vdash P(h(\bar{1}, a))))))) |$$

We have : $\mathrm{pr}_0^{\varphi,\Omega_2} \to \Xi(\varphi(0), \Omega_2)$

Also we have $CL(\Psi) \downarrow_1 = CL(\sigma(1), \Omega_1) =$

$$= \{P(a) \vdash P(f(g(\bar{0}, a))) \; ; \; \vdash P(a) \; ; \; P(g(\bar{1}, a)) \vdash \}$$

and we map its clauses to the **LKS**-proofs:

$PR(\varphi, \; P(a) \vdash P(f(g(\bar{0}, a))) \;) :$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(f(g(\bar{0},a))) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle \twoheadrightarrow}}{P(a) \vdash P(a) \qquad P(f(a)) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle \to : l}}{P(a), P(a) \to P(f(a)) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle \to : l}}{P(a), \forall x(P(x) \to P(f(x))) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle \to : l}}{\forall x(P(g(\bar{1},x)) \to P(h(\bar{1},x))), P(a), \forall x(P(x) \to P(f(x))) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle w : l}}{\forall x(P(g(\bar{1},x)) \to P(h(\bar{1},x))), P(a), \forall x(P(x) \to P(f(x))) \vdash P(f(g(\bar{0},a))), P(a) \to P(h(\bar{1},a))}\,{\scriptstyle w : r}$$

$PR(\varphi, P(a) \vdash \;) :$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(a) \vdash P(a)}}{P(a), P(g(\bar{1},a)) \to P(h(\bar{1},a)) \vdash P(a)}\,{\scriptstyle w : l}}{P(a), P(g(\bar{1},a)) \to P(h(\bar{1},a)) \vdash P(a), P(h(\bar{1},a))}\,{\scriptstyle w : r}}{P(g(\bar{1},a)) \to P(h(\bar{1},a)) \vdash P(a), P(a) \to P(h(\bar{1},a))}\,{\scriptstyle \to : r}}{\forall x(P(g(\bar{1},x)) \to P(h(\bar{1},x))) \vdash P(a), P(a) \to P(h(\bar{1},a))}\,{\scriptstyle \forall : l}}{\forall x(P(x) \to P(f(x))), \forall x(P(g(\bar{1},x)) \to P(h(\bar{1},x))) \vdash P(a), P(a) \to P(h(\bar{1},a))}\,{\scriptstyle w : l}$$

$PR(\varphi, \; \vdash P(g(\bar{1}, a))) :$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(g(\bar{1},a)) \vdash P(g(\bar{1},a)) \qquad P(h(\bar{1},a)) \to P(h(\bar{1},a))}{P(g(\bar{1},a)), P(g(\bar{1},a)) \to P(h(\bar{1},a)) \vdash P(h(\bar{1},a))}\,{\scriptstyle \twoheadrightarrow}}{P(a), P(g(\bar{1},a)), P(g(1,a)) \to P(h(1,a)) \vdash P(h(1,a))}\,{\scriptstyle w:\,l}}{P(g(\bar{1},a)), P(g(\bar{1},a)) \to P(h(\bar{1},a)) \vdash P(a) \to P(h(\bar{1},a))}\,{\scriptstyle \to:\,r}}{P(g(\bar{1},a)), \forall x(P(g(\bar{1},x)) \to P(h(\bar{1},x))) \vdash P(a) \to P(h(\bar{1},a))}\,{\scriptstyle \forall:\,l}}{\forall x(P(x) \to P(f(x))), P(g(\bar{1},a)), \forall x(P(g(\bar{1},x)) \to P(h(\bar{1},x))) \vdash P(a) \to P(h(\bar{1},a))}\,{\scriptstyle w:\,l}$$

The formulas in blue color will appear as *atomic cuts* in the construction of the $ACNF$ in the next section.

3.3 Schematic $ACNF$

To produce an Atomic Cut Normal Form, we need to transform the resolution refutation into an **LKS**-proof skeleton. The $ACNF$ is produced by substituting each clause at the leaf nodes of this skeleton by the corresponding projections and appending necessary contractions at the end of the proof. A ground resolution term can be transformed into a tree straightforwardly:

Definition (resolution term to tree) Let ϱ be a ground resolution refutation. Then the transformation $T(\varrho)$ is defined inductively:

- if $\varrho = C$ for a clause C, then $T(\varrho) = C$
- if $\varrho = r(\varrho_1; \varrho_2; P)$, then $T(\varrho)$ is:

$$\dfrac{\dfrac{(T(\varrho_1))}{\Gamma \vdash \Delta, P, \ldots, P}\,c{:}\ r* \qquad \dfrac{(T(\varrho_2))}{P, \ldots, P, \Pi \vdash \Lambda}\,c{:}\ l*}{\Gamma, \Pi \vdash \Delta, \Lambda}\,cut$$

Example: In order to compute the $ACNF$ of the proof schema Ψ, first we should give a resolution refutation schema for the schematic characteristic clause set $CL(\Psi)$ defined above. Since the resolution schema is not constructable, i.e. we do not have an effective way how to construct it from the schematic clause set before the instantiation of the parameter. Therefore, we have to specify it by ourselves by looking at some instances of the schematic clause set:

$CL(\Psi) \downarrow_0 = CL(\pi_1, \Omega_1) = \{ \vdash \}$

$$CL(\Psi)\downarrow_1 = CL(\nu_1(\bar{1}),\Omega_1) = CL(\pi_2,\Omega_2)\cup\{\vdash P(a)\ ;\ P(g(\bar{1},a))\vdash\} =$$

$$= \{P(a)\vdash P(f(g(\bar{0},a)))\ ;\ \vdash P(a)\ ;\ P(g(\bar{1},a))\vdash\} \twoheadrightarrow$$

$$\twoheadrightarrow \{P(a)\vdash P(f(a))\ ;\ \vdash P(a)\ ;\ P(f(a))\vdash\}$$

$$CL(\Psi)\downarrow_2 = CL(\nu_1(2),\Omega_1) = CL(\nu_2(1),\Omega_2)\cup\{\vdash P(a)\ ;\ P(g(\bar{2},a))\vdash\} =$$

$$= CL(\pi_2,\Omega_2)\cup\{P(z_1)\vdash P(z_1)\ ;\ P(g(\bar{1},z_1))\vdash P(f(g(\bar{1},z_1)))\ ;$$
$$\vdash P(a)\ ;\ P(g(\bar{2},a))\vdash\} =$$

$$= \{P(a)\vdash P(f(g(\bar{0},a)))\ ;\ P(z_1)\vdash P(z_1)\ ;\ P(g(\bar{1},z_1))\vdash P(f(g(\bar{1},z_1)))\ ;$$
$$\vdash P(a)\ ;\ P(g(\bar{2},a))\vdash\} =$$

$$= \{P(a)\vdash P(f(a))\ ;\ P(z_1)\vdash P(z_1)\ ;\ P(f(z_1))\vdash P(f(f(z_1)))\ ;$$
$$\vdash P(a)\ ;\ P(f(f(a)))\vdash\}$$

Let $R = ((\varrho,\delta),\mathcal{R})$, where $\mathcal{R}$ is the following rewriting system:

$$\begin{array}{rcl}
\varrho(\bar{0},x) & \to & r(\delta(\bar{0},x)\ ;\ P(a)\vdash\ ;\ P(a)) \\
\varrho(k+1,x) & \to & r(\delta(k+1,x)\ ;\ P(f(g(k,z_k)))\vdash\ ; P(f(g(k,z_k)))\ ) \\
\delta(\bar{0},x) & \to & \vdash P(a) \\
\delta(k+1,x) & \to & r(\delta(k,x)\ ; P(g(k,z_k))\vdash P(f(g(k,z_k)))\ ; P(g(k,z_k))\ )
\end{array}$$

Computing $\delta(1,x)$ and transforming the resulting ground term to a tree we see indeed that it is a resolution refutation $\mathcal{R}$ of Ψ for instance $k=0$:

$$\dfrac{\dfrac{\vdash P(a) \qquad P(a)\vdash P(f(g(\bar{0},a)))}{\vdash P(f(g(\bar{0},a)))}\,cut \qquad P(g(\bar{1},a))\vdash}{\vdash}\,cut,\twoheadrightarrow$$

Mapping the proof projections computed above to $\mathcal{R}$ and considering the term-rewriting system for g we obtain the $ACNF$ of Ψ for $k=0$. The cuts are marked with red color and they are all atomic:

$$\dfrac{\dfrac{\dfrac{PR(P(a)\vdash)}{A\ ;\ B\vdash P(a)\ ;\ C} \qquad \dfrac{PR(\ P(a)\vdash P(f(g(\bar{0},a)))\ )}{A\ ;\ P(a)\ ;\ B\vdash P(f(g(\bar{0},a)))\ ;\ C}\ cut}{A\ ;\ B\ ;\ A\ ;\ B\vdash P(f(g(\bar{0},a)))\ ;\ C} \qquad \dfrac{PR(\vdash P(g(\bar{1},a))\ )}{A\ ;\ P(g(\bar{1},a))\ ;\ B\vdash\ ;\ C\ ;\ C}\ cut,\twoheadrightarrow}{\dfrac{A\ ;\ B\ ;\ A\ ;\ B\ ;\ A\ ;\ B\vdash C\ ;\ C\ ;\ C}{A\ ;\ B\vdash C}\ c\colon l^*, c\colon r^*}$$

where $A \equiv \forall x(P(x) \to P(f(x)))$, $B \equiv \forall x(P(g(\bar{1},x)) \to P(h(\bar{1},x)))$ and $C \equiv P(a) \to P(h(\bar{1},a))$.

Chapter 4

The GAPT System

The **GAPT** system (General Architecture for Proof Theory) is a proof theory framework that aims at providing data structures, algorithms and user interfaces for analyzing and transforming formal proofs. **GAPT** was created to replace and expand the scope of the *CERes* system [BHL+05]. The *CERes* system was defined mainly for cut-elimination by resolution in first-order logic [BL00]. Through a more flexible implementation based on basic data structures for simply-typed lambda calculus and for sequent and resolution proofs, implemented in the hybrid functional object-oriented language Scala [OSV10], **GAPT** has already allowed the generalization of the cut-elimination by resolution method to proofs in higher-order logic [HLWWP08a] and to proof schemata [Ruk12]. Furthermore, methods for structuring and compressing proofs, such as cut-introduction [HLW12] and Herbrand sequent extraction [WP08] have recently been implemented.

So far the development of the **GAPT** system has led to a general purpose experimental tool, rather than just an implementation of the *CERES* method. It contains an implementation of various data structures and data types required for various algorithms, parsers, proof transformations, theorem provers, graphical user interface as well as a command line interface (for the advanced UNIX users) and another components which are very useful in the automated analysis of proof theoretic constructions.

GAPT contains datastructures which implement the languages of first- and higher-order logic as well as the regular schematic logic [ACP09]. All of them are contained in the corresponding types in order to model the mathematical theory in a sound way. Several Gentzen-type calculi were implemented such as **LK** for first-order logic and Robinson resolution [Rob65] for proving unsatisfiability. Furthermore, several new calculi were developed

and implemented such as **LKsk** [HLWWP08a] and **LKskc** - for higher-order logic, **LKS** - for first-order regular schemata [Ruk12]. The algorithms implemented within **GAPT** cover the standard set of proof theoretic concepts used in automated deduction such as unification, subsumption, skolemization, normalization, term-rewriting, matching, auto-propositional mode for proving propositional tautologies, etc. Some of the algorithms, such as the higher-order unification algorithm of Huet [Hue02], require an interactive mode because, as it is well known, this problem is undecidable.

Several parsers are implemented to make the use of the algorithms and transformations easier for the end user. The input format for all objects such as first-, higher- or schematic formulas, sequents, proof derivations, term-rewriting systems, etc. is very intuitive and easy for typing in a .txt file. A language called *SHLK* has been described and implemented for parsing **LKS**-proofs. A language for parsing resolution proofs and clause schemata has also been implemented. The latter is absolutely necessary for the $CERES_s$ method, because the user should provide the corresponding rewrite systems for obtaining the *ACNF* as it was shown in the previous chapter.

The core of the system has several transformations on proof derivations. The most prominent transformation is the one that automates the *CERES* method. Such transformations exist for $CERES_s$ and $CERES_\omega$ [HLWWP08a], Gentzen's reductive cut-elimination method, and Herbrand sequent extraction as well. From now on we will focus our attention on the automation of $CERES_s$ method which is the main contribution of the this thesis. As a part of the automation, a parser was implemented for the input language. The basic theoretical backbone of $CERES_s$ such as the schematic characteristic clause set, schematic proof projections and the construction of the *ACNF*, as well as all the term-rewriting machinery has also been implemented. Finally, this process was built into the graphical user interface *ProofTool* [DLL$^+$12b] and presented to the user in a human readable format.

The resolution prover *TAP* has been integrated into the **GAPT** system and was extended to support proof replaying. It turned out to be very useful in analyzing of mathematical proofs such as the "Tape proof" [DLL$^+$12a]. Proof replaying increases the comprehensibility of the output for mathematicians. The proofs obtained using Prover9 [McC10] tend to be illegible for untrained users. Namely, *TAP* unfolds some complex inferences, recomputes the unifiers and translates Prover9's inference rules to predefined corresponding sequences of resolution and paramodulation steps. A good example for that is the **rewrite** inference [McC10] of Prover9.

All of the features which have been just mentioned are integrated into a graphical user interface called *ProofTool*. With its navigation, buttons and pop-up menus it is a user-friendly environment for loading and displaying proofs and transformations which were applied on them. Also it has a functionality such as the marking of cut-ancestors up to the axioms and hiding branches of the proof derivation order to help visualize big proofs. It also has features for searching a formula in the proof. Furthermore, it can visualize, in a nice format, the characteristic clause and projection terms. Finally, each object can be exported to a LaTeX or PDF format and plugged into a paper, presentation, etc.

Now we go deeper into the construction of the **GAPT** system and take a more engineering point of view. The programming language of choice was *Scala* [OSV10]. There are several reasons for this choice. The most important one is that *Scala* is a multi-paradigm programming language. On one hand, it is considered as a "better Java", i.e. it has all the benefits that object-oriented programming provides : defining (algebraic) objects and types, abstraction over datastructures, encapsulation, modularity, polymorphism, inheritance. For example, the polymorphism allows the application of some design patterns. The benefit of applying design patterns is that they reveal relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. For example, such a design-pattern is the "Visitor design pattern" which could be integrated in the algorithms which traverse the proof tree. This will be explained in details in the next chapter. Another benefit of using *Scala* is that, as a Java-based language, it is built on top of a virtual machine which allows the **GAPT** system to be installed on any operating system. The handling of exceptions and the garbage collector decrease the number of possible bugs in the system. Due to the complex algorithms, this was one of the main problems with the predecessor of **GAPT** - the old system *CERes* [HLWWP08d].

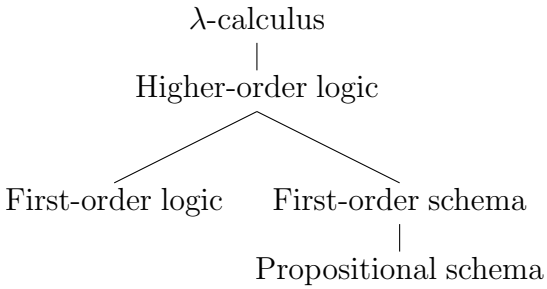
On the other hand, *Scala* also allows the use of the functional programming paradigm which treats computation as the evaluation of mathematical functions. Functional programming has its roots in the λ-*calculus* - a formal system developed in the 1930s to investigate function definition, function application and recursion. This perfectly fits into our needs, because almost all of the objects in the language as well as in the calculi used in Proof theory are defined via recursion/induction. Therefore, the implementation follows the theory. This does not only support the creation of a correct code, but also makes the implementation of the algorithms straightforward. It also al-

lows the application of higher-order functions which are very efficient when a certain operation/function is applied on objects of the same type. The result of its application can be either an object or again a function. For example, imagine that we want to apply a function on a proof derivation which contains as a parameter a function which is applied only on the cut-ancestors of each node of the proof derivation.

Now we will focus on the different levels of abstraction on top of which are implemented the corresponding languages, calculi, algorithms and transformations.

4.1 Layers of abstraction

The **GAPT** system has the following general architecture:



As it is shown in the picture, on the top of the hierarchy is the λ-calculus. It forms the kernel of the whole system. On one hand this formalism is strong enough to describe all computable functions, on other hand the implementation of the calculus is straightforward. Also the recursive definition of the terms uses the full power of the functional programming style where every object is an expression. Since we use the typed λ-calculus, the corresponding type is attached to each expression.

On the base of the λ-calculus the language for higher-order logic is built. For example, atoms and functions are represented as applications. The difference between the two concepts is their types - the atoms have type $\tau \to o$, whereas functions have type $\tau \to \iota$, where τ, o and ι are the standard types

introduced in chapter Preliminaries.

The first order schema logic layer is based on the layer of higher-order logic. This allows us to define second-order variables which are applied on a ground indexed expression and the resulting object serves as a (indexed) first-order variable. Another reason is that in schema language we have the special constants for $\bigvee$ and $\bigwedge$. The details will be explained in the following sections.

4.1.1 λ-calculus layer

The λ-calculus is a formal system in mathematical logic for expressing computation with variable binding and substitution. It was first formulated by Alonzo Church as a way to formalize mathematics through the notion of functions, in contrast to the field of set theory. The λ-calculus found a big successes in the area of computability theory and computer science. It is also the basis of the functional programming paradigm. The λ-calculus consists of a language of lambda terms along with an equational theory. Since the names of functions are largely a maner of convenience, the lambda calculus has no means of naming a function. Furthermore, since all functions expecting more than one argument can be transformed into equivalent functions accepting a single input (via Currying), the lambda calculus has no need to create a function that accepts more than one argument. Finally, since the names of arguments are largely irrelevant, the notion of equality on λ-terms is α-equivalence. The λ-terms are:

- all variables x of type ι or ω
- if t is a λ-term of arbitrary type τ and x is a variable $\tau_1 \in \{\iota, \omega\}$, then $\lambda x.t$ is a λ-term of type $\tau_1 \to \tau$ (λ-abstraction)
- if t and s are λ-terms of type $\tau_1 \to \tau_2$ and τ_1 respectively, then ts is a λ-term of type τ_2 (λ-application)

The variable x in the second case is called a *bound variable*. Associating to each term a corresponding type is crucial for the termination. Otherwise, we can end up in the following case : $(\lambda x.x)(\lambda x.x)$ which is a loop. In the **GAPT** system we define the notion of λ-expression in the following way:

```
trait LambdaExpression extends LambdaFactoryProvider w i t h Ordered[LambdaExpression] {
  def type : TA
```

```
  def getFreeAndBoundVariables() : Pair[Set[Var],Set[Var]]
  def noUnboundedBounded : Boolean
  def variant
  ...
}
```

The meaning of the declaration above is the following : an abstract type LambdaExpression is defined. It has components exptype which associates a type to this object and a function getFreeAndBoundVariables which returns a pair of sets containing the free and the bound variables respectively. For the types we also have the following declaration:

```
abstract class TA {
  def →(this:TA)
}
abstract class TAtomicA extends TA
abstract class TComplexA extends TA
case class Tindex() extends TAtomicA {override def toString = "ω"}
case class Ti() extends TAtomicA {override def toString = "ι"}
case class To() extends TAtomicA {override def toString = "o"}
case class →(in:TA, out:TA) extends TComplexA {override def toString = ...}
```

We see that this declaration defines the abstraction for the three types ι, ω and o which are atomic types (TAtomicA) exactly as they were defined in chapter Preliminaries. We also see the definition the constructor $\rightarrow$ for complex types (TComplexA). Both the atomic type objects and the complex type one are wrapped by the abstract class TA (abbreviation from TypeAbstract). This wrapping of all types into one is necessary because the algorithms do not know in advance the concrete type of the objects they work on. The objects are decomposed during runtime.

Having the types, we can now show how the λ-terms are defined in **GAPT**:

```
class Var protected[typedLambdaCalculus](name: SymbolA, exptype: TA, dbInd: Option[Int])
    extends LambdaExpression {
  val dbIndex: Option[Int] = dbInd // represents a bound variable and its de Bruijn index
  override def equals(a: Any) // alpha equals as ignores bound variable names
  def isFree = dbIndex == None
  def isBound = !isFree
  ...
}
```

We see that a variable is described by its name, type and *de Bruijn index* which is a very nice solution to distinguish the nested bound variables in a λ-term. De Bruijn indices will be described later. We also see that the object representing a variable has a function which answers the question whether it is free or bound. There is also an implementation of the α-equality.

```
class Abs protected[typedLambdaCalculus](variable: Var, expression: LambdaExpression)
      extends LambdaExpression {
   require (variable.isFree, "Cannot abstract over a bound variable!")
   def exptype: TA = →(variable.exptype, expression.exptype)
   override def equals(a: Any)
   def syntaxEquals(e: LambdaExpression)
   override def toString() = "Abs(" + variableInScope + "," + expressionInScope + ")"
   def variant
   def createDeBruijnIndex(Var, LambdaExpression, nextDBIndex): LambdaExpression
   ...
}
```

We see that the description of the λ-abstraction is characterized by a variable and a λ-expression. It has a constructor $\rightarrow$ which makes the corresponding type. It has also a requirement that the variable which we abstract over must not be already bounded. It also has a function for computing the de Bruijn index of the variable we abstract over. We will explain these construction in details later.

Finally, we show the declaration of the application:

```
class App protected[typedLambdaCalculus](function: LambdaExpression, argument: LambdaExpression)
      extends LambdaExpression {
   require(...) //Correct types for constructing the application function(argument)
   def variant
   def exptype: TA
   override def equals(a: Any)
   def syntaxEquals(e: LambdaExpression)
   override def toString() = "App(" + function + "," + argument + ")"
   ...
}
```

Like the previous two λ-terms, the *application* also has a type. It also has a requirement which checks whether the construction of the object is sound,

namely whether the corresponding types of the objects fit. Furthermore, there is a function which returns a variant of the resulting term.

The λ-terms in the **GAPT** system are created via a design pattern called *Abstract factory*. The pattern separates the details of implementation of a set of objects from their general usage. It defines a functionality via a generic interface which is specified in each level in the hierarchy, i.e. for the layers for first-order logic, higher-order logic and first-order schema. Exactly here we employ the object-oriented paradigm of *Scala*. Here is how the abstract interface looks like:

```
t r a i t LambdaFactoryA {
  def createVar(name: SymbolA, exptype: TA, dbInd: Option[Int])
  def createAbs(variable: Var, exp: LambdaExpression )
  def createApp(fun: LambdaExpression, arg: LambdaExpression )
}
```

Each object of a language from each layer is created by using this interface. It has only three functions, namely the ones who create the λ-terms. In the layer for λ-calculus we define the interface and give it a meaning, i.e. we construct the real objects/terms:

```
o b j e c t LambdaFactory extends LambdaFactoryA {
  def createVar(...) : Var = new Var( name, exptype, dbInd )
  def createAbs(...) : Abs = new Abs( variable, exp )
  def createApp(...) : App = new App( fun, arg )
}
```

An interesting question from a software engineering point of view is how in **GAPT** we distinguish between (nested) bound and free variables in a λ-term. They are many possibilities, but the most efficient so far is the one suggested by Dutch mathematician Nicolaas de Bruijn. He assigns to a variable an index which makes the λ-term invariant under α-conversion. Hence, the check for α-equivalence is the same as the check for syntactic equality.

Definition (de Bruijn index) A *de Bruijn index* is a natural number that represents an occurrence of a variable in a λ-term and denotes the number of binders that are in the scope between that occurrence and its corresponding binder. The definition of a de Bruijn indexed λ-term is:

- n, where $n \in \mathbb{N}$ and

  - n is an index of a bound variable, if it is in the scope of at most n binders
  - n is an index of a free variable, otherwise

- if M and N are de Bruijn indexed λ-terms, then MN is also a de Bruijn indexed λ-term
- if M is a de Bruijn indexed λ-term, then λM is also a de Bruijn indexed λ-term

Example: Consider the K-combinator $\lambda x.\lambda y.x$. Its de Bruijn indexing is : $\lambda.\lambda.2$. That means that the index of x is 2, i.e. the second binder which x is in the scope of, starting counting from the occurrence of x to the left. In the term there is no variable y, although there is a binder for y. Therefore, we have only one de Bruijn index in this term. Another example, consider the S-combinator $\lambda x.\lambda y.\lambda z.xz(yz)$. Its de Bruijn indexing is : $\lambda.\lambda.\lambda.$ 3 1 (2 1). That means that x, y and z have de Bruijn indices 3, 2 and 1, respectively. Starting from the occurrence of x to the left there are 3 binders is scope; starting from the occurrence of y to the left there are 2 binders in scope, etc.

A free-variable in a λ-term has a de Bruijn index which is an arbitrary number bigger then the depth of the deepest nesting of binders. For example, in the term $\lambda x.yx$ we have that x has a de Bruijn index 1 and y has a de Bruijn index 2. When β-reduction is applied to a λ-term, the de Bruijn indices of all variables should be recomputed. This is done as follows:

- find all variables n in M that are bound in M
- decrease the free variables of M *to* so that they are still bigger then the number of the binders in scope
- replace n in M with N and, if necessary, increasing the free variables so that they are still bigger then the number of the binders in scope

Example: Consider the λ-term $(\lambda x.\lambda y.zx(\lambda u.ux))(\lambda x.wx)$. Its de Bruijn indexing before the application of the β-reduction is:

$$(\lambda\lambda\ 4\ 2\ (\lambda\ 1\ 3))\ (\lambda\ 5\ 1)$$

After one step of β-reduction we obtain the λ-term : $\lambda y.z(\lambda x.wx)(\lambda u.u(\lambda x.wx))$ which corresponds to the term λ 3 (λ 6 1) (λ 1 (λ 7 1)).

So far we described the kernel of the **GAPT** system. We continue with the next layer which defines the higher-order logic on top of the λ-calculus.

4.1.2 Higher-order logic layer

In the **GAPT** system the layer for higher-order logic (HOL) is divided in two parts - a language and a calculi module. In the language module the data structures representing HOL-terms are defined . The abstractions for HOL-constants and HOL-variables are built on top of the definition for Var in the λ-calculus layer. The difference comes from the additional data which specifies whether the name of the object is of type constant symbols or variable symbols:

```
class HOLVar protected[hol] (name: VariableSymbolA, exptype: TA, dbInd: Option[Int])
    extends Var(name, exptype, dbInd) with HOLExpression {
    ...
}
class HOLConst protected[hol] (name: ConstantSymbolA, exptype: TA)
    extends Var(name, exptype, None) with Const with HOLExpression {
    ...
}
```

Due to the polymorphism the datastructures HOLVar and HOLConst inherit all features which the objects of type Var have. That means that both of them have a type associated. In contrast to the HOL-variables, the HOL-constants do not have a de Bruijn index as it is seen from the code above - in the first case we have dbInd, in the second we have None (in bold).

The functions and the atoms are defined as λ-application. An HOL-variable or an HOL-constant with the corresponding type are applied to a list of arguments with the corresponding type. The difference between the two applications is that in the case of an atom we construct an object of type o, whereas in the case of function we construct an object of type ι. For the construction of the application we use again a factory, namely HOLFactory, which is similar to the LambdaFactory, but with additional information:

```
object HOLFactory extends LambdaFactoryA {
    def createVar( name: SymbolA, exptype: TA, dbInd: Option[Int]) : Var =
```

```
      name match {
       case a: ConstantSymbolA =>
         if (isFormula(exptype))
            new HOLConstFormula(a)
         else
            new HOLConst(a, exptype)
       case a: VariableSymbolA =>
         if (isFormula(exptype))
            new HOLVarFormula(a, dbInd)
         else
            new HOLVar(a, exptype, dbInd)
    }
    def createApp( fun: LambdaExpression, arg: LambdaExpression ) : App = {...}
    def createAbs( variable: Var, exp: LambdaExpression ) : Abs = {...}
    def isFormula(typ: TA): Boolean = {...}
    ...
}
```

Having defined the atoms, the next step is to define the datastructures for formulas in the language. This is done by using predefined constants - for each unary or binary logical connective there is a constant variable of the corresponding type. For example, the constants are defined as follows:

```
case object NegC extends HOLConst(NegSymbol, "(o → o)")
case object AndC extends HOLConst(AndSymbol, "(o → (o → o))")
...
```

The constructors of the corresponding applications are defined below. The arguments respect the type of the corresponding logical connective:

```
object Neg {
    def apply(sub: HOLFormula) = App(NegC,sub).asInstanceOf[HOLFormula]
    ...
}
object And {
    def apply(left: HOLFormula, right: HOLFormula) = (App(App(AndC,left),right)).asInstanceOf[HOLFormula]
    ...
}
```

For the quantifiers we also have predefined logical connectives:

```
class ExQ protected[hol](e:TA) extends HOLConst(ExistsSymbol, →(e,"o"))
class AllQ protected[hol](e:TA) extends HOLConst(ForallSymbol, →(e,"o"))
```

and the corresponding constructors of the quantified formulas are:

```
object Ex {
  def apply(sub: LambdaExpression) = App(new ExQ(sub.exptype),sub).asInstanceOf[HOLFormula]
  ...
}
object All {
  def apply(sub: LambdaExpression) = App(new AllQ(sub.exptype),sub).asInstanceOf[HOLFormula]
  ...
}
```

Until now we explained the datastructures for the HOL-language. In the **GAPT** system we also support different calculi such as **LK, LKS, LK**$^{\omega}$, etc. Now, we describe how a derivation in **GAPT** looks like. In our case all derivations (**LK**, Robinson Resolution, etc) are trees containing sequents and some additional information in the nodes, for example the auxiliary and principal formulas, the type of the inference rule, etc. We have the following abstractions for a tree:

```
trait Tree[V] extends AGraph[V] {
  require {isTree}
  private[trees] def isTree: Boolean
  val vertex: V
  def name: String
  ...
}
class LeafTree[V](override val vertex: V) extends LeafAGraph[V](vertex) with Tree[V] {
  ...
}
class UnaryTree[V](override val vertex: V, override val t: Tree[V])
                        extends UnaryAGraph[V](vertex, t) with Tree[V] {
  ...
}
class BinaryTree[V](override val vertex: V, override val t1: Tree[V], override val t2: Tree[V])
                        extends BinaryAGraph[V](vertex, t1, t2) with Tree[V] {
  ...
}
```

The three data structures above are a basis for describing the nullary (axioms and proof-links), unary and binary inference rules in all calculi in **GAPT**. We now describe the notion of a proof in the system : this is a tree with a root, inference rule and subproof(s):

```
traitProof[+V] extends AGraph[V] {
  def root = vertex
  def rule: RuleTypeA
  ...
}
trait NullaryProof[+V] extends LeafAGraph[V] with Proof[V] {
  ...
}
trait UnaryProof[+V] extends UnaryAGraph[V] with Proof[V] {
  def uProof = t.asInstanceOf[Proof[V]]
  ...
}
trait BinaryProof[+V] extends BinaryAGraph[V] with Proof[V] {
  def uProof1 = t1.asInstanceOf[Proof[V]]
  def uProof2 = t2.asInstanceOf[Proof[V]]
  ...
}
```

Merging the above abstractions in a suitable way, we obtain the description of a proof-tree with still not specified vertex of type V:

```
trait NullaryTreeProof[V] extends LeafTree[V] with NullaryProof[V] with TreeProof[V]
trait UnaryTreeProof[V] extends UnaryTree[V] with UnaryProof[V] with TreeProof[V]
trait BinaryTreeProof[V] extends BinaryTree[V] with BinaryProof[V] with TreeProof[V]
```

The specification depends on the type of the calculus. For **LK**, for example, the vertex of a proof-tree is a sequent (pair of multisets of formulas) and an **LK**-proof is described as follows:

```
trait LKProof extends TreeProof[Sequent] with Tree[Sequent] {
  ...
}
trait NullaryLKProof extends LeafTree[Sequent] with LKProof with NullaryTreeProof[Sequent] {
  ...
}
trait UnaryLKProof extends UnaryTree[Sequent] with LKProof with UnaryTreeProof[Sequent] {
```

```
    override def uProof = t.asInstanceOf[LKProof]
    ...
}
trait BinaryLKProof extends BinaryTree[Sequent] with LKProof with BinaryTreeProof[Sequent] {
    override def uProof1 = t1.asInstanceOf[LKProof]
    override def uProof2 = t2.asInstanceOf[LKProof]
    ...
}
```

Finally, the last feature needed for defining the object of an **LK**-proof is the datastructures for auxiliary and principal formulas. With their help we create the inference rules. For example, the cut rule:

```
object CutRule {
    def apply(s1: LKProof, s2: LKProof, term1oc: Occurrence, term2oc: Occurrence) = {
      ...
      new BinaryTree[Sequent](sequent, s1, s2) with BinaryLKProof with AuxiliaryFormulas {
        def rule = CutRuleType
        def aux = (term1::Nil)::(term2::Nil)::Nil
        override def name = "cut"
      }
      ...
}
```

All other inference rules of all other calculi which the **GAPT** system supports are implemented in a similar way.

4.1.3 First-order logic layer

The first-order logic layer is built on top of the higher-order logic layer, because its specification is less general. That means that the creation of first-order formulas can be done like the creation of a second-order formulas with additional restrictions, namely the first-order terms should be of type ι. Therefore, analogously to the previous section, we have to define a first-order *factory* which creates the elements of the first-order language. This construction is analogous to the ones showed previously and we will not show it here. We will present only the abstraction for first-order term and the type of first-order variable which is in fact a higher-order variable of type ι:

```
trait FOLTerm extends HOLExpression with FOL {
   require( exptype == ι )
   ...
}
class FOLVar (name: VariableSymbolA, dbInd: Option[Int])
                     extends HOLVar(name, ι, dbInd) with FOLTerm {
   ...
}
```

4.1.4 First-order schemata layer

First-order schemata layer is based on HOL, because it goes beyond the FOL. In this section we look at some software solutions for schemata. Again, we have a special factory for creating schema expressions:

```
trait Schema extends HOL {
   override def factory: LambdaFactoryA = SchemaFactory
}
trait SchemaExpression extends HOLExpression with Schema
```

Here we define the abstraction for an integer term of type ω. Having this, we can define a schema formula, arithmetic term, schematic term, etc:

```
trait SchemaFormula extends SchemaExpression with HOLFormula
object aTerm {
   def apply(name: HOLConst, ind: IntegerTerm): IntegerTerm = {
      SchemaFactory.createApp(name, ind).asInstanceOf[IntegerTerm]
   }
}
object sTerm {
   def apply(f: String, i: IntegerTerm, x: HOLExpression): HOLExpression = {
      val func = HOLConst(new ConstantStringSymbol(f), →(ω , → (ι, ι)))
      return HOLApp(HOLApp(func, i), x).asInstanceOf[HOLExpression]
}
```

The schema iterations $\bigvee$ and $\bigwedge$ are defined as higher-order constants of type $((\omega \to o) \to (\omega \to (\omega \to o)))$:

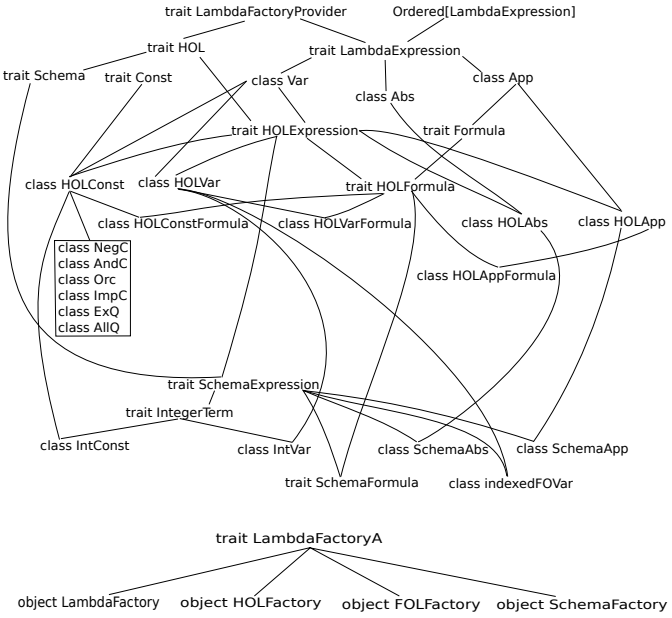


Figure 4.1: Class hierarchy

case object BigAndC extends HOLConst(BigAndSymbol, $\to(\to(\omega, o), \to(\omega, \to(\omega, o))))$ with Schema
case object BigOrC extends HOLConst(BigOrSymbol, $\to(\to(\omega, o), \to(\omega, \to(\omega, o))))$ with Schema

The last datastructure which will be presented here is that for an indexed first-order variable. This is an object which is in fact a higher-order variable of type $\omega \to \iota$:

class indexedFOVar(override val name: VariableStringSymbol) extends HOLVar(name, $\omega \to \iota$) {
  ...
}

In the **GAPT** system we have extended **LK** with new inference rules for the **LKS** calculus which are defined in a similar fashion to the ones in **LK**, so we will not describe them here. Instead, we will give a part of the class hierarchy showing the connection between the layers:

4.1.5 Higher-order functions

One of the biggest benefits of using the functional programming paradigm are the *higher-order functions*. These are functions that take another functions as an argument or they return a function as a result. For example, the

Map function is a higher-order function. It takes as arguments a function f and a list of elements, and as result, returns a new set with f applied to each element from the list. For example, let $f\colon \omega \to \omega$ be the square function $\lambda x.x * x$ and $S = \{1, 2, 3\}$ be a set of elements of type ω. Then, the function $Map : (\omega \to \omega, \mathbb{N}) \to \mathbb{N}$ on input (f, S) gives $\{f(1), f(2), f(3)\} = \{1, 4, 9\}$.

Another important higher-order function is $Fold$ (also know as accumulator). It takes as arguments a binary higher-order function g, an initial value $init$ of the type of $Range(g)$ and a set S of elements of the type of $Range(g)$. For example, if $g : \omega \to \omega \to \omega$ is defined as $\lambda x \lambda y.x + y$, $init = 0$ and $S = \{1, 2, 3\}$, then $Fold(g, init, S)$ returns $(((3 + 2) + 1) + 0) = 6$ which is the sum of the elements of S.

In the **GAPT** system we use the higher-order functions in many cases, for example to construct the type of the higher-order constant h for the s-term $h(k + 1, x)$, where $k + 1$ is an arithmetic expression of type ω, $x\colon \iota$, we have to create a higher-order constant $h\colon \omega \to \iota \to \iota$. This we do as follows:

$Fold(\to, \iota, Map(getType, \{k + 1, x\})$

and the result of it, i.e. the type of h, is: $\omega \to \iota \to \iota$, where $\to$ is the constructor for types in λ-calculus.

Remark: *The higher-order functions in Scala are defined in such a way that they can be applied to objects of different type. But it is important that there is a type matching of the corresponding types.*

In **GAPT** the following piece of code corresponds to the last example above:

```
{
  ...
  val type = args.map(x => x.exptype).foldRight(ι)((x,t) => →(x, t))
  val h = HOLConst("h", type)
  ...
},
```

where args is a list of arguments, exptype is the type of the expression x and h is the higher-order constant that we want to create.

Chapter 5

The language SHLK

SHLK is a language for fast and convenient typing of **LK**-style proofs. It is a successor of the language handy LK (*HLK*) [HLWWP08c]. *SHLK* is invented mainly for the needs of the system $CERES_s$, but it can be used also in every system which works with **LK** and **LKS** calculi. Also a subset of it can be used by any system which works with higher-order formulas or first-order regular schemata. In the **GAPT** system an LL-parser is implemented which parses the grammar of the language *SHLK*. *SHLK* is superior to its ancestor *HLK* in several ways. First, it is more expressive than *HLK*, because it handles schematic definitions in a suitable way. Second, the user has a better control on the inferences. For example, *HLK* does automatically some structural inferences which may change the ancestor relation in a way that it differs from the proof which was meant by the user. Third, in *SHLK* the sequents are pairs of multisets. That eliminates the need for permutation rules which were part of *HLK*. Fourth, looking from the software engineering point of view, the parser of *SHLK* creates objects directly and datastructures defined in the logical layers which we explained in the previous chapter. Therefore, the **GAPT** system can directly use these objects in its transformations. On the contrary, the output of *HLK* was an intermediate .xml file which later was parsed from the former system *CERes*. Fifth, the parser for *SHLK* can be and is integrated directly in the new *ProofTool* which makes the whole automation process, from writing the input file to loading it and displaying the proofs, more transparent and user-friendly.

5.1 A Grammar for $SHLK$

The proof input format is designed to write **LKS**-proofs in a machine readable form using ASCII characters. It extends the input format of the *RegSTAB* system [ACP10]. Below we define a formal grammar of *SHLK*. We use the Kleene's concept of regular expressions:

$$
\begin{array}{rcl}
\langle lks\_file\rangle & ::= & [\langle lks\_statement\rangle]^* \\
\langle lks\_statement\rangle & ::= & \langle definition\rangle \\
 & ::= & \langle proof\rangle \\
\langle definition\rangle & ::= & \langle formula\rangle := \langle formula\rangle \\
\langle proof\rangle & ::= & proof\ \langle proof\_name\rangle\ proves\ \langle sequent\rangle \\
 & ::= & base\ \{inference\_list\} \\
 & ::= & step\ \{inference\_list\} \\
\langle proof\_name\rangle & ::= & [\backslash]^*[a-z, 0-9]^+ \\
\langle sequent\rangle & ::= & [\langle formulaList\rangle]\ \ |-\ \ [\langle formulaList\rangle] \\
\langle formulaList\rangle & ::= & \langle formula\rangle \\
 & ::= & \langle formula\rangle, \langle formulaList\rangle \\
\langle inference\_list\rangle & ::= & [\langle inference\rangle]^+ \\
\langle inference\rangle & ::= & \langle id\rangle\ :\ \langle rule\rangle \\
 & ::= & \langle root\rangle\ :\ \langle rule\rangle \\
\langle id\rangle & ::= & [0-9, a-z]^+ \\
\langle predicateName\rangle & ::= & [A-Z]^+[a-z, 0-9]^* \\
\langle indexedPredicate\rangle & ::= & \langle predicateName\rangle(\langle aTermList\rangle) \\
\langle atom\rangle & ::= & \langle foAtom\rangle \\
 & ::= & \langle sAtom\rangle \\
\langle foAtom\rangle & ::= & \langle foAtomName\rangle(\langle foTermList\rangle) \\
\langle sAtom\rangle & ::= & \langle sAtomName\rangle(\langle aTerm\rangle, \langle foTermList\rangle) \\
\langle atom\rangle & ::= & \langle foAtom\rangle \\
 & ::= & \langle sAtom\rangle \\
\langle foAtom\rangle & ::= & \langle foAtomName\rangle(\langle foTermList\rangle) \\
\langle atom\rangle & ::= & \langle foAtom\rangle \\
 & ::= & \langle sAtom\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle atom\rangle & ::= & \langle foAtom\rangle \\
 & ::= & \langle sAtom\rangle \\
\langle foAtom\rangle & ::= & \langle foAtomName\rangle(\langle foTermList\rangle) \\
\langle sAtom\rangle & ::= & \langle sAtomName\rangle(\langle aTerm\rangle, \langle foTermList\rangle) \\
\langle aTerm\rangle & ::= & \langle aVar\rangle \ + \ \langle aConst\rangle \\
 & ::= & \langle aConst\rangle \\
 & ::= & \langle aVar\rangle \\
\langle indexedFOvar\rangle & ::= & \langle indexedFOvarName\rangle(\langle aTerm\rangle) \\
\langle foTerm\rangle & ::= & \langle foVar\rangle \\
 & ::= & \langle foConst\rangle \\
 & ::= & \langle indexedFOvar\rangle \\
 & ::= & \langle foTermSymbol\rangle(\langle foTermList\rangle) \\
 & ::= & \langle foTermSymbol\rangle(\langle sTermList\rangle) \\
\langle foConst\rangle & ::= & [a, b, c, d][0 - 9]^* \\
\langle foVar\rangle & ::= & [x, y][0 - 9]^* \\
\langle aVar\rangle & ::= & [i, j, k, l, m, n]^+[0 - 9]^* \\
\langle aConst\rangle & ::= & [0 - 9]^+ \\
\langle indexedFOVar & ::= & [z][0 - 9]^* \\
\langle foTermList\rangle & ::= & \langle foTerm\rangle \\
 & ::= & \langle foTerm\rangle, \langle foTermList\rangle \\
\langle aTermList\rangle & ::= & \langle aTerm\rangle \\
 & ::= & \langle aTerm\rangle, \langle aTermList\rangle \\
\langle sTerm\rangle & ::= & \langle sTermName(\langle aTerm\rangle, [\langle foTermList\rangle]^*, [\langle aTermList\rangle]^*) \\
\langle term\rangle & ::= & \langle foTerm\rangle \\
 & ::= & \langle sTerm\rangle \\
\langle formula\rangle & ::= & \langle indexedPredicate\rangle \\
 & ::= & \sim \langle formula\rangle \\
 & ::= & \langle atom\rangle \\
 & ::= & \langle formula\rangle /\backslash \langle formula\rangle \\
 & ::= & \langle formula\rangle \backslash / \langle formula\rangle \\
 & ::= & \langle formula\rangle - > \langle formula\rangle \\
 & ::= & \langle iteration\rangle \langle formula\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle iteration\rangle & ::= & \langle iterSymbol\rangle(\langle aVar\rangle = \\
 & & \langle arithmExpr\rangle..\langle arithmExpr\rangle) \\
\langle iterSymbol\rangle & ::= & BigAnd \\
 & ::= & BigOr \\
\langle arithmExprList\rangle & ::= & \langle arithmExpr\rangle \\
 & ::= & \langle arithmExpr\rangle, \langle arithmExprList\rangle \\
\langle arithmExpr\rangle & ::= & \langle aVar\rangle \\
 & ::= & \langle aConst\rangle \\
 & ::= & \langle aVar\rangle + \langle aConst\rangle \\
\langle rule\rangle & ::= & ax(\langle sequent\rangle) \\
 & ::= & pLink((\langle proof\_name\rangle, \langle index\rangle)\ \langle sequent\rangle) \\
 & ::= & negL(\langle id\rangle, \langle formula\rangle) \\
 & ::= & negR(\langle id\rangle, \langle formula\rangle) \\
 & ::= & andL1(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & andL2(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & andL(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & impR(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & impL(\langle id\rangle, \langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & andR(\langle id\rangle, \langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & orL(\langle id\rangle, \langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & orR1(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & contrL(\langle id\rangle, \langle id\rangle, \langle formula\rangle) \\
 & ::= & contrR(\langle id\rangle, \langle id\rangle, \langle formula\rangle) \\
 & ::= & orR2(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & orR(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & weakL(\langle id\rangle, \langle formula\rangle) \\
 & ::= & weakR(\langle id\rangle, \langle formula\rangle) \\
 & ::= & arrowL(\langle id\rangle, \langle formula\rangle) \\
 & ::= & arrowR(\langle id\rangle, \langle formula\rangle) \\
 & ::= & andEqL1(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & andEqR1(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & andEqL2(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
 & ::= & andEqR2(\langle id\rangle, \langle formula\rangle, \langle formula\rangle)
\end{array}
$$

$$
\begin{aligned}
::= \quad & andEqL3(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & andEqR3(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & orEqL1(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & orEqR1(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & orEqL2(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & orEqR2(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & orEqL3(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & orEqR3(\langle id\rangle, \langle formula\rangle, \langle formula\rangle) \\
::= \quad & autoprop(\langle id\rangle, \langle sequent\rangle)
\end{aligned}
$$

For better understanding of the calculus and the grammar described above, we illustrate it with simple example. Consider the following proof schema $\Psi = \langle(\psi(0), \psi(k+1))\rangle$ of a sequent $P_0, \bigwedge_{i=0}^{k}(\neg P_i \vee P_{i+1}) \vdash P_{k+1}$, where $\psi(0)$ is:

$$
\dfrac{\dfrac{\dfrac{P_0 \vdash P_0}{\neg P_0, P_0 \vdash}\neg\colon l \qquad P_1 \vdash P_1}{P_0, \neg P_0 \vee P_1 \vdash P_1}\vee\colon l}{P_0, \bigwedge_{i=0}^{0} \neg P_i \vee P_{i+1} \vdash P_1}\equiv\colon \wedge 3
$$

and $\psi(k+1)$ is:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\psi, k)}{P_0, \bigwedge_{i=0}^{k}(\neg P_i \vee P_{i+1}) \vdash P_{k+1}} \qquad \dfrac{\dfrac{P_{k+1} \vdash P_{k+1}}{\neg P_{k+1}, P_{k+1} \vdash}\neg\colon l \qquad P_{k+2} \vdash P_{k+2}}{P_{k+1}, \neg P_{k+1} \vee P_{k+2} \vdash P_{k+2}}\vee\colon l}{P_0, \bigwedge_{i=0}^{k}(\neg P_i \vee P_{i+1}), \neg P_{k+1} \vee P_{k+2} \vdash P_{k+2}}\,cut}{P_0, \bigwedge_{i=0}^{k}(\neg P_i \vee P_{i+1}) \wedge (\neg P_{k+1} \vee P_{k+2}) \vdash P_{k+2}}\wedge\colon l1, \wedge\colon l2, c\colon l}{P_0, \bigwedge_{i=0}^{k+1}(\neg P_i \vee P_{i+1}) \vdash P_{k+2}}\equiv\colon \wedge 1
$$

The sequence of inferences $\wedge\colon l_1, \wedge\colon l_1, c\colon l$ can be thought of as a single unary rule $\wedge\colon l$ which is not formally in the **LK** calculus. Then this proof can be written in our grammar in the following way:

```
proof   \psi proves P(0), BigAnd(i=0..k) (~ P(i) \/ P(i+1)) |- P(k+1)
base {
        1: ax(P(0) |- P(0))
        2: negL(1, P(0))
        3: ax(P(1) |- P(1))
        4: orL(2, 3, ~ P(0), P(1))
        root: andEqL3(4, (~ P(0) \/ P(1)), BigAnd(i=0..0) (~ P(i) \/ P(i+1)))
}
step {
        1: pLink((\psi, k) P(0), BigAnd(i=0..k) (~ P(i) \/ P(i+1)) |- P(k+1))
        2: ax(P(k+1) |- P(k+1))
        3: negL(2, P(k+1))
        4: ax(P(k+2) |- P(k+2))
        5: orL(3, 4, ~ P(k+1), P(k+2))
        6: cut(1, 5, P(k+1))
        7: andL(6, BigAnd(i=0..k) (~ P(i) \/ P(i+1)), (~ P(k+1) \/ P(k+2)))
        root: andEqL1(7, (BigAnd(i=0..k) (~ P(i) \/ P(i+1)) /\ (~ P(k+1) \/ P(k+2)))
              BigAnd(i=0..k+1) (~ P(i) \/ P(i+1)))
}
```

5.2 The Auto-propositional mode

The *auto-propositional* mode is an option integrated in the parser for *SHLK*. It is a complete algorithm for generating *propositional cut-free* **LK(S)**-proofs. As input the algorithm takes a sequent which is a propositional tautology. The output is a proof derivation of that sequent. The axioms of the leafs of the derivation in the current implementation is required to be atomic, i.e. sequents of the form $A \vdash A$, where A is either an atom or an indexed proposition. We make use of the well known fact that an **LK**-proof has the *subformula property*. Furthermore, taking into account that the derivation is cut-free, all of the formulas in the axiom appear in the end-sequent of the proof. Intuitively it works as follows : assume that the input is not an atomic sequent. Then at each step the algorithm chooses a non-atomic formula from the antecedent or succedent part of the sequent. Then, the algorithm is applied recursively to the obtained sequent(s) with the decomposed subformula and the duplication, if necessary, of the rest of the context of the sequent. The duplication is required in all cases of binary rules and in some cases of

unary rules. After each recursive call we apply the corresponding contraction inferences if before that there was a duplication. Decomposing the sequent in such a way will end up with a sequent S containing atoms only. Then we find the subsequent S' of S which is an atomic axiom and apply as many left or right weakening inferences as it is needed to derive the sequent S. Since during the decomposition of the formulas we duplicate the context of the current end-sequent in the upper subproof(s), the algorithm is complete. The duplications are removed by contractions, as it was already mentioned. Here we give an example (the auxiliary formulas of each inference are marked with red):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{A \vdash A, B}\, w:r}{A \vdash A, B, C}\, w:r}{\vdash A \to C, A, B}\, \to:r \quad \dfrac{\dfrac{A \vdash A}{A, C \vdash A}\, w:l}{A, C \vdash A, C}\, w:r}{C \vdash A \to C, A}\, \to:r}{B \to C \vdash A \to C, A, A \to C, A}\, \to:l}{B \to C \vdash A \to C, A, A \to C}\, c:r}{B \to C \vdash A \to C, A}\, c:r \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{B \vdash B}{B \vdash B, A \to C}\, w:r}{B, B, B \to C \vdash A \to C, A \to C}\, \to:l}{B, B \to C \vdash A \to C, A \to C}\, c:l}{B, B \to C \vdash A \to C}\, c:r \quad \dfrac{\dfrac{\dfrac{C \vdash C}{C, A \vdash C}\, w:l}{B, C, A \vdash C}\, w:l}{B, C \vdash A \to C}\, \to:r}{A \to B, B \to C, B \to C \vdash A \to C, A \to C}\, \to:l}{A \to B, B \to C, B \to C \vdash A \to C, A \to C}\, c:r}{A \to B, B \to C \vdash A \to C}\, c:l$$

The formulas A, B and C are arbitrary atoms. The derivation shows clearly how the ideas of decomposing a non-atomic formula and duplication of the context have been used. The proof derivation has 23 nodes. This fact shows that the integration of the auto-propositional mode in the parser for *SHLK* is of considerable help in typing or creating propositional derivations. In the case of a sequent with ground schematic formulas the procedure applies straightforwardly. In this case we just have first to decompose the formula in such a way : if we have $\bigwedge_{i=1}^{3} A_i$, then we apply the equivalence rules and get the formula $A_1 \wedge \bigwedge_{i=2}^{3} A_i$ and we proceed as in the example above. Let us show a few steps of the auto-propositional mode which takes as input the sequent:

$$S = A \to B,\ B \to C \vdash A \to C$$

and returns a proof derivation ρ of S.

We see that both antecedent and succedent part of S contain non-atomic formulas only. We start to decompose each one of them. The first one to be decomposed is $A \to B$ on the left side. It was introduced by a binary rule $\to_l$. Hence, we conclude that it is obtained after applying $\to_l$ to two subproofs ρ_1 and ρ_2 with the following end-sequents, respectively: $S_1 = B \to C \vdash A \to C,\ A$ and $S_2 = B,\ B \to C \vdash A \to C$. Applying $\to_l$ to ρ_1 and ρ_2 gives a proof ρ' with end-sequent:

$$S' = A \to B,\ B \to C,\ B \to C \vdash A \to C,\ A \to C$$

Applying to ρ' two contractions, namely $c\colon l$ and $c\colon r$, we obtain a proof with end-sequent S. The algorithm is applied recursively to ρ_1 and ρ_2.

Now we give a formal definition of the algorithm for auto-propositional proof generation in pseudocode:

formulas: A, B
formula multisets: Γ, Δ
sequent: S
proof variables: $\varphi, \varphi_1, \varphi_2, \varphi_3$

function: AutoProp
input: sequent //*propositional tautology*
output: arithmetically ground **SLK**-proof

1: **if** $S = \Gamma, A \vdash A, \Delta$ **then**
2: WeakeningsLeftRight(S)
3: **else**
4: **if** $S = A \to B, \Gamma \vdash \Delta$ **then**
5: $\varphi_1 :=$ AutoProp($\Gamma \vdash \Delta, A$)
6: $\varphi_2 :=$ AutoProp($B, \Gamma \vdash \Delta$)
7: $\varphi_3 :=$ ImpLeft($\varphi_1, \varphi_2, A, B$)
8: **return** ContractionsLeftRight($\varphi_3, \Gamma \vdash \Delta$)
9: **else**
10: **if** $S = \Gamma \vdash \Delta, A \wedge B$ **then**
11: $\varphi_1 :=$ AutoProp($\Gamma \vdash \Delta, A$)
12: $\varphi_2 :=$ AutoProp($\Gamma \vdash \Delta, B$)
13: $\varphi_3 :=$ AndRight($\varphi_1, \varphi_2, A, B$)
14: **return** ContractionsLeftRight($\varphi_3, \Gamma \vdash \Delta$)
15: **else**
16: **if** $S = A \vee B, \Gamma \vdash \Delta$ **then**
17: $\varphi_1 :=$ AutoProp($A, \Gamma \vdash \Delta$)
18: $\varphi_2 :=$ AutoProp($B, \Gamma \vdash \Delta$)
19: $\varphi_3 :=$ OrLeft($\varphi_1, \varphi_2, A, B$)
20: **return** ContractionsLeftRight($\varphi_3, \Gamma \vdash \Delta$)
21: **else**
22: **if** $S = \Gamma \vdash \Delta, A \to B$ **then**
23: $\varphi :=$ AutoProp($A, \Gamma \vdash \Delta, B$)
24: **return** ImpRight(φ, A, B)
25: **else**

26: **if** $S = \neg A, \Gamma \vdash \Delta$ **then**
27: $\varphi :=$ AutoProp$(\Gamma \vdash \Delta, A)$
28: **return** NegLeft(φ, A)
29: **else**
30: **if** $S = \Gamma \vdash \Delta, \neg A$ **then**
31: $\varphi :=$ AutoProp$(A, \Gamma \vdash \Delta)$
32: **return** NegRight(φ, A)
33: **else**
34: **if** $S = \Gamma \vdash \Delta, \bigvee_{i=a}^{b} A_i$ **then**
35: $\varphi :=$ AutoProp$(\Gamma \vdash \Delta, A_a \vee \bigvee_{i=a+1}^{b} A_i)$
36: **return** OrEquivalenceRight$(\varphi, \bigvee_{i=a}^{b} A_i)$
37: **else**
38: **if** $S = \bigwedge_{i=a}^{b} A_i, \Gamma \vdash \Delta$ **then**
39: $\varphi :=$ AutoProp$(A_a \wedge \bigwedge_{i=a+1}^{b} A_i, \Gamma \vdash \Delta)$
40: **return** AndEquivalenceLeft$(\varphi, \bigwedge_{i=a}^{b} A_i)$
41: **else**
42: **if** $S = A \wedge B, \Gamma \vdash \Delta$ **then**
43: $\varphi_1 :=$ AutoProp$(A, B, \Gamma \vdash \Delta)$
44: **return** AndLeft(φ_1, A, B)
45: **else**
46: **if** $S = \Gamma \vdash \Delta, A \vee B$ **then**
47: $\varphi_1 :=$ AutoProp$(\Gamma \vdash \Delta, A, B)$
48: **return** OrRight(φ_1, A, B)
49: **end if**
50: **end if**
51: **end if**
52: **end if**
53: **end if**
54: **end if**
55: **end if**
56: **end if**
57: **end if**
58: **end if**
59: **end if**

Remark: *The input of the* AutoProp *function should be an arithmetically ground sequent. Otherwise the unfolding of a schematic formula is not possible.*

The function WeakeningsLeftRight(S) from line 2 in the algorithm is trivial. It applies weakening rules with auxiliary formulas each formula in $\Gamma \vdash \Delta$.

For example, if $S = A, B, C \vdash A, D$, then WeakeningsLeftRight(S) gives the derivation:

$$\dfrac{\dfrac{A \vdash A}{A, B, C \vdash A}\, w\colon l^*}{A, B, C \vdash A, D}\, w\colon r$$

5.3 Propositional proof compression

The propositional proof which is obtained after applying the auto-propositional mode to a propositional tautology is usually not optimal. This happens, because some of the formulas in the duplicated context (for example in $\rightarrow_l$) may be introduced by weakening inferences. In this section we will describe an algorithm that removes those weakenings whose principal formulas are contracted downwards in the proof. Informally, the algorithm works as follows. The proof derivation is traversed bottom-up (i.e. from the root to the leafs). Let ρ be an inference rule. Then

- if $\rho \in \{c\colon l\ ,\ c\colon r\}$, then we check if both auxiliary formulas a_1 and a_2 are introduced by weakenings. If this is the case, then we remove one of these weakenings. Otherwise, if only a_1 is introduced by weakening, then remove the weakening which introduces it. Otherwise we apply ρ.

- if $\rho \in \{w\colon l\ ,\ w\colon r\ ,\ cut\ ,\ \vee\colon l\ ,\ \rightarrow\colon r\}$, then we apply again ρ.

- if $\rho \in \{\vee\colon l\ ,\ \wedge\colon r\ ,\ \rightarrow\colon l\}$ (binary rules), then we check whether some of the auxiliary formulas a_1 and a_2 is introduced by weakening. If this is a_1, then we remove the whole branch ρ_2. Then we remove the weakening which introduced a_1. Then we apply weakenings to the last proof to obtain the end-sequent of ρ. Otherwise, we apply ρ. (ρ_i is the subproof where a_i belongs, for $i = 1, 2$)

- if $\rho \in \{\neg\colon l\ ,\ \neg\colon r\ ,\ \wedge\colon l1\ ,\ \wedge\colon l2\ ,\ \vee\colon r1\ ,\ \vee\colon r2\ ,\ \forall\colon l\ ,\ \forall\colon r\ ,\ \exists\colon l\ ,\ \exists\colon r\}$, then we check if the auxiliary formula a is introduced by a weakening. If this is the case, then we remove the weakening rule which introduced a and then apply weakenings to obtain the end-sequent of ρ. Otherwise, we apply ρ.

Example: Consider the propositional proof in the previous section. The reader probably has already noticed that many formulas introduced by weakening are contracted later. Applying the idea which we just explained informally, we obtain the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{A \vdash A}{A \vdash A, C}\, w\colon r}{A \vdash A, A \to C}\, \to\colon r \qquad \dfrac{B \vdash B \qquad \dfrac{\dfrac{C \vdash C}{A, C \vdash C}\, w\colon l}{C \vdash A \to C}\, \to\colon r}{B, B \to C \vdash A \to C}\, \to\colon l}{A \to B, B \to C \vdash A \to C, A \to C}\, \to\colon l}{A \to B, B \to C \vdash A \to C}\, c\colon r$$

The proof above has 10 nodes which is more then twice less of the size of the one obtained by auto-propositional algorithm. This is not of major importance, because it does not shrink the size of the characteristic clause set. Nevertheless, it makes the extraction of the clause set a bit faster, because the proof derivation is smaller. Now we describe formally the algorithm in the pseudocode below:

auxiliary formulas: A, A_1, A_2
principal (main) formula: M
end-sequent: ES
inference rules: ρ, ρ_1, ρ_2
proof variables: $\varphi, \varphi_1, \varphi_2, \psi$
function: RemoveStructuralRules
input: inference rule in the proof
output: **SLK**-proof

```
1:  if (ρ ∈ {w: l, w: r}, ρ1, A) then
2:      p1 := RemoveStructuralRules(ρ1)
3:      apply(ρ, p1, A)
4:  else
5:      if (ρ ∈ {c: l, c: r}, ρ1, A1, A2) then
6:          φ1 := RemoveStructuralRules(ρ1)
7:          if A1 and A2 are weakened in φ1 then
8:              RemoveWeakeningOn(A2, φ1)
9:          else
10:             if A1 is weakened in φ1 then
11:                 RemoveWeakeningOn(A1, φ1)
12:             else
13:                 if A2 is weakened in φ2 then
```

```
14:                 RemoveWeakeningOn(A_2, φ_2)
15:             else
16:                 apply(ρ, ES(φ_1), A_1, A_2)
17:             end if
18:         end if
19:     end if
20: else
21:     if (ρ ∈ {∨: l , ∧: r , →: l}, ρ_1, ρ_2, A_1, A_2) then
22:         φ_1 := RemoveStructuralRules(ρ_1)
23:         φ_2 := RemoveStructuralRules(ρ_2)
24:         if A_1 is weakened in φ_1 then
25:             RemoveWeakeningOn(A_2, φ_1)
26:             AddWeakenings(φ_1, ES(ρ))
27:         else
28:             if A_2 is weakened in φ_2 then
29:                 RemoveWeakeningOn(A_2, φ_2)
30:                 AddWeakenings(φ_2, ES(ρ))
31:             else
32:                 apply(ρ, ES(φ_1), ES(φ_2), A_1, A_2)
33:             end if
34:         end if
35:     else
36:         if (ρ ∈ {¬: l , ∧: l1 , ∧: l2 , ∀: l , ∃: l}, ρ_1, A, M) then
37:             φ := RemoveStructuralRules(ρ_1)
38:             if A_1 is weakened in φ then
39:                 ψ := RemoveWeakeningOn(A_1, φ)
40:                 apply(w: l, ES(ψ), M)
41:             else
42:                 apply(ρ, ES(φ), A)
43:             end if
44:         else
45:             if (ρ ∈ {¬: r, ∨: r1, ∨: r2, ∀: r, ∃: r}, ρ_1, A, M) then
46:                 φ := RemoveStructuralRules(ρ_1)
47:                 if A_1 is weakened in φ then
48:                     ψ := RemoveWeakeningOn(A, φ)
49:                     apply(w: r, ES(ψ), M)
50:                 else
51:                     apply(ρ, ES(φ), A)
52:                 end if
53:             end if
54:         end if
```

55: **end if**
56: **end if**
57: **end if**

The function **RemoveWeakeningOn**(A, φ) removes the weakening which introduced the formula A. The function **apply**$(\rho, ES(\varphi), A)$ applies the unary inference ρ to the end-sequent of the proof φ with auxiliary formula A. The binary inference case is analogous. The function **AddWeakenings**$(\varphi, ES(\rho))$ applies weakenings to the proof φ so that the obtained proof has end-sequent like the end-sequent of ρ.

Remark: *The algorithm works on formulas, not on formula occurrences.*

The presented algorithm above removes all formulas which are introduced by weakening and contracted later on. In the example above still there are superfluous inferences. There is a contraction-right inference with auxiliary formula $A \to C$ which contains a subformula C which was introduced by weakening-right. Despite the fact that $A \to C$ is contracted, we can not remove the weakening rule which introduces C. In order to avoid such a situation we can make a better choice of the formula to be decomposed in the auto-propositional algorithm. Namely, we first have to apply all possible unary rules and then the binary rules. This will guarantee that the duplicated context will be weakened at some point. I.e. the formula which will be contracted is the one which which was introduced by weakening. Hence, the proof will be shorter. In our example this is even the shortest proof:

$$\dfrac{\dfrac{\dfrac{B \vdash B \qquad C \vdash C}{B \to C, B \vdash C} \to : l}{A \vdash A \qquad B \to C, A \to B, A \vdash C} \to : l}{B \to C, A \to B \vdash A \to C} \to : r$$

Chapter 6

Algorithms for proof schemata transformation

The automation of *CERES*$_s$ requires a development of specific algorithms and datastructures. For example, such an algorithm is the one for computing all relevant configurations in a proof schemata. These are all reachable from $\langle \psi_1, (|) \rangle$ configurations, where ψ_1 is the smallest proof symbol in a proof schema. By *cut-configuration* or just a *configuration* (see p.43) we mean a configuration (i.e. set of formula occurrences) which appears on the place of the proof-link when the proof schema is unfolded for a specific instance of the parameter. For the automated extraction of the schematic characteristic clause set and the schematic projections this is crucial because it considerably prunes the search space of all possible configurations. Furthermore, shrinking the size of the schematic characteristic clause set makes it easier to define a resolution proof which after instantiation refutes it. As we have already said, full automation of the refutation is not possible because of the undecidability of the unification problem for s-terms. Even if one restricts to some decidable class of s-terms, the problem still remains undecidable.

6.1 Computing relevant configurations

Let as consider the following example. Let $\Psi = \langle \psi, \varphi \rangle$ be a proof schema, where $\psi = (\pi_1, \nu_1(k+1))$ and $\varphi = (\pi_2, \nu_2(k+1))$ and $\psi < \varphi$. The base-case proofs are skipped, because they are irrelevant for the computation of the configurations.

- π_2:

$$\dfrac{\vdots}{\bigwedge_{i=0}^{0} A_i \vdash \bigwedge_{i=0}^{0} A_i}$$

- $\nu_2(k+1)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{(\varphi,k)}{\bigwedge_{i=0}^{k} A_i \vdash \bigwedge_{i=0}^{k} A_i} \quad A_{k+1} \vdash A_{k+1}}{\bigwedge_{i=0}^{k} A_i \,,\, A_{k+1} \vdash \bigwedge_{i=0}^{k} A_i \wedge A_{k+1}}\wedge : r}{\bigwedge_{i=0}^{k} A_i \wedge A_{k+1} \,,\, A_{k+1} \vdash \bigwedge_{i=0}^{k+1} A_i}\wedge : l1}{\bigwedge_{i=0}^{k} A_i \wedge A_{k+1} \,,\, \bigwedge_{i=0}^{k} A_i \wedge A_{k+1} \vdash \bigwedge_{i=0}^{k+1} A_i}\wedge : l2}{\bigwedge_{i=0}^{k+1} A_i \vdash \bigwedge_{i=0}^{k+1} A_i}c : l$$

- π_1:

$$\dfrac{\vdots}{A_0 \,,\, \bigwedge_{i=0}^{0} A_i \to A_{i+1} \vdash \bigwedge_{i=0}^{1} A_i}$$

- $\nu_1(k+1)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overset{(\psi,k)}{A_0 \,,\, \bigwedge_{i=0}^{k}(A_i \to A_{i+1}) \vdash \bigwedge_{i=0}^{k+1} A_i} \quad \dfrac{\dfrac{\dfrac{\dfrac{\overset{(\varphi,k+1)}{\bigwedge_{i=0}^{k+1} A_i \vdash \bigwedge_{i=0}^{k+1} A_i} \quad \dfrac{A_{k+1} \vdash A_{k+1} \quad A_{k+2} \vdash A_{k+2}}{A_{k+1} \,,\, A_{k+1} \to A_{k+2} \vdash A_{k+2}}\to : l}{\bigwedge_{i=0}^{k+1} A_i \,,\, A_{k+1} \,,\, A_{k+1} \to A_{k+2} \vdash \bigwedge_{i=0}^{k+1} A_i \wedge A_{k+2}}\wedge : r}{\bigwedge_{i=0}^{k+1} A_i \,,\, \bigwedge_{i=0}^{k} A_i \wedge A_{k+1} \,,\, A_{k+1} \to A_{k+2} \vdash \bigwedge_{i=0}^{k+2} A_i}\wedge : l2}{\bigwedge_{i=0}^{k+1} A_i \,,\, A_{k+1} \to A_{k+2} \vdash \bigwedge_{i=0}^{k+2} A_i}c : l}{A_0 \,,\, \bigwedge_{i=0}^{k} A_i \to A_{i+1} \,,\, A_{k+1} \to A_{k+2} \vdash \bigwedge_{i=0}^{k+2} A_i}cut}{A_0 \,,\, \bigwedge_{i=0}^{k} A_i \to A_{i+1} \,,\, \bigwedge_{i=0}^{k} A_i \to A_{i+1} \wedge A_{k+1} \to A_{k+2} \vdash \bigwedge_{i=0}^{k+2} A_i}\wedge : l1}{A_0 \,,\, \bigwedge_{i=0}^{k} A_i \to A_{i+1} \wedge \to A_{k+2} \to A_{i+1} \,,\, \bigwedge_{i=0}^{k} A_i \to A_{i+1} \wedge A_{k+1} \to A_{k+2} \vdash \bigwedge_{i=0}^{k+2} A_i}\wedge : l2}{A_0 \,,\, \bigwedge_{i=0}^{k+1} A_i \to A_{i+1} \vdash \bigwedge_{i=0}^{k+2} A_i}c : l$$

The formulas in red color are the (local) cut-ancestors in the step-case of ψ. We see that the proof-links (ψ, k) and (φ, k) also contain cut-ancestors. That means that, if the proof is unfolded for a given instance of the parameter, the ancestors of these red formulas in the upper subderivation will be propagated also as cut-ancestors. For the proof-link (φ, k) these are the formulas in magenta color in $\nu_2(k+1)$. The configuration in this case is $\Omega_1 = (\bigwedge_{i=0}^{k+1} A_i \mid)$. Analogously, the proof-link (ψ, k) we have the configuration $\Omega_2 = (\mid \bigwedge_{i=0}^{k+1} A_i)$ and the propagated ancestors are the formulas marked with blue color in $\nu_1(k+1)$. Such cut-ancestors will be called global-

or Ω-ancestors, because they are cut-ancestors relative to $\nu_1(k+2)$, but not in $\nu_1(k+1)$. Having the local cut-ancestors (in red) and the global cut-ancestors (in blue) in $\nu_1(k+1)$, we see that appears a third configuration for the proof-link (φ, k), namely $\Omega_3 = (\bigwedge_{i=0}^{k+1} A_i \mid \bigwedge_{i=0}^{k+1} A_i)$. We have also a fourth configuration which is the trivial one, namely $\Omega_4 = (\ \mid\ )$ for $\nu_1(k+1)$. In summary, the relevant configurations are in a certain order, i.e. one configuration may generate a new one. They are computed iteratively. The maximum number of the iterations is bounded by the length of the longest end-sequent of a step-case proof in Ψ. Here is a formal description of the algorithm in pseudocode:

Let $\Psi = \langle \psi_1, \psi_2, \ldots, \psi_\alpha \rangle$ be a proof schema, where $\alpha \in \mathbb{N}$. The function below extracts the (relevant) configurations in Ψ. Its input is a set of pairs $\langle \psi_i, \Omega \rangle$, where Ω is a configuration for the end-sequent of the step case for ψ_i and $i \in \{1, \ldots, \alpha\}$.

sets of pairs $\langle$**proof-symbol, configuration**$\rangle$:
setOfPairs, tmpSetOfPairs, ResultSet
step-case proof for a proof symbol φ: φ_{step}
arithmetical term: a
configurations: Ω, Ω_φ
initial: setOfPairs := $\{\langle \psi_1, (\ \mid\ ) \rangle\}$, ResultSet := $\emptyset$

```
1: function ExtractRelevantCC(setOfPairs, ResultSet)
2:     for all pairs ⟨φ, Ω_φ⟩ ∈ setOfPairs do
3:         tmpSetOfPairs := ∅
4:         cutAncSet := getCutAncestors(φ_step)
5:         omegaAncSet := getOmegaAncestors(Ω_φ)
6:         ancSet := cutAncSet ∪ omegaAncSet
7:         proofLinksSet := getProofLinksIn(φ_step)
8:         for all proof-links (ψ, a) ∈ proofLinksSet do
9:             Ω := getCC((ψ, a), ancSet)
10:            if ⟨ψ, Ω⟩ ∉ tmpSetOfPairs ∪ ResultSet then
11:                tmpSetOfPairs := tmpSetOfPairs ∪ {⟨ψ, Ω⟩}
12:            end if
13:        end for
14:        ResultSet := ResultSet ∪ {⟨φ, Ω_φ⟩}
15:        setOfPairs := setOfPairs − {⟨φ, Ω_φ⟩}
16:        setOfPairs := setOfPairs ∪ tmpSetOfPairs
```

17: **end for**
18: **return** ResultSet
19: **end function**

The iterative function **ExtractRelevantCC** works as follows. We start with the set of pairs $\{\langle\psi_1, (\ |\ )\rangle\}$ and we want to compute all reachable from it configurations. For each *pair* $\in$ *setOfPairs* we extract all reachable configurations and store the new of them in the set *tmpSetOfPairs*. That means that in *tmpSetOfPairs* are added only those pairs which appear for first time. Doing so we avoid redundancy, i.e. we do not recompute already computed configurations. Then we remove *pair* from the set *setOfPairs* and add it to the set *ResultSet*. The computed from *pair* reachable configurations we add to the set *setOfPairs*. We iterate this steps until *setOfPairs* is not empty, i.e. until no further reachable configurations is possible to be computed. When the set *setOfPairs* is empty the iteration stops and we return the set *ResultSet* which stores all reachable from $\langle\psi_1, (\ |\ )\rangle$ configurations.

The function **getCutAncestors**(φ) returns the set of all cut-ancestors in a proof φ, i.e. $\{f \mid f$ is a cut-ancestor in $\varphi\}$. The function **getOmegaAncestors**(Ω_φ) returns the set of all Ω_φ-ancestors. This set is obtained in the following way : we take the union of the sets of all ancestors of a given formula occurrence in Ω_φ in the proof φ, i.e. $\bigcup_{A\in\Omega_\varphi}\{B \mid B$ is ancestor of A in $\varphi\}$. The function **getCC**$((\psi, a), S)$ computes the configuration for the a-th isntance of a proof-symbol ψ with respect to the set S of formula occurrences, where a is an arithmetic term, i.e. only the set of those formulas in the proof-link will be returned which are members of S.

Example: Let us apply the algorithm above to extract the relevant configurations of the proof schema $\Psi = \langle\psi, \varphi\rangle$ defined above. Let S^i_j be the set of pairs which the algorithm returns at round i, iteration j. In the first round **ExtractRelevantCC**$(\psi, (\ |\ ))$ returns the set of pairs $S^1_1 = \{\langle\psi, (\ |\ )\rangle\ ;\ \langle\psi, (\ |\ \bigwedge_{i=0}^{k+1} A_i)\rangle\ ;\ \langle\varphi, (\ \bigwedge_{i=0}^{k+1} A_i\ |\ )\rangle\}$. Since $|S^1_1| = 3$ we have 3 iterations in the second round. The first iteration gives $S^2_1 = S^1_1$. The second iteration gives $S^2_2 =$ **ExtractRelevantCC**$(\langle\psi, (\ |\ \bigwedge_{i=0}^{k+1} A_i)\rangle) = \{\langle\psi, (\ |\ \bigwedge_{i=0}^{k+1} A_i)\rangle\ ;\ \langle\varphi, (\bigwedge_{i=0}^{k+1} A_i\ |\ \bigwedge_{i=0}^{k+1} A_i)\rangle\}$. We see that a new configuration for φ was generated, namely $(\bigwedge_{i=0}^{k+1} A_i\ |\ \bigwedge_{i=0}^{k+1} A_i)$. The third iteration gives $S^2_3 =$ **ExtractRelevantCC**$(\langle\varphi, (\ \bigwedge_{i=0}^{k+1} A_i\ |\ )\rangle = \{\langle\varphi, (\ \bigwedge_{i=0}^{k+1} A_i\ |\ )\rangle\}$. Hence, the second round of the algorithm gives the set $S^2 = S^2_1 \cup S^2_2 \cup S^2_3 =$

$= \{\langle\psi, (\ |\ )\rangle\ ;\ \langle\psi, (\ |\ \bigwedge_{i=0}^{k+1} A_i)\rangle\ ;\ \langle\varphi, (\ \bigwedge_{i=0}^{k+1} A_i\ |\ )\rangle\ ;\ \langle\varphi, (\bigwedge_{i=0}^{k+1} A_i\ |\ \bigwedge_{i=0}^{k+1} A_i)\rangle\}$. The third round of the algorithm does not generate new cut configurations, because the configuration $(\bigwedge_{i=0}^{k+1} A_i\ |\ \bigwedge_{i=0}^{k+1} A_i)$ in φ generates itselve. Hence, $S^3 = S^2$. Therefore, S^2 is the fixed point of the computation, i.e. the set of all relevant configurations for the proof schema Ψ.

Remark: *The worst case complexity of the algorithm is the case where all configurations are configurations. This gives us a set of configurations of length* $2^{|ES(\nu_1(k+1))|} + \ldots + 2^{|ES(\nu_\alpha(k+1))|}$, *where* $\Psi = (\psi_1, \ldots, \psi_\alpha)$ *is the proof schema,* $\nu_i(k+1)$ *is the step case proof for* ψ_i *and* $|ES(\nu_i(k+1))|$ *is the size of the end-sequent of* $\nu_i(k+1)$, *for* $i = 1, \ldots, \alpha$.

6.2 Extracting the characteristic term

The notion of a *characteristic term* is similar to the notion of *struct* which was introduced in [WP09] for the purposes related with the analysis of the cut-elimination. It is a term of atomic formulas over $\oplus$ and $\otimes$ and which is used as a compact way to store the cut-pertinent [WP09] information of an **LK** proof. The characteristic clause-term which we defined in chapter 3 is similar to the struct, but not identical. Whereas in an **LK**-proof the set of all cut-ancestors is always known, this is not the case for an **LKS**-proof. As it was shown in chapter 3, we encode the cut- and Ω-ancestors in the proof-links with special symbols $cl^{\varphi,\Omega}$, where φ is a proof, Ω is a configuration and $\vec{x}_i : \iota$ (where $x_i : \iota$) is a vector of the free variables is ψ. Here we present an algorithm in pseudocode for extracting the schematic struct, where ρ is an inference in the step-case proof of ψ and Ω is a configuration. When it is necessary we will indicate the auxiliary formula(s) of ρ with a subscript.

terms over $\oplus$ **and** $\otimes$: t_1, t_2
(sub)proofs : ρ_1, ρ_2
configuration : Ω

1: **function** ExtractStruct(ψ, ρ, Ω)
2: **if** ρ is an axiom $A \vdash A$ in ψ **then**
3: **if** A and A are cut- or Ω-ancestor in ψ **then**
4: $[A \vdash A]$
5: **else**
6: **if** A is a cut- or Ω-ancestor in ψ **then**

```
7:                return [A ⊢ ]
8:            else
9:                if A is a cut- or Ω-ancestor in ψ then
10:                   return [ ⊢ A]
11:               else
12:                   ⊢
13:               end if
14:           end if
15:       end if
16:   else
17:       if ρ is a binary rule in ψ with auxiliary formulas
              A_1 and A_2 and subproofs ρ_1 and ρ_2, respectively then
18:           if A_1 and A_2 are cut- or Ω-ancestors in ψ then
19:               return ExtractStruct(ψ, ρ_1, Ω) ⊕ExtractStruct(ψ, ρ_2, Ω)
20:           else
21:               return ExtractStruct(ψ, ρ_1, Ω) ⊗ExtractStruct(ψ, ρ_2, Ω)
22:           end if
23:       else
24:           if ρ is a unary rule in ψ with subproof ρ_1 then
25:               return ExtractStruct(ψ, ρ_1, Ω)
26:           else
27:               if ρ is a proof-link (φ, a) in ψ with end-sequent Γ ⊢ Δ then
28:                   aSet := {A | A ∈ Γ and A is a cut- or Ω-ancestor in ψ}
29:                   sSet := {B | B ∈ Δ and B is a cut- or Ω-ancestor in ψ}
30:                   Ω' := (aSet | sSet)
31:                   return cl^{φ,Ω'}_{a,x_1,...,x_α}
32:               end if
33:           end if
34:       end if
35:   end if
36: end function
```

The algorithm above extracts the schematic struct for a given **LKS**-proof ψ. In order to obtain the struct for the whole proof schemata Ψ, we have to apply the algorithm on all pairs for $\psi \in \Psi$. Here is the point where the set of relevant configurations plays a role. The struct for Ψ is a finite iteration over the proof-symbols $\psi \in \Psi$ and the **corresponding** to each ψ set S_ψ of relevant configurations:

$$\bigoplus_{\psi_i \in \Psi} \bigoplus_{\Omega \in S_{\psi_i}} ((cl^{(\psi_i,\Omega)}_{0,x_1,\ldots,x_\alpha} \vdash \ \otimes \mathbf{ExtractStruct}(\pi_i, \rho_0^{\pi_i}, \Omega))$$

$$\oplus$$

$$(cl^{(\psi_i,\Omega)}_{k+1,x_1,\ldots,x_\alpha} \vdash \ \otimes \ \mathbf{ExtractStruct}(\nu_i(k+1), \rho_0^{\nu_i}, \Omega)))$$

where $\rho_0^{\pi_i}$ and $\rho_0^{\nu_i}$ are respectively the root-inferences of the base- and step-case proofs corresponding to the proof symbol ψ_i, for $i = 1, \ldots, \alpha$ and $S = \bigcup_{i=1}^{\alpha} S_{\psi_i}$ is the set of all relevant configurations computed in the previous section. The resulting term is similar to the one which we have already seen in chapter 3 where we extracted the schematic clause-set term which after instantiation of the parameter and rewriting of the symbols $cl_a^{(\varphi,\Omega)}$ followed by *clausification* [WP08], i.e. distributing $\oplus$ over $\otimes$, is transformed to the clause set. A detailed example of the extraction of the characteristic term is given in Chapter Experiments.

6.3 Computing the characteristic projection term

The computation of the projection term is similar in a sense to the computation of the characteristic clause-set term. In both cases we use a special symbols in the terms denoting the proof-links. In the case of proof projections this symbol $pr^{(\varphi,\Omega)}$ will denote the set of proofs corresponding to the proof-symbol φ and configuration Ω.

Nevertheless, there are some differences between the computation of the two terms. The first one is that the projection term rewrites to a projection set-schema which represents a set of proofs. The second one is that we take care of the auxiliary formulas because we need them when we unfold the term and construct the proof.

Here is a pseudocode of the algorithm. Here with ρ_1 (ρ_1, ρ_2) we denote the immediate upper inference(s) of ρ. We build the projection terms over the set:
$\{ \oplus\ ,\ \otimes_\rho\ ,\ c\colon l\ ,\ c\colon r\ ,\ w^{\Gamma\vdash\Delta}\ ,\ \wedge\colon l1\ ,\ \wedge\colon l2\ ,\ \wedge\colon r\ ,\ \vee\colon l\ ,\ \vee\colon r1\ ,\ \vee\colon r2\ ,$
$\rightarrow\colon l\ ,\ \rightarrow\colon r\ ,\ \neg\colon l\ ,\ \neg\colon r\ ,\ \forall\colon l\ ,\ \forall\colon r\ ,\ \exists\colon l\ ,\ \exists\colon r\ \}$ as follows:

1: **function** ProjectionTerm(ψ, ρ, Ω)
2: **if** ρ is an axiom $S = A \vdash A$ **then**
3: **return** S
4: **else**
5: **if** $\rho \in \{\neg\colon l\ ,\ \neg\colon r\ ,\ \wedge\colon l1\ ,\ \wedge\colon l2\ ,\ \vee\colon r1\ ,\ \forall\colon l\ ,\ \forall\colon r\ ,$
$\vee\colon r2\ ,\ w\colon l\ ,\ w\colon r\ ,\ c\colon l\ ,\ c\colon r\ ,\ \exists\colon l\ ,\ \exists\colon r\}$ **then**
6: $t_1 :=$ ProjectionTerm(ψ, ρ_1, Ω)
7: **if** auxiliary formula A is cut- or Ω-ancestor **then**
8: **return** t_1
9: **else**
10: **return** $\rho_A(t_1)$)
11: **end if**
12: **else**
13: **if** $\rho \in \{\wedge\colon r\ ,\ \vee\colon l\ ,\ \rightarrow\colon l\ ,\ cut\}$ **then**
14: $t_1 :=$ ProjectionTerm(ψ, ρ_1, Ω)
15: $t_2 :=$ ProjectionTerm(ψ, ρ_2, Ω)
16: **if** aux. f-las A_1, A_2 are cut- or Ω-ancestors **then**
17: $S_1 :=$ RemoveCutOmegaAnc($ES(t_2)$)
18: $S_2 :=$ RemoveCutOmegaAnc($ES(t_1)$)
19: **return** $w^{S_2}(t_1) \oplus w^{S_1}(t_2)$
20: **else**
21: **return** $t_1 \otimes_{\rho_{A_1,A_2}} t_2$
22: **end if**
23: **else**
24: **if** ρ is a proof-link (φ, a) with end-sequent $\Gamma \vdash \Delta$ **then**
25: antS:= $\{A | A \in \Gamma$ and A is cut- or Ω-ancestor in $\psi\}$
26: sucS:= $\{B | B \in \Delta$ and B is cut- or Ω-ancestor in $\psi\}$
27: $\Omega' :=$ (antS | sucS)
28: **return** $pr^{\varphi,\Omega'}_{a,x_1,\ldots,x_\alpha}$
29: **end if**
30: **end if**
31: **end if**
32: **end if**
33: **end function**

In the algorithm above the immediate subproofs of ρ are denoted with ρ_1 and ρ_1, ρ_2 for the case of unary and binary inference, respectively. The auxiliary formulas (more precisely formula occurrences) of ρ are denoted with A and A_1, A_2 for the case of unary and binary inference, respectively. They are in-

dicated as a subscript of ρ. The function **RemoveCutOmegaAnc**$(ES(t))$ returns a subsequent of the end-sequent of the proof t which does not contain cut- and Ω-ancestors. After instantiating the parameter and applying the rewrite rules for the symbol $pr^{\varphi,\Omega'}$ to the computed term followed by distributing $\oplus$ over $\otimes$, we get a ground term which represents the projection-set schema. Straightforwardly, we can construct the set of projections which in fact is a set of **LK**-proofs. A detailed example of the extraction of the projection term is given in the next chapter.

Chapter 7

Experiments

In this chapter we will make an experiment which compares the two transformations *CERES* and $CERES_s$. In the graphic below the dashed branch performs an instantiation of a proof schemata of **LKS**-proofs, unfolds the proofs for ψ_1 and apply the *CERES* method to the obtained **LK**-proof. On other hand, the solid-line branch performs the $CERES_s$ method, namely we compute the schematic characteristic clause-set term and the characteristic projection term. Then we construct a resolution schema. Afterwards, for a given instance of the parameter, we compute the ground clause set and projections as well as a ground resolution refutation. Finally, we construct the *ACNF*. We expect that the two *ACNF*s prove the same statement for every instance of the parameter.

7.1 The schematic EXP Proof

Let $\Psi = (\psi_1, \psi_2)$, where $\langle \pi_i, \nu_i(k+1) \rangle$ is a pair of the base- and step-case proofs for ψ_i, where $i = 1, 2$. We want to proof the following statement:

$$P(a), \forall x(P(x) \to P(f(x))) \vdash P(f(g(k+1, a)))$$

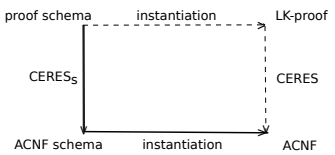


Figure 7.1: Alternative approaches

where the rewriting system for $g \in \mathcal{F}_a^2$ (where $z\colon \iota$) is:

$$\begin{array}{rcl} g(\bar{0}, z) & \to & z \\ g(k+1, z) & \to & f(g(k, z)) \end{array}$$

For example, for $\alpha \in \mathbb{N}$ we have $g(\alpha, z) = f^\alpha(z)$.

[1] π_1 is:

$$\dfrac{\dfrac{P(a) \vdash P(a) \qquad \dfrac{P(f(g(\bar{0},a))) \vdash P(f(g(\bar{0},a)))}{P(f(a)) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle \twoheadrightarrow}}{P(a), P(a) \to P(f(a)) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle \to\colon l}}{P(a), \forall x(P(x) \to P(f(x))) \vdash P(f(g(\bar{0},a)))}\,{\scriptstyle \forall\colon l}$$

$\nu_1(k+1)$ is:

$$\dfrac{\dfrac{\overline{\forall x(\overline{P}(x) \to \overline{P}(f(x))) \vdash \forall x(\overline{P}(x) \to \overline{P}(f(g(\overline{k+1}, x))))}^{(\psi_2, k+1)} \qquad \dfrac{\dfrac{P(a) \vdash P(a) \qquad P(f(g(k+1,a))) \vdash P(f(g(k+1,a)))}{P(a), P(a) \to P(f(g(k+1,a))) \vdash P(f(g(k+1,a)))}\,{\scriptstyle \to\colon l}}{P(a), \forall x(P(x) \to P(f(g(k+1,x)))) \vdash P(f(g(k+1,a)))}\,{\scriptstyle \forall\colon l}}{P(a), \forall x(P(x) \to P(f(x))) \vdash P(f(g(k+1,a)))}\,{\scriptstyle cut}$$

π_2 is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(z_0) \vdash P(z_0) \qquad \dfrac{P(f(g(\bar{0},z_0))) \vdash P(f(g(\bar{0},z_0)))}{P(f(z_0)) \vdash P(f(g(\bar{0},z_0)))}\,{\scriptstyle \twoheadrightarrow}}{P(z_0), P(z_0) \to P(f(z_0)) \vdash P(f(g(\bar{0},z_0)))}\,{\scriptstyle \to\colon l}}{P(z_0) \to P(f(z_0)) \vdash P(z_0) \to P(f(g(\bar{0},z_0)))}\,{\scriptstyle \to\colon r}}{\forall x(P(x) \to P(f(x))) \vdash P(z_0) \to P(f(g(\bar{0},z_0)))}\,{\scriptstyle \forall\colon l}}{\forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(f(g(\bar{0},x))))}\,{\scriptstyle \forall\colon r}$$

The proof χ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(z_{k+1}) \vdash P(z_{k+1}) \qquad \dfrac{\dfrac{P(g(k+1,z_{k+1})) \vdash P(g(k+1,z_{k+1}))}{P(f(g(k,z_{k+1}))) \vdash P(g(k+1,z_{k+1}))}\,{\scriptstyle \twoheadrightarrow} \qquad P(f(g(k+1,z_{k+1}))) \vdash P(f(g(k+1,z_{k+1})))}{P(f(g(k,z_{k+1}))), P(g(k+1,z_{k+1})) \to P(f(g(k+1,z_{k+1}))) \vdash P(f(g(k+1,z_{k+1})))}\,{\scriptstyle \to\colon l}}{P(z_{k+1}), P(g(k+1,z_{k+1})) \to P(f(g(k+1,z_{k+1}))), P(z_{k+1}) \to P(f(g(k,z_{k+1}))) \vdash P(f(g(k+1,z_{k+1})))}\,{\scriptstyle \to\colon l}}{P(g(k+1,z_{k+1})) \to P(f(g(k+1,z_{k+1}))), P(z_{k+1}) \to P(f(g(k,z_{k+1}))) \vdash P(z_{k+1}) \to P(f(g(k+1,z_{k+1})))}\,{\scriptstyle \to\colon r}}{P(z_{k+1}) \to P(f(g(k,z_{k+1}))), \forall x(P(x) \to P(f(x))) \vdash P(z_{k+1}) \to P(f(g(k+1,z_{k+1})))}\,{\scriptstyle \forall\colon l}}{\forall x(P(x) \to P(f(x))), \forall x(P(x) \to P(f(g(k,x)))) \vdash P(z_{k+1}) \to P(f(g(k+1,z_{k+1})))}\,{\scriptstyle \forall\colon l}$$

$\nu_2(k+1)$ is:

[1] The cut- and Ω-ancestors are marked with red and blue, respectively.

$$\dfrac{\dfrac{\dfrac{}{\forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k,x))))}\,(\psi_2,k) \qquad \dfrac{\chi}{\forall x(P(x)\to P(f(x))),\forall x(P(x)\to P(f(g(k,x))))\vdash \forall x(P(x)\to P(f(g(k+1,x))))}\,\forall:r}{\forall x(P(x)\to P(f(x))),\forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k+1,x))))}\,cut}{\forall x(P(x)\to P(f(x)))\vdash \forall x(P(x)\to P(f(g(k+1,x))))}\,c:l$$

We fix an instance of the parameter $k = 1$. First we apply the usual *CERES* method. We compute $\Psi\downarrow_1$ and we obtain an **LK**-proof $\nu_1(1)$, where:

ϕ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{P(z_0) \vdash P(z_0) \qquad P(f(z_0)) \vdash P(f(z_0))}{P(z_0), P(z_0) \to P(f(z_0)) \vdash P(f(z_0))} \to : l}{P(z_0) \to P(f(z_0)) \vdash P(z_0) \to P(f(z_0))} \to : r}{\forall x(P(x) \to P(f(x))) \vdash P(z_0) \to P(f(z_0))} \forall : l}{\forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(f(x)))} \forall : r$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(f(z_1)) \vdash P(f(z_1)) \qquad P(f^2(z_1)) \vdash P(f^2(z_1))}{P(z_1) \vdash P(z_1) \qquad P(f(z_1)), P(f(z_1)) \to P(f^2(z_1)) \vdash P(f^2(z_1))} \to : l}{P(z_1), P(f(z_1)) \to P(f^2(z_1)), P(z_1) \to P(f(z_1)) \vdash P(f^2(z_1))} \to : r}{P(f(z_1)) \to P(f^2(z_1)), P(z_1) \to P(f(z_1)) \vdash P(z_1) \to P(f^2(z_1))} \forall : l}{P(z_1) \to P(f(z_1)), \forall x(P(x) \to P(f(x))) \vdash P(z_1) \to P(f^2(z_1))} \forall : l}{\forall x(P(x) \to P(f(x))), \forall x(P(x) \to P(f(x))) \vdash P(z_1) \to P(f^2(z_1))} \forall : r}{\forall x(P(x) \to P(f(x))), \forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(f^2(x)))} cut$$

$$\forall x(P(x) \to P(f(x))) \ , \ \forall x(P(x) \to P(f(x))) \ \vdash \ \forall x(P(x) \to P(f^2(x)))$$

$\nu_1(1)$ is:

$$\dfrac{\dfrac{\phi}{\forall x(P(x) \to P(f(x))) \vdash \forall x(P(x) \to P(f^2(x)))} c : l \qquad \dfrac{\dfrac{P(a) \vdash P(a) \qquad P(f^2(a)) \vdash P(f^2(a))}{P(a), P(a) \to P(f^2(a)) \vdash P(f^2(a))} \to : l}{P(a), \forall x(P(x) \to P(f^2(x))) \vdash P(f^2(a))} \forall : l}{P(a) \ , \ \forall x(P(x) \to P(f(x))) \ \vdash \ P(f^2(a))} cut$$

The clause set $CL(\nu_1(1))$ after tautology elimination is:

$$\{ \ P(z_0) \vdash P(f(z_0)) \ ; \ P(f(z_1)) \vdash P(f(f(z_1))) \ ; \ \vdash P(a) \ ; \ P(f^2(a)) \vdash \ \}$$

which after subsumption elimination becomes:

$$\{ \ P(z_0) \vdash P(f(z_0)) \ ; \ \vdash P(a) \ ; \ P(f^2(a)) \vdash \ \}$$

The resolution refutation $\mathcal{R}$ is:

$$\dfrac{\dfrac{\dfrac{\vdash P(a) \qquad (P(z_0) \vdash P(f(z_0)))\{z_0 \leftarrow a\}}{\vdash P(f(a))} cut \qquad (P(z_0) \vdash P(f(z_0)))\{z_0 \leftarrow f(a)\}}{\vdash P(f(f(a)))} cut \qquad P(f(f(a))) \vdash}{\vdash} cut$$

The projection $PR(\ P(z_0) \vdash P(f(z_0))\ )$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(z_0) \vdash P(z_0) \qquad P(f(z_0)) \vdash P(f(z_0))}{P(z_0) \ , \ P(z_0) \to P(f(z_0)) \vdash P(f(z_0))} \to : l}{P(z_0) \ , \ \forall x(P(x) \to P(f(x))) \vdash P(f(z_0))} \forall : l}{P(z_0) \ , \ \forall x(P(x) \to P(f(x))) \ , \ \forall x(P(x) \to P(f(x))) \vdash P(f(z_0))} w : l}{P(z_0) \ , \ \forall x(P(x) \to P(f(x))) \vdash P(f(z_0))} c : l}{P(a) \ , \ P(z_0) \ , \ \forall x(P(x) \to P(f(x))) \ \vdash \ P(f(z_0)) \ , \ P(f^2(a))} w : l$$

The projection $PR(\ \vdash P(a)\ )$ is:

$$\dfrac{\dfrac{P(a) \vdash P(a)}{P(a) \ \vdash \ P(a)\ ,\ P(f^2(a))}\ w\colon r}{P(a)\ ,\ \forall x(P(x) \to P(f(x))) \ \vdash \ P(a)\ ,\ P(f^2(a))}\ w\colon l$$

The projection $PR(\ P(f^2(a)) \vdash\ )$ is:

$$\dfrac{\dfrac{P(f^2(a)) \vdash P(f^2(a))}{P(a) \ \vdash \ P(f^2(a))\ ,\ P(f^2(a))}\ w\colon l}{P(a)\ ,\ \forall x(P(x) \to P(f(x)))\ ,\ P(f^2(a)) \ \vdash \ P(f^2(a))}\ w\colon l$$

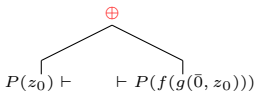
Finally, the *ACNF* is:

$$\dfrac{\dfrac{\dfrac{PR(\ \vdash P(a)\ )}{\Gamma \vdash \Delta, P(a)} \quad \dfrac{PR(P(z_0) \vdash P(f(z_0)))\{z_0 \leftarrow a\}}{P(a), \Gamma \vdash \Delta, P(f(a))}}{\Gamma \vdash \Delta, P(f(a))}\ cut, c\colon l, r \quad \dfrac{PR(P(z_0) \vdash P(f(z_0)))\{z_0 \leftarrow f(a)\}}{P(f(a)), \Gamma \vdash \Delta, P(f^2(a))}}{\Gamma \vdash \Delta, P(f(f(a)))}\ cut, c\colon l, r \quad \dfrac{PR(\ P(f^2(a)) \vdash\ )}{P(f^2(a)), \Gamma \vdash \Delta}}{\Gamma \vdash \Delta}\ cut$$

where $\Gamma = P(a),\ \forall x(P(x) \to P(f(x)))$ and $\Delta = P(f^2(a))$. This completes the dashed branch of Figure 7.1

Now we apply to Ψ the transformation $CERES_s$, i.e. the solid branch of the Figure 7.1. We start with the extraction of the characteristic term for each of the **LKS**-proofs in Ψ:

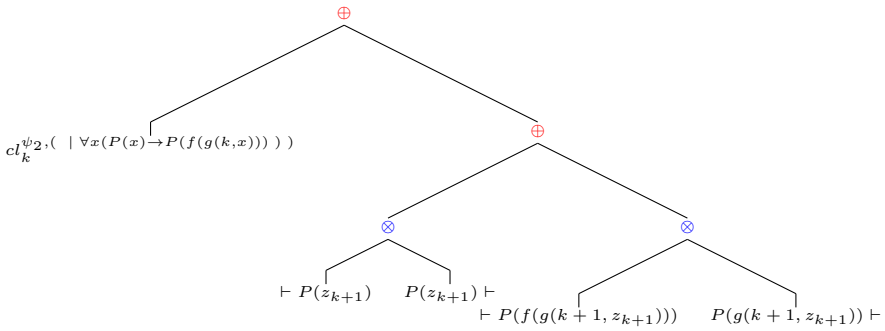
- $\Theta(\ \pi_2, (\ \ \mid \forall x(P(x) \to P(f(g(\bar{0}, z_0))))\ )\ )$:



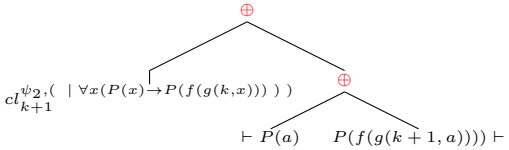
- $\Theta(\ \pi_1, (\ \mid\ )\ )$:

$$\vdash$$

- $\Theta(\ \nu_2(k+1), (\ \ \mid \forall x(P(x) \to P(f(g(k, x))))\ )\ )$:



- $\Theta(\ \nu_1(k+1), (\ |\ )\ )$:



Combining the terms above we obtain a characteristic clause-term which after rewriting gives us the standard characteristic clause set $\mathrm{CL}(\Psi)$, where $\mathcal{C}$ and $\mathcal{D}$ denote the clause sets for $cl^{\psi_2,(\ |\ \forall x(P(x)\to P(f(g(k,x))))\ )}$ and $cl^{\psi_1,(\ |\ )}$, respectively:

$$\Big\{\mathcal{D}(0) \to \{\vdash\}\ ;$$
$$\mathcal{D}(k+1) \to \mathcal{C}(k+1)\ \circ\ \{\vdash P(a)\ ;\ P(f(g(k+1,a))) \vdash\}\ ;$$
$$\mathcal{C}(0) \to \{\ P(z_0) \vdash P(f(g(\bar{0}, z_0)))\ \}\ ;$$
$$\mathcal{C}(k+1)\ \to\ \mathcal{C}(k)\ \circ\ \{\ P(g(k+1, z_{k+1})) \vdash P(f(g(k+1, z_{k+1})))\ \}\ \Big\}$$

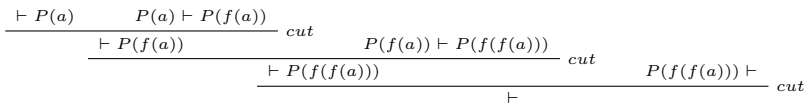
For the instance $k = 1$ we apply the rewriting rules for the symbols cl and we obtain (after tautology elimination) the characteristic clause set $CL(\Psi)\downarrow_1$:

$$\{P(z_0) \vdash P(f(z_0))\ ;\ P(f(z_1)) \vdash P(f(f(z_1)))\ ;\ \vdash P(a)\ ;\ P(f(f(z_1))) \vdash\}$$

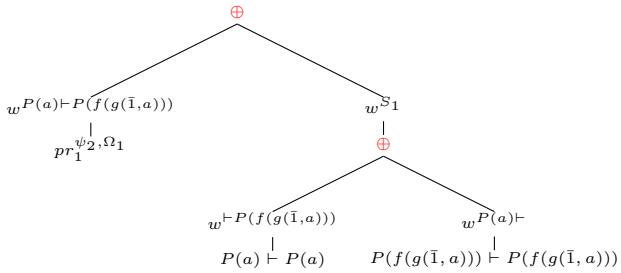
Let $R = ((\varrho, \delta), \mathcal{R})$, where $\mathcal{R}$ is the following rewriting system (constructed by hand, not automatically generated):

$$\begin{aligned}
\varrho(\bar{0}, x) &\to r(\delta(\bar{0}, x)\ ;\ P(a) \vdash\ ;\ P(a)) \\
\varrho(k+1, x) &\to r(\delta(k+1, x)\ ;\ P(f(g(k, z_k))) \vdash\ ; P(f(g(k, z_k)))\ ) \\
\delta(\bar{0}, x) &\to \vdash P(a) \\
\delta(k+1, x) &\to r(\delta(k, x)\ ; P(g(k, z_k)) \vdash P(f(g(k, z_k)))\ ; P(g(k, z_k))\ )
\end{aligned}$$

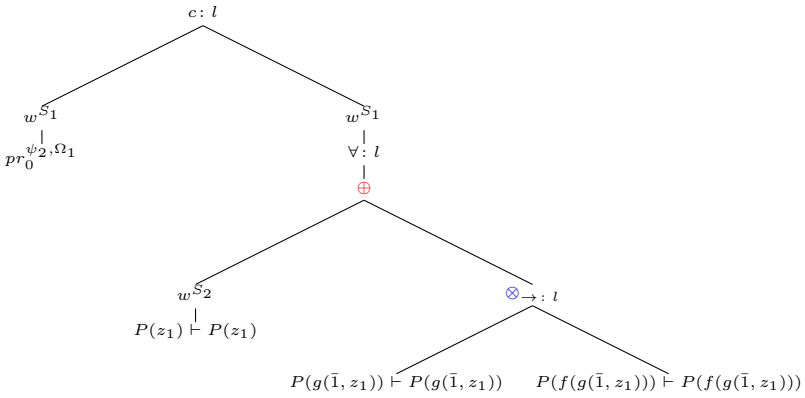
For the instance $k = 1$ and ground substitution $\{z_0 \leftarrow a, z_1 \leftarrow a\}$ we compute $\delta(2, x)$ and transform the resulting ground term after term-rewriting to a tree which indeed is a resolution refutation $\mathcal{R}$ of $\Psi\downarrow_1$:



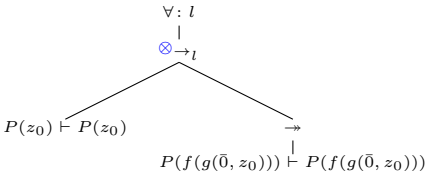
The next step is to compute the proof projections. For $\Xi(\nu_1(1), \emptyset)$ we have the following projection term represented as a tree:



where the leaf $pr_1^{\psi_2,\Omega_1}$ rewrites to the term:



and the leaf $pr_0^{\psi_2,\Omega_1}$ rewrites to the term:



where:

$\Omega_1 = (\ |\ \forall x(P(x) \rightarrow P(f(g(k,x))))\ )$
$S_1 = \forall x(\ P(x) \rightarrow P(f(x))\ ) \vdash$
$S_2 = P(g(k+1, z_{k+1})) \rightarrow P(f(g(k+1, z_{k+1}))) \vdash$

Since $\Xi(\nu_1(1), \emptyset)$ is a ground term, we can compute the set of proof projections $|\Xi(\nu_1(1), \emptyset)|$. More precisely, we need only those projections which are in the range of the mapping which maps a leaf from the ground resolution refutation $\mathcal{R}$ to a (ground) proof projection from $|\Xi(\nu_1(1), \emptyset)|$.

The projection $PR(\ \vdash P(a)\ )$ is:

$$\dfrac{\dfrac{P(a) \vdash P(a)}{P(a) \ \vdash \ P(a)\ ,\ P(f^2(a))}\, w\colon r}{P(a)\ ,\ \forall x(P(x) \to P(f(x))) \ \vdash \ P(a)\ ,\ P(f^2(a))}\, w\colon l$$

The projection $PR(\ P(z_0) \vdash P(f(z_0))\ )$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(z_0) \vdash P(z_0) \qquad \dfrac{P(f(g(\bar{0}, z_0))) \vdash P(f(g(\bar{0}, z_0)))}{P(f(z_0)) \vdash P(f(z_0))}\, \twoheadrightarrow}{P(z_0)\ ,\ P(z_0) \to P(f(z_0)) \vdash P(f(z_0))}\, \to\colon l}{P(z_0)\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f(z_0))}\, \forall\colon l}{P(z_0)\ ,\ \forall x(P(x) \to P(f(x)))\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f(z_0))}\, w\colon l}{P(z_0)\ ,\ \forall x(P(x) \to P(f(x)))\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f(z_0))}\, c\colon l}{P(a)\ ,\ P(z_0)\ ,\ \forall x(P(x) \to P(f(x))) \ \vdash \ P(f(z_0))\ ,\ P(f^2(a))}\, w\colon l$$

The projection $PR(\ P(f(z_1)) \vdash P(f(f(z_1)))\ )$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P(g(\bar{1}, z_1)) \vdash P(g(\bar{1}, z_1)) \qquad P(f(g(\bar{1}, z_1))) \vdash P(f(g(\bar{1}, z_1)))}{P(g(\bar{1}, z_1))\ ,\ P(g(\bar{1}, z_1)) \to P(f(g(\bar{1}, z_1))) \vdash P(f(g(\bar{1}, z_1)))}\, \to\colon l}{P(g(\bar{1}, z_1))\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f(g(\bar{1}, z_1)))}\, \forall\colon l}{P(g(\bar{1}, z_1))\ ,\ \forall x(P(x) \to P(f(x)))\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f(g(\bar{1}, z_1)))}\, w\colon l}{P(g(\bar{1}, z_1))\ ,\ \forall x(P(x) \to P(f(x)))\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f(g(\bar{1}, z_1)))}\, c\colon l}{P(f(z_1))\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f^2(z_1))}\, \twoheadrightarrow}{P(a)\ ,\ P(f(z_1))\ ,\ \forall x(P(x) \to P(f(x))) \vdash P(f^2(z_1))\ ,\ P(f^2(a))}\, w\colon l, w\colon r$$

The projection $PR(\ \vdash P(f^2(a))\ )$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{P(f(g(\bar{1}, a))) \vdash P(f(g(\bar{1}, a)))}{P(a)\ ,\ P(f(g(\bar{1}, a))) \vdash P(f(g(\bar{1}, a)))}\, w\colon l}{\forall x(P(x) \to P(f(x)))\ ,\ P(a)\ ,\ P(f(g(\bar{1}, a))) \vdash P(f(g(\bar{1}, a)))}\, w\colon l}{\forall x(P(x) \to P(f(x)))\ ,\ P(a)\ ,\ P(f^2(a)) \ \vdash \ P(f^2(a))}\, \twoheadrightarrow$$

Finally, the *ACNF* is:

$$\dfrac{\dfrac{PR(\ \vdash P(a)\ )}{\Gamma \vdash \Delta, P(a)} \qquad \dfrac{PR(P(z_0) \vdash P(f(z_0)))\{z_0 \leftarrow a\}}{P(a), \Gamma \vdash \Delta, P(f(a))}}{\Gamma \vdash \Delta, P(f(a))}\, cut, c\colon l, r \qquad \dfrac{PR(P(f(z_1)) \vdash P(f^2(z_1)))\{z_1 \leftarrow a\}}{P(f(a)), \Gamma \vdash \Delta, P(f^2(a))}}{\Gamma \vdash \Delta, P(f(f(a)))}\, cut, c\colon l, r \qquad \dfrac{PR(P(f^2(a) \vdash\ )}{P(f^2(a)), \Gamma \vdash \Delta}$$
$$\Gamma \vdash \Delta \qquad cut$$

This completes the solid branch of figure 7.1 and shows some of the advantages of the method *CERES*$_s$ over the method *CERES*. Namely, for small characteristic clause sets, the refutation can easily be constructed without producing a potentially large input proof. A disadvantage of *CERES*$_s$ is that it is not fully automatable because the mathematician has to provide a resolution schema. On the contrary, the method *CERES* requires an instantiation of the proof schema which can be exponential in the size. The limitation in this case comes also from the fact that the refutations of $\mathrm{CL}(\psi_1 \downarrow_n)$ get larger with n and at some point the automated theorem prover fails to refute it.

Bibliography

[ACP09] V. Aravantinos, R. Caferra, and N. Peltier. A schemata calculus for propositional logic. In *18th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux 2009)*. Springer, 2009.

[ACP10] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. RegSTAB: A SAT-Solver for Propositional Iterated Schemata. In *International Joint Conference on Automated Reasoning*, pages 309–315, 2010.

[ACP11] V. Aravantinos, R. Caferra, and N. Peltier. Decidability and undecidability results for propositional schemata. *Journal of Artificial Intelligence Research*, 40:599–656, 2011.

[AZ99] M. Aigner and G. Ziegler. *Proofs from THE BOOK*. Springer, 1999.

[BHL+05] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Cut-elimination: Experiments with ceres. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 481–495. Springer, 2005.

[BHL+06] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Proof transformation by ceres. In Jonathan M. Borwein and William M. Farmer, editors, *Mathematical Knowledge Management (MKM) 2006*, volume 4108 of *Lecture Notes in Artificial Intelligence*, pages 82–93. Springer, 2006.

[BHL+08] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. CERES: An analysis of Fürsten-

berg's proof of the infinity of primes. *Theoretical Computer Science*, 403:160–175, 2008.

[BL74] Walter S. Brainerd and Lawrence H. Landweber. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, USA, 1974.

[BL00] Matthias Baaz and Alexander Leitsch. Cut-elimination and redundancy-elimination by resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000.

[BL06] Matthias Baaz and Alexander Leitsch. Towards a clausal analysis of cut-elimination. *Journal of Symbolic Computation*, 41(3-4):381–410, 2006.

[Coo04] S.B. Cooper. *Computability Theory*. Chapman and Hall, 2004.

[DLL+12a] Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller, and Bruno Woltzenlogel-Paleo. System Feature Description: Importing Refutations into the GAPT Framework. In David Pichardie and Tjark Weber, editors, *Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012)*, volume 878 of *CEUR Workshop Proceedings*, pages 51–57, 2012.

[DLL+12b] Tsvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller, and Bruno Woltzenlogel-Paleo. Prooftool: Gui for the gapt framework. In *UITP 2012*, 2012.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(1):176–210, dec 1935.

[HLW12] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. Towards algorithmic cut-introduction. In *LPAR*, pages 228–242, 2012.

[HLWWP08a] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. CERES in second-order logic. Technical report, Vienna University of Technology, 2008.

[HLWWP08b] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. Herbrand sequent extraction. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge,

Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 462–477. Springer Berlin, 2008.

[HLWWP08c] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. Proof analysis with HLK, CERES and ProofTool: Current status and future directions. In Geoff Sutcliffe, Simon Colton, and Stephan Schulz, editors, *Proceedings of CICM Workshop ESARM'08*, pages 18–41, 2008.

[HLWWP08d] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. Transforming and analyzing proofs in the CERES-system. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate Schmidt, and Stephan Schulz, editors, *Proceedings of the LPAR 2008 Workshops*, pages 77–91, 2008.

[Hue02] Gérard Huet. Higher order unification 30 years later. In *Theorem Proving in Higher Order Logics (TPHOLs) 2002*, volume 2410 of *Lecture Notes in Computer Science*, pages 241–258. Springer Berlin, 2002.

[LWWP$^+$] Tomer Libal, Daniel Weller, Bruno Woltzenlogel-Paleo, Tsvetan Dunchev, Mikheil Rukhaia, and Martin Riener. Generic Architecture for Proofs. http://code.google.com/p/gapt.

[McC10] W. McCune. Prover9 and mace4. http://www.cs.unm.edu/ mccune/prover9/, 2005–2010.

[Ore82] V. P. Orevkov. Lower bounds for increasing complexity of derivations after cut elimination. *Journal of Mathematical Sciences*, 20(4):2337–2350, 1982.

[OSV10] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide (2nd ed.)*. Artima Inc., 2010.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Ruk12] Mikheil Rukhaia. *CERES in proof schemata*. PhD thesis, Vienna University of Technology, 2012.

[Tur36] Alan Turing. In *On computable numbers, with an application to the Entscheidungsproblem*, pages 230–265. London Mathematical Society, Series 2, 42, 1936.

[WP08] Bruno Woltzenlogel Paleo. *Herbrand Sequent Extraction*. VDM-Verlag, Saarbruecken, Germany, 2008.

[WP09] Bruno Woltzenlogel-Paleo. *A General Analysis of Cut-Elimination by CERes*. PhD thesis, Vienna University of Technology, 2009.

Index