TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DISSERTATION

Inductive theorem proving using tree grammars

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Assoc. Prof. Dr. techn. Stefan Hetzl
E104 – Institut für Diskrete Mathematik und Geometrie

eingereicht an der Technischen Universität Wien
Fakultät für Mathematik und Geoinformation

von

Gabriel Ebner
(Matrikelnummer 0726022)
Max Havelaarlaan 355A
1183LW Amstelveen

Wien, am

Gabriel Ebner

Deutsche Kurzfassung der Dissertation

Der Satz von Herbrand [45], ein grundlegendes Ergebnis der Logik und Beweistheorie, charakterisiert die Gültigkeit von quantifizierten Formeln der klassischen Logik erster Ordnung durch die Existenz einer tautologischen endlichen Menge von quantorenfreien Grundinstanzen. Im einfachsten Fall entspricht einer gültigen, rein existenziellen Formel $\exists x\, \varphi(x)$ eine tautologische Disjunktion $\varphi(t_1) \vee \cdots \vee \varphi(t_n)$, eine sogenannte Herbrand-Disjunktion.

Schnittfreie Beweise enthalten die zur Bildung einer solchen Herbrand-Disjunktion notwendigen Terme [16] unmittelbar in ihren Quantorenschlüssen. Die fundamentalste Operation der Beweistheorie, die Gentzensche Schnittelimination [43], enthält einen Algorithmus, der aus Beweisen die Schlussfigur des Schnitts eliminiert, und somit aus einem Beweis einer rein existenziellen Formel eine Herbrand-Disjunktion berechnet.

Ein moderner Ansatz, um diese Schnittelimination auf Ebene der Quantorenschlüsse mittels formalsprachlichen Methoden zu verstehen, wurde von Hetzl vorgestellt [46]: jedem Beweis π in einer geeigneten Klasse wird eine Grammatik $G(\pi)$ auf solche Art zugeordnet, dass die Grammatik unter der Schnitteliminationsoperation erhalten bleibt. Die von der Grammatik erzeugte Sprache $L(G(\pi))$ ist dann isomorph zu einer Herbrand-Disjunktion; um die Schnittelimination zu verstehen, reicht es folglich, diese Grammatiken zu verstehen.

Die erste Instanz dieses Homomorphismus in [46] verbindet Beweise von schwach quantifizierten Pränexsequenten, deren Schnittformeln aus Pränexformeln ohne Quantorenalternation bestehen, mit vektoriellen totalrigiden Baumgrammatiken (VTRATGs). Darauffolgende Verallgeimeinerungen auf umfangreichere und kompliziertere Beweisklassen verlangen dementsprechend ausdrucksstärkere Grammatikklassen [31, 55, 2].

Für Beweise mit Induktion ist es sogar erforderlich, den Begriff der Herbrand-Disjunktion selbst zu verallgemeinern. Ein Beweis π von $\forall x\, \varphi(x)$ in einer geeigneten Klasse von Beweisen mit Induktion induziert auf natürliche Weise für jeden Konstruktorterm t einen induktionsfreien Instanzbeweis π_t von $\varphi(t)$. Somit entsteht eine durch Konstruktorterme indizierte Familie von Herbrand-Disjunktionen $(L(\pi_t))_t$, die jeweils durch die von der Grammatik des Induktionsbeweises instanzierten Grammatik $I(G(\pi), t) \supseteq L(\pi_t)$ überdeckt werden.

Die Umkehrung der Schnittelimination ist vielleicht die größte Herausforderung der Beweistheorie überhaupt. Beweise mit Schnitt können im allgemeinen nicht-elementar kleiner sein als schnittfreie Beweise; eine Umkehrung der Schnittelimination, in anderen Worten eine *Schnitteinführung*, ermöglicht daher direkt eine enorme Beweiskompression. Weiters fasst die Schlussfigur des Schnitts auf formale Weise den Begriff des mathematischen Satzes; die Schnitteinführung führt somit auch Hilfssätze in Beweise ein, und entdeckt folglich in einem gewissen Sinne sogar neue mathematischen Konzepte.

Auf Ebene der Gentzenschen Schnittreduktionsrelation ist die Umkehrung der Schnittelimination jedoch völlig aussichtslos: allein um nur einen einzigen Schritt umzukehren, gibt es schon unendliche viele Möglichkeiten (man denke nur an den Fall der Verdünnungsreduktion). Unter dem Bild des grammatikalischen Homomorphismus hingegen betrachtet, entbart sich die Schnitteinführung nicht nur als praktisch erfolgreich durchführbar [36], sondern sie zerfällt sogar in zwei klar abgegrenzte Teilprobleme. Angenommen wir beginnen mit einem schnittfreien Beweis π. Dann bildet ersteres Teilproblem das direkte formalsprachliche Gegenstück zur Umkehrung der Schnittelimination, und besteht aus der Lösung des Überdeckungsproblem, also eine Grammatik G zu finden, sodass $L(G) \supseteq L(\pi)$. Das zweite Teilproblem übersetzt die Grammatik aus der formalsprachlichen Welt zurück zu einem Beweis π, sodass $G(\pi) = G$.

Derselbe grammatikalische Ansatz glückt auch für die Einführung von Induktionsschlussfiguren, wobei im Unterschied zur Schnitteinführung die Nichtanalytizität der Induktionsformeln—dass sie also nicht bereits wörtlich im zu beweisenden Sequent vorkommen—von essenziellem und unerlässlichem Charakter ist. Eberhard und Hetzl [31] schlagen diesen Ansatz als zukunftsweisendes Paradigma für das induktive Beweisen vor: schnittfreie Instanzbeweise sind durch automatische Theorembeweiser mühelos zu erzeugen, und können dann zu einem Beweis mit Induktion verallgemeinert werden.

Diese Dissertation nimmt sich das Ziel, eine praktisch einsetzbare Implementierung dieses neuen Paradigmas umzusetzen. Diesem Ziel vorangehend, erkunden wir zunächst ein reichhaltiges Spektrum an theoretischen Fragestellungen, die unser Verständnis für die zugrundelegenden Strukturen vertiefen.

Eine grundlegende Frage, die sich beim Finden von überdeckenden Grammatiken stellt, ist eine komplexitätstheoretische: wie schwierig ist es, eine

kleinstmögliche überdeckende Grammatik zu finden? Wie schwierig ist es überhaupt, festzustellen, ob eine Grammatik eine gegebene Termmenge überdeckt? Oder ob zwei Grammatiken dieselbe Sprache erzeugen? Fragen dieser Art werden wir in Kapitel 3 für unterschiedliche Klassen von Grammatiken untersuchen und auch mit anderen formalsprachlichen Modellen in Beziehung setzen.

Darauffolgend ergibt sich natürlich die Frage, wie wir praktisch überdeckende Grammatiken finden können. Dazu stellen wir in Kapitel 4 drei unterschiedliche Algorithmen vor.

Das Erzeugen einer überdeckenden Grammatik ist jedoch nur der erste Zwischenschritt: danach gilt es, einen Beweis zu erzeugen, dem diese Grammatik zugeordnet ist. Konkret sind die Matrizen der Schnitt- und Induktionsformeln zu finden. Die Bedingungen, denen diese Matrizen zu genügen haben, bilden eine sogenannte Formelgleichung, welche in engem Zusammenhang mit dem Lemma von Ackermann [1] auf der theoretischen Seite, und mit bedingten Hornklauseln [12] auf der praktischen Seite stehen. In Kapitel 5 beschreiben wir diesen Zusammenhang, untersuchen Fragestellungen zur Lösbarkeit von den relevanten Formelgleichungen, und stellen zwei Lösungsalgorithmen vor.

Um Herbrand-Disjunktionen von automatischen Theorembeweisern zu erhalten, stellen wir in Kapitel 6 einen neuen Algorithmus vor. Dieser wandelt Resolutionsbeweise in Expansionsbäume—eine Verallgemeinerung von Herbrand-Disjunktionen—um, ohne als Zwischenschritt einen schnittfreien Beweis zu konstruieren.

Zum Schluss, in Kapitel 7, fügen wir die Puzzlesteine aus den vorangehenden Kapiteln zu einer vollständigen Implementierung zusammen, und evaluieren diese. Die Testergebnisse erschließen einen tiefliegenden Unterschied zwischen automatisch erzeugten Beweisen und Beweisen, die aus der Induktionselimination gewonnen sind. Diese Andersartigkeit ist selbst auf Ebene der Herbrand-Disjunktionen eindeutig erkennbar.

Acknowledgements

I would like to thank my advisor, Stefan Hetzl, without whom this thesis would have found neither a beginning nor an end.

Contents

1 Introduction

Herbrand's theorem [45, 16] captures the fundamental insight in logic that the validity of a quantified formula is characterized by the existence of a tautological finite set of quantifier-free instances. In the simplest case, a proof of a purely existential formula $\exists x\, \varphi(x)$ gives rise to the existence of a tautological disjunction of quantifier-free instances $\varphi(t_1) \vee \cdots \vee \varphi(t_n)$, a Herbrand disjunction.

The fundamental importance of Herbrand's theorem is underlined by the variety of its applications. Herbrand disjunctions directly contain the answer substitutions of logic programs. Luckhardt [66] used them to give a polynomial bound for Roth's theorem. Herbrand's theorem also turns up in automated reasoning. The leaves of a ground resolution refutation are the instances in a Herbrand disjunction. Lifting the ground refutation to a first-order refutation then shows the completeness of first-order resolution [9]. Even going beyond classical logic, we can use the information contained in Herbrand disjunctions to constructivize classical proofs into intuitionistic proofs in a practically highly effective way [35]. Closer to the topic of this thesis, compressing Herbrand disjunctions using tree grammars has been used to introduce non-analytic quantified lemmas [51, 50, 48, 36], and prove theorems with induction in [31].

The information necessary to form Herbrand disjunctions is directly contained in the quantifier inferences in cut-free proofs [45, 16]. Herbrand disjunctions generalize naturally to Herbrand sequents, where the validity of a prenex sequent is characterized by the existence of a tautological sequent of instances. Another generalization of Herbrand disjunctions are expansion trees, where the validity of a (not necessarily prenex) sequent is characterized by the existence of an expansion sequent whose deep sequent is tautological [71]. The last characterization reaches beyond first-order logic and even applies to elementary type theory.

Gentzen's cut-elimination [43] is perhaps the most fundamental operation in proof theory. It provides an algorithmic means to eliminate cut inferences from proofs, and hence compute Herbrand disjunctions. By extending cut-elimination to also unfold induction inferences into sequences of cuts, it is possible to eliminate induction and cut inferences from proofs of existential formulas, thus showing the consistency of Peano arithmetic [41].

Cut-elimination transforms complicated proofs with cuts into simple proofs in a cut-free normal form. Concretely, cut inferences formalize the notion of lemmas, which capture the deep mathematical insights contained in proofs. Understanding cut-elimination is thus one of the most important endeavours in mathematical logic, and allows us to gain insight in very nature of lemmas. If we understood cut-elimination well-enough to be able to reverse it algorithmically, then we would have an algorithm to find interesting lemmas, compress large proofs, and structure automatically-generated proofs.

One novel approach to understand cut-elimination based on formal languages was introduced by Hetzl [46], introducing a connection between a class of proofs where cut formulas are restricted to purely universally or existentially quantified prenex formulas and totally rigid regular tree grammars. In this approach there is a function that assigns to every proof π a grammar $G(\pi)$ such that the language generated by the grammar $L(G(\pi))$ corresponds to a Herbrand sequent. This grammar describes the quantifier inferences in the proof π, and is also preserved under cut-elimination in a suitable way. Cut-elimination hence corresponds to language generation of the grammar. In order to understand cut-elimination on the level of quantifier inferences, it suffices to understand the language generation operation of the grammars.

This correspondence has been extended to a class of proofs with induction in [31]. Given proof π of $\forall x\ \varphi(x)$ in that class (where x ranges over the natural numbers), we get for every numeral n a proof π_n of the instance $\varphi(n)$. The grammar $G(\pi)$ for the induction proof is then *instantiated* to a grammar $I(G(\pi), t)$ for the proof π_n, and $L(I(G(\pi), t))$ is a Herbrand sequent of the instance for n. In this setting, language generation of the grammar corresponds to induction- and cut-elimination.

Understanding induction- and cut-elimination on the level of grammars allows us to reverse this process [51, 50, 48, 31, 36]. Starting from a Herbrand

sequent or cut-free proof, we can first find a grammar that *covers* the Herbrand sequent (i.e., $L(G) \supseteq L$ where L corresponds to the Herbrand sequent). This grammar determines the quantifier inferences in the proof with cut. In the next step we need to find the quantifier-free matrices of the cut formulas, these form the solution to a so-called formula equation that is induced by the grammar.

An analogous approach allows us to reverse induction-elimination (in the restricted class of proofs with induction that we consider). Starting from a finite family of proofs of instances $\varphi(n_1), \ldots, \varphi(n_k)$, we first find a covering grammar and then solve the induced formula equation. The case for proofs with induction is more complicated than for cuts: the induced formula equation might not be solvable in general. The proofs of $\varphi(n_1), \ldots$ can also be produced by automated theorem provers: then this approach yields an automated inductive theorem prover. On a theoretical level this was introduced in [31].

This thesis builds a practical implementation of this approach to automated inductive theorem proving. Each of the steps in the approach poses challenges. The first step requires us to construct algorithms that find small covering grammars. For this it is important to have a general understanding of the complexity of decision problems on the classes of grammars that we consider. For example, it is a priori not even clear how hard it is to decide whether a grammar covers a set of terms. (It is NP-hard for all classes that we consider, as we will see in Theorems 3.2.1 and 3.12.2.) We will explore these complexity questions in Chapter 3. Equipped with this knowledge, we will then discuss concrete algorithm to find covering grammars in Chapter 4.

The central concept of the second step is that of a formula equation. A solution to the formula equation induced by a grammar will contain exactly the information necessary to produce a proof with this grammar. These formula equations and algorithms to solve them will be discussed in Chapter 5.

One way to obtain the Herbrand sequents (or cut-free proofs) consists of using automated theorem provers. Chapter 6 presents an efficient algorithm to convert resolution proofs produced by such automated theorem provers to expansion proofs, a generalization of Herbrand sequents.

Finally, we will combine these puzzle pieces in Chapter 7, describing a prac-

tical implementation of the approach. We will evaluate this implementation on real-world benchmarks. The results of this evaluation will uncover an interesting difference between automatically generated proofs and proofs generated by induction elimination. A detailed case study will illustrate this difference that even leaves traces on the level of Herbrand disjunctions.

2 Proofs with induction

In this chapter we will define the basic notation that we require. We will begin with the sequent calculus that we use and the cut-reduction for this system. Based on this calculus, we will define several classes of proofs and corresponding classes of tree grammars that describe the cut-elimination of the proofs, in such a way that the language of the grammar is isomorphic a Herbrand sequent. One typical example of such a theorem that will describe the relation between proofs and grammars will be Lemma 2.7.5, stating that (for a class of proofs where all cut formulas are purely universally or existentially quantified) the language generated by the grammar is preserved under cut-reduction. This implies in particular that cut-elimination commutes with language generation in the following sense:

$$\begin{array}{ccc} \pi & \xrightarrow{\quad\text{cut}\quad} & \pi^* \\ \downarrow & & \downarrow \\ G(\pi) & \longmapsto & L(G(\pi)) \supseteq L(\pi^*) \end{array}$$

2.1 Calculus

We treat many-sorted terms as a subset of terms in a simply-typed lambda calculus, i.e. those terms consisting only of constants, variables, and applications that are of first-order type. The arity of a function symbol is the number of the arguments it can be applied to. In this sense, constants are just nullary functions. A vector of terms $\overline{t} = (t_1, \ldots, t_n)$ is a finite sequence of terms. If a function symbol f has type $\alpha \to \beta \to \gamma$ and t, s have types α, β, resp., then we write the application as $f(t, s)$.

Positions p are finite sequences of natural numbers; $\mathrm{Pos}(t)$ is the set of positions in a term, $t|_p$ is the subterm of t at position p, and $t[s]_p$ is the term t

where the position p is replaced by another term s. The depth $\mathrm{depth}(t)$ of a term t is the maximum length of a position $p \in \mathrm{Pos}(t)$. The set $\mathrm{st}(t) = \{t|_p \mid p \in \mathrm{Pos}(t)\}$ is the set of subterms of t. We also define the relations $t \trianglelefteq s$ iff $t \in \mathrm{st}(s)$, and $t \triangleleft s$ iff $t \in \mathrm{st}(s) \setminus \{s\}$. The set of subterms $\mathrm{st}(L)$ of a set of terms $L \subseteq \mathcal{T}(\Sigma)$ is given by $\mathrm{st}(L) = \bigcup_{t \in L} \mathrm{st}(L)$. A term t subsumes a term s, written $t \preceq s$, if there exists a substitution σ such that $t\sigma = s$.

For a term t, the set $\mathrm{FV}(t)$ is called the set of free variables in t, the term t is called ground if $\mathrm{FV}(t) = \emptyset$. Variables can be substituted by terms: let $x_1, \ldots, x_k \in X$ be variables and $s_1, \ldots, s_k \in \mathcal{T}(\Sigma \cup X)$ be terms, then a (parallel) substitution $\sigma = [x_1 \backslash s_1, x_2 \backslash s_2, \ldots, x_k \backslash s_k]$ is a finite map from variables to terms of the same type, and is extended to all terms recursively. The domain the substitution σ is the set of variables $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_k\}$ being substituted. Each of the terms s_i may contain any variable, including any of the variables x_j. We write $t\sigma$ for the application of a substitution σ to a term t. Substitutions are extended to vectors as well by substituting in each element: $\overline{t}\sigma = (t_1\sigma, \ldots, t_n\sigma)$. We will also define substitutions using vectors: $[\overline{x} \backslash \overline{t}] = [x_1 \backslash t_1, \ldots, x_n \backslash t_n]$.

We consider a many-sorted first-order logic where some sorts are structurally inductive data types. A *structurally inductive data type* is a sort ρ with distinguished functions $c_1, \ldots, c_n$ called *constructors*. Each constructor c_i has the type $\tau_{i,1} \to \cdots \to \tau_{i,n_i} \to \rho$, that is, the arguments have the types $\tau_{i,1}, \ldots, \tau_{i,n_i}$ and the return type of the constructor is ρ. It may be the case that $\tau_{i,j} = \rho$, then the index j of such an argument is called a *recursive occurrence* in the constructor c_i. We do not consider mutually inductive types, etc. The intended semantics is that ρ is the set of finite terms freely generated by the constructors and values of the other argument types. Our proof system will only ensure that ρ is inductively generated by the constructors, other properties of the free term algebra will be explicit assumptions in the proofs.

Example 2.1.1. Natural numbers have the type ω with constructors 0^{ω} and $s^{\omega \to \omega}$. The first argument of s is a recursive occurrence.

Example 2.1.2. Let τ be a type, then the lists with elements from τ have the type list with the constructors $\mathsf{nil}^{\mathsf{list}}$ and $\mathsf{cons}^{\tau \to \mathsf{list} \to \mathsf{list}}$. The constructor cons has two arguments, one representing the first element of the list, the other representing the rest of the list. Only the second argument is a recursive

occurrence.

A sequent $\Gamma \vdash \Delta$ consists of two multisets of formulas Γ and Δ, and is interpreted as the formula $\bigwedge \Gamma \to \bigvee \Delta$. The multiset Γ is called the *antecedent* and Δ is the *succedent* of the sequent. Figure 2.1 shows the sequent calculus $\mathsf{LK}(T)$ that we consider.

We assume a background theory T, and the inference rule T then allows us to infer any atomic sequent that follows from the background theory. The main background theory that we will use is the theory of equality; the inference rule T can then infer sequents such as $a = b, f(b) = b \vdash g(b) = g(f(f(f(a))))$. For an atomic sequent, it is decidable whether it follows from the theory equality. This is the main motivation for the restriction to atomic sequents here. Another way to incorporate equality would be to add inference rules for rewriting and reflexivity:

$$\frac{}{\vdash t = t}\ \text{refl} \qquad \frac{\Gamma \vdash \Delta, t = s \qquad \varphi(t), \Pi \vdash \Lambda}{\Gamma, \varphi(s), \Pi \vdash \Delta, \Lambda}\ \text{rw}_l^{\to}$$

However this choice of inference complicates cut-elimination, as it would require us to permute the rewriting inference with inferences in the two subproofs. If we move equational reasoning into the leaves, then it does not get in the way. In addition, the formalism becomes more general, for example we could also use Presburger arithmetic as a background theory.

Given an inference rule such as the right-introduction rule $\wedge_r$ for conjunction:

$$\frac{\Gamma \vdash \Delta, \varphi \qquad \Pi \vdash \Lambda, \psi}{\Gamma, \Pi \vdash \Delta, \Lambda, \varphi \wedge \psi}\ \wedge_r$$

The sequents $\Gamma \vdash \Delta, \varphi$ and $\Pi \vdash \Lambda, \psi$ on top are called the *premises* of the inference, and the sequent $\Gamma, \Pi \vdash \Delta, \Lambda, \varphi \wedge \psi$ below is called the *conclusion*. The formulas φ and ψ in the premises are called *auxiliary formulas*, the formula $\varphi \wedge \psi$ in the conclusion is called the *main formula*. The inferences $\forall_l$ and $\exists_r$ are called *weak quantifier inferences*, and $\forall_r$ and $\exists_l$ are called *strong quantifier inferences*.

For every inductive sort ρ there is a corresponding structural induction rule. This rule has one premise for each constructor c_i of the inductive type, and for every recursive argument α_{ji} of the constructor there is an inductive hypothesis.

$$\frac{}{\varphi \vdash \varphi}\,\mathrm{ax} \quad \text{(if } \varphi \text{ is an atomic formmula)}$$

$$\frac{\Gamma \vdash \Delta}{\varphi, \Gamma \vdash \Delta}\,w_l \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \varphi}\,w_r \qquad \frac{\varphi, \varphi, \Gamma \vdash \Delta}{\varphi, \Gamma \vdash \Delta}\,c_l \qquad \frac{\Gamma \vdash \Delta, \varphi, \varphi}{\Gamma \vdash \Delta, \varphi}\,c_r$$

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

$$\frac{}{\Gamma \vdash \Delta}\,\mathrm{T} \quad \text{(if } \Gamma \vdash \Delta \text{ is an atomic sequent entailed by } T\text{)}$$

$$\frac{}{\vdash \top}\,\top_r \qquad \frac{}{\bot \vdash}\,\bot_l \qquad \frac{\Gamma \vdash \Delta, \varphi}{\neg\varphi, \Gamma \vdash \Delta}\,\neg_l \qquad \frac{\varphi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg\varphi}\,\neg_r$$

$$\frac{\Gamma \vdash \Delta, \varphi, \psi}{\Gamma \vdash \Delta, \varphi \vee \psi}\,\vee_r \qquad \frac{\varphi, \Gamma \vdash \Delta \qquad \psi, \Pi \vdash \Lambda}{\varphi \vee \psi, \Gamma, \Pi \vdash \Delta, \Lambda}\,\vee_l$$

$$\frac{\varphi, \psi, \Gamma \vdash \Delta}{\varphi \wedge \psi, \Gamma \vdash \Delta}\,\wedge_l \qquad \frac{\Gamma \vdash \Delta, \varphi \qquad \Pi \vdash \Lambda, \psi}{\Gamma, \Pi \vdash \Delta, \Lambda, \varphi \wedge \psi}\,\wedge_r$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\,\to_r \qquad \frac{\Gamma \vdash \Delta, \varphi \qquad \psi, \Pi \vdash \Lambda}{\varphi \to \psi, \Gamma, \Pi \vdash \Delta, \Lambda}\,\to_l$$

$$\frac{\Gamma \vdash \Delta, \varphi(t)}{\Gamma \vdash \Delta, \exists x\, \varphi(x)}\,\exists_r \qquad \frac{\varphi(\alpha), \Gamma \vdash \Delta}{\exists x\, \varphi(x), \Gamma \vdash \Delta}\,\exists_l$$

$$\frac{\varphi(t), \Gamma \vdash \Delta}{\forall x\, \varphi(x), \Gamma \vdash \Delta}\,\forall_l \qquad \frac{\Gamma \vdash \Delta, \varphi(\alpha)}{\Gamma \vdash \Delta, \forall x\, \varphi(x)}\,\forall_r$$

$$\frac{\Gamma, \varphi(\alpha_{j_1}), \ldots, \varphi(\alpha_{j_{k_i}}) \vdash \Delta, \varphi(c_i(\alpha_1, \ldots, \alpha_{m_i})) \qquad \text{(for each } c_i\text{)}}{\Gamma \vdash \Delta, \varphi(t)}\,\mathrm{ind}_\rho$$

Figure 2.1: The calculus $\mathsf{LK}(T)$ for classical many-sorted first-order logic with background theory T. The variable α in the $\exists_l$ and $\forall_r$ inferences is called an eigenvariable, and may not occur in $\Gamma, \Delta, \forall x\, \varphi(x)$ as a free variable.

The variables $\alpha_1, \ldots, \alpha_{m_i}$ (for each constructor c_i) are eigenvariables of the inference, that is, they may not occur in Γ or Δ. The term t is called the *main term* of the induction inference.

Example 2.1.3. Consider the instance of the induction inference $\mathrm{ind}_{\mathsf{list}}$ for lists. There are two premises, one for each constructor. The inference has two eigenvariables, α_1 and α_2, both occurring in the premise for the cons constructor. The eigenvariable α_2 is a recursive occurrence (of type list) and hence has a corresponding induction hypothesis in the antecedent of that premise: $\varphi(\alpha_2)$. The other eigenvariable α_1 is not a list, and is hence not a recursive occurrence and therefore has no corresponding induction hypothesis.

$$\frac{\Gamma \vdash \Delta, \varphi(\mathsf{nil}) \qquad \varphi(\alpha_2), \Gamma \vdash \Delta, \varphi(\mathsf{cons}(\alpha_1, \alpha_2))}{\Gamma \vdash \Delta, \varphi(t)}\,\mathrm{ind}_{\mathsf{list}}$$

Given a proof π of a sequent $\Gamma \vdash \Delta$, we can apply a substitution σ to the proof to obtain a proof $\pi\sigma$ of $\Gamma\sigma \vdash \Delta\sigma$. This substitution can be computed recursively: for all inferences except $\forall_r$ and $\exists_l$ it is enough to apply the substitution recursively to the premises and to the sequents in the inference. However for the strong quantifier inferences we might need to rename the eigenvariable to satisfy the eigenvariable condition of the inference:

$$\left(\frac{\begin{array}{c}(\pi)\\ \Gamma \vdash \Delta, \varphi(\alpha)\end{array}}{\Gamma \vdash \Delta, \forall x\, \varphi(x)}\,\forall_r\right)\sigma \quad = \quad \frac{\begin{array}{c}(\pi[\alpha\backslash\beta]\sigma)\\ \Gamma\sigma \vdash \Delta\sigma, \varphi(\beta)\sigma\end{array}}{\Gamma\sigma \vdash \Delta\sigma, (\forall x\, \varphi(x))\sigma}\,\forall_r$$

Where the variable β does not occur in the domain or range of the substitution σ, nor as free variable in the sequent $\Gamma \vdash \Delta, \forall x\, \varphi(x)$. This renaming in proof substitution is analogous to the capture-avoiding substitution of lambda calculus terms, where we have to rename the bound variable x in $(\lambda x\, xy)[y\backslash x] = (\lambda z\, zx)$.

Definition 2.1.1. Let π be an $\mathsf{LK}(T)$-proof. Then $|\pi|$ is the number of inferences in π (counted as a tree), and $|\pi|_q$ is the number of weak quantifier inferences ($\forall_l, \exists_r$).

$$\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta}\, w_r \qquad (\pi_2)\quad \varphi, \Pi \vdash \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow} \quad \dfrac{(\pi_1)}{\Gamma \vdash \Delta}\, w_l^*, w_r^*$$

$$\dfrac{(\pi_1)\quad \Gamma \vdash \Delta, \varphi \qquad \dfrac{(\pi_2)}{\Pi \vdash \Lambda}\, w_l}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow} \quad \dfrac{(\pi_2)}{\Pi \vdash \Lambda}\, w_l^*, w_r^*$$

Figure 2.2: Erasing reduction rules for weakening (the double line indicates an abbreviation of multiple inferences)

$$\dfrac{\dfrac{}{\varphi \vdash \varphi}\,\text{ax} \qquad (\pi)\quad \varphi, \Gamma \vdash \Delta}{\varphi, \Gamma \vdash \Delta}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow} \quad (\pi)\quad \varphi, \Gamma \vdash \Delta$$

$$\dfrac{(\pi)\quad \Gamma \vdash \Delta, \varphi \qquad \dfrac{}{\varphi \vdash \varphi}\,\text{ax}}{\Gamma \vdash \Delta, \varphi}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow} \quad (\pi)\quad \Gamma \vdash \Delta, \varphi$$

$$\dfrac{\dfrac{}{\Gamma \vdash \Delta, \varphi}\,\text{T} \qquad \dfrac{}{\varphi, \Pi \vdash \Lambda}\,\text{T}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow} \quad \dfrac{}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{T}$$

Figure 2.3: Grade-reduction rules for axioms

2.2 Cut- and induction-reduction

The cut inference can be eliminated from proofs without induction. This result goes back to Gentzen [43] and uses local rewriting rules to simplify the cuts by pushing them upwards towards the leaves of the proof and thereby eliminate them.

We will use three reduction relations: non-erasing cut-reduction $\overset{\text{ne}}{\longrightarrow}$ (Figures 2.3 to 2.7), cut-reduction $\overset{\text{cut}}{\longrightarrow}$ (in addition Figure 2.2), and combined cut- and induction-reduction $\overset{\text{ind}}{\longrightarrow}$ (in addition Figure 2.8). Each extends the previous one: $\overset{\text{ne}}{\longrightarrow} \subset \overset{\text{cut}}{\longrightarrow} \subset \overset{\text{ind}}{\longrightarrow}$. All of the three relations are closed under reflexivity, transitivity, and congruences.

The rank-reduction rules in Figure 2.7 have subtle implications. First, let us

$$\dfrac{\dfrac{\overset{(\pi_1)}{\Gamma \vdash \Delta, \varphi, \varphi}}{\Gamma \vdash \Delta, \varphi}\, c_r \qquad \overset{(\pi_2)}{\varphi, \Pi \vdash \Lambda}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow}$$

$$\dfrac{\overset{(\pi_1)}{\Gamma \vdash \Delta, \varphi, \varphi} \qquad \overset{(\pi_2)}{\varphi, \Pi \vdash \Lambda}}{\dfrac{\Gamma, \Pi \vdash \Delta, \Lambda, \varphi \qquad \overset{(\pi_2)}{\varphi, \Pi \vdash \Lambda}}{\dfrac{\Gamma, \Pi, \Pi \vdash \Delta, \Lambda, \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\, c_l^*, c_r^*}\,\text{cut}}\,\text{cut}$$

$$\dfrac{\overset{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \qquad \dfrac{\overset{(\pi_2)}{\varphi, \varphi, \Pi \vdash \Lambda}}{\varphi, \Pi \vdash \Lambda}\, c_l}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow}$$

$$\dfrac{\dfrac{\overset{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \qquad \dfrac{\overset{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \qquad \overset{(\pi_2)}{\varphi, \varphi, \Pi \vdash \Lambda}}{\varphi, \Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut}}{\dfrac{\Gamma, \Gamma, \Pi \vdash \Delta, \Delta, \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\, c_l^*, c_r^*}\,\text{cut}$$

Figure 2.4: Grade-reduction rules for contraction

note that they also apply for $\rho = \text{cut}$, i.e., we can permute cuts:

$$\dfrac{\dfrac{\overset{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \qquad \dfrac{\overset{(\pi_2)}{\Pi \vdash \Lambda, \psi} \qquad \overset{(\pi_3)}{\varphi, \psi, \Sigma \vdash \Xi}}{\varphi, \Pi, \Sigma \vdash \Lambda, \Xi}\,\text{cut}}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Xi}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow}$$

$$\dfrac{\dfrac{\overset{(\pi_2)}{\Pi \vdash \Lambda, \psi} \qquad \dfrac{\overset{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \qquad \overset{(\pi_3)}{\varphi, \psi, \Sigma \vdash \Xi}}{\Gamma, \psi, \Sigma \vdash \Delta, \Xi}\,\text{cut}}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Xi}\,\text{cut}$$

Using the same reduction rule, we can also permute the cuts back. This is one reason why $\overset{\text{ne}}{\longrightarrow}$ is already non-terminating. Another subtlety is that permuting a cut with a strong quantifier rule (i.e., $\forall_r$ or $\exists_l$) may require us to

$$\dfrac{\dfrac{\dfrac{(\pi_1)}{\varphi, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta, \neg\varphi}\,\neg_r \quad \dfrac{\dfrac{(\pi_2)}{\Pi \vdash \Lambda, \varphi}}{\neg\varphi, \Pi \vdash \Lambda}\,\neg_l}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow} \quad \dfrac{\dfrac{(\pi_2)}{\Pi \vdash \Lambda, \varphi} \quad \dfrac{(\pi_1)}{\varphi, \Gamma \vdash \Delta}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut}$$

$$\dfrac{\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi(t)}}{\Gamma \vdash \Delta, \exists x\, \varphi(x)}\,\exists_r \quad \dfrac{\dfrac{(\pi_2)}{\varphi(\alpha), \Pi \vdash \Lambda}}{\exists x\, \varphi(x), \Pi \vdash \Lambda}\,\exists_l}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow}$$

$$\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi(t)} \quad \dfrac{(\pi_2[\alpha\backslash t])}{\varphi(t), \Pi \vdash \Lambda}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut}$$

$$\dfrac{\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi(\alpha)}}{\Gamma \vdash \Delta, \forall x\, \varphi(x)}\,\forall_r \quad \dfrac{\dfrac{(\pi_2)}{\varphi(t), \Pi \vdash \Lambda}}{\forall x\, \varphi(x), \Pi \vdash \Lambda}\,\forall_l}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow}$$

$$\dfrac{\dfrac{(\pi_1[\alpha\backslash t])}{\Gamma \vdash \Delta, \varphi(t)} \quad \dfrac{(\pi_2)}{\varphi(t), \Pi \vdash \Lambda}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut}$$

Figure 2.5: Grade-reduction rules for $\neg, \exists, \forall$

$$\dfrac{\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi, \psi}}{\Gamma \vdash \Delta, \varphi \vee \psi} \vee_r \quad \dfrac{\dfrac{(\pi_2)}{\varphi, \Pi \vdash \Lambda} \quad \dfrac{(\pi_3)}{\psi, \Sigma \vdash \Xi}}{\varphi \vee \psi, \Pi, \Sigma \vdash \Lambda, \Xi} \vee_l}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Xi}\,\text{cut} \quad \overset{\text{cut}}{\rightarrow}$$

$$\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi, \psi} \quad \dfrac{(\pi_2)}{\varphi, \Pi \vdash \Lambda}}{\Gamma, \Pi \vdash \Delta, \Lambda, \psi}\,\text{cut} \quad \dfrac{(\pi_3)}{\psi, \Sigma \vdash \Xi}}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Sigma}\,\text{cut}$$

$$\dfrac{\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \quad \dfrac{(\pi_2)}{\Pi \vdash \Lambda, \psi}}{\Gamma, \Pi \vdash \Delta, \Lambda, \varphi \wedge \psi} \wedge_r \quad \dfrac{\dfrac{(\pi_3)}{\varphi, \psi, \Sigma \vdash \Xi}}{\varphi \wedge \psi, \Sigma \vdash \Xi} \wedge_l}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Xi}\,\text{cut} \quad \overset{\text{cut}}{\rightarrow}$$

$$\dfrac{\dfrac{(\pi_2)}{\Gamma \vdash \Delta, \psi} \quad \dfrac{\dfrac{(\pi_1)}{\Pi \vdash \Lambda, \varphi} \quad \dfrac{(\pi_3)}{\varphi, \psi, \Sigma \vdash \Xi}}{\psi, \Pi, \Sigma \vdash \Lambda, \Xi}\,\text{cut}}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Xi}\,\text{cut}$$

$$\dfrac{\dfrac{\dfrac{(\pi_1)}{\Gamma, \varphi \vdash \psi}}{\Gamma \vdash \varphi \rightarrow \psi} \rightarrow_r \quad \dfrac{\dfrac{(\pi_2)}{\Pi \vdash \Lambda, \varphi} \quad \dfrac{(\pi_3)}{\psi, \Sigma \vdash \Xi}}{\varphi \rightarrow \psi, \Pi, \Sigma \vdash \Lambda, \Xi} \rightarrow_l}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Xi}\,\text{cut} \quad \overset{\text{cut}}{\rightarrow}$$

$$\dfrac{\dfrac{\dfrac{(\pi_2)}{\Pi \vdash \Lambda, \varphi} \quad \dfrac{(\pi_1)}{\Gamma, \varphi \vdash \psi}}{\Gamma, \Pi \vdash \Delta, \Lambda, \psi}\,\text{cut} \quad \dfrac{(\pi_3)}{\psi, \Sigma \vdash \Xi}}{\Gamma, \Pi, \Sigma \vdash \Delta, \Lambda, \Xi}\,\text{cut}$$

Figure 2.6: Grade-reduction rules for $\vee$, $\wedge$, $\rightarrow$

$$\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi}\,\rho}{\dfrac{\Gamma' \vdash \Delta', \varphi \qquad (\pi_2)\ \varphi, \Pi \vdash \Lambda}{\Gamma', \Pi \vdash \Delta', \Lambda}\,\mathrm{cut}} \quad \overset{\mathrm{cut}}{\rightarrow} \quad \dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad (\pi_2)\ \varphi, \Pi \vdash \Lambda}{\dfrac{\Gamma, \Pi \vdash \Delta, \Lambda}{\Gamma', \Pi \vdash \Delta', \Lambda}\,\rho}\,\mathrm{cut}$$

$$\dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad \dfrac{(\pi_2)\ \varphi, \Pi \vdash \Lambda}{\varphi, \Pi' \vdash \Lambda'}\,\rho}{\Gamma, \Pi' \vdash \Delta, \Lambda'}\,\mathrm{cut} \quad \overset{\mathrm{cut}}{\rightarrow} \quad \dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad (\pi_2)\ \varphi, \Pi \vdash \Lambda}{\dfrac{\Gamma, \Pi \vdash \Delta, \Lambda}{\Gamma, \Pi' \vdash \Delta, \Lambda'}\,\rho}\,\mathrm{cut}$$

$$\dfrac{\dfrac{(\pi_1)\ \Gamma \vdash \Delta \qquad (\pi_2)\ \Pi \vdash \Lambda, \varphi}{\Gamma', \Pi' \vdash \Delta', \Lambda', \varphi}\,\rho \qquad (\pi_3)\ \varphi, \Sigma \vdash \Xi}{\Gamma', \Pi', \Sigma \vdash \Delta', \Lambda', \Xi}\,\mathrm{cut} \quad \overset{\mathrm{cut}}{\rightarrow} \quad \dfrac{(\pi_1)\ \Gamma \vdash \Delta \qquad \dfrac{(\pi_2)\ \Pi \vdash \Lambda, \varphi \qquad (\pi_3)\ \varphi, \Sigma \vdash \Xi}{\Pi, \Sigma \vdash \Lambda, \Xi}\,\mathrm{cut}}{\Gamma', \Pi', \Sigma \vdash \Delta', \Lambda', \Xi}\,\rho$$

$$\dfrac{\dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad (\pi_2)\ \Pi \vdash \Lambda}{\Gamma', \Pi' \vdash \Delta', \Lambda', \varphi}\,\rho \qquad (\pi_3)\ \varphi, \Sigma \vdash \Xi}{\Gamma', \Pi', \Sigma \vdash \Delta', \Lambda', \Xi}\,\mathrm{cut} \quad \overset{\mathrm{cut}}{\rightarrow} \quad \dfrac{\dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad (\pi_3)\ \varphi, \Sigma \vdash \Xi}{\Gamma, \Sigma \vdash \Delta, \Xi}\,\mathrm{cut} \qquad (\pi_2)\ \Pi \vdash \Lambda}{\Gamma', \Pi', \Sigma \vdash \Delta', \Lambda', \Xi}\,\rho$$

$$\dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad \dfrac{(\pi_2)\ \varphi, \Pi \vdash \Lambda \qquad (\pi_3)\ \Sigma \vdash \Xi}{\varphi, \Pi', \Sigma' \vdash \Lambda', \Xi'}\,\rho}{\Gamma, \Pi', \Sigma' \vdash \Delta, \Lambda', \Xi'}\,\mathrm{cut} \quad \overset{\mathrm{cut}}{\rightarrow} \quad \dfrac{\dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad (\pi_2)\ \varphi, \Pi \vdash \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\mathrm{cut} \qquad (\pi_3)\ \Sigma \vdash \Xi}{\Gamma, \Pi', \Sigma' \vdash \Delta, \Lambda', \Xi'}\,\rho$$

$$\dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad \dfrac{(\pi_2)\ \Pi \vdash \Lambda \qquad (\pi_3)\ \varphi, \Sigma \vdash \Xi}{\varphi, \Pi', \Sigma' \vdash \Lambda', \Xi'}\,\rho}{\Gamma, \Pi', \Sigma' \vdash \Delta, \Lambda', \Xi'}\,\mathrm{cut} \quad \overset{\mathrm{cut}}{\rightarrow} \quad \dfrac{(\pi_2)\ \Pi \vdash \Lambda \qquad \dfrac{(\pi_1)\ \Gamma \vdash \Delta, \varphi \qquad (\pi_3)\ \varphi, \Sigma \vdash \Xi}{\Gamma, \Sigma \vdash \Delta, \Xi}\,\mathrm{cut}}{\Gamma, \Pi', \Sigma' \vdash \Delta, \Lambda', \Xi'}\,\rho$$

Figure 2.7: Rank-reduction rules for unary/binary inference rules ρ

rename the eigenvariable of the inference, if for example the eigenvariable α occurs as a free variable in Γ or Δ:

$$\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \quad \dfrac{(\pi_2)}{\dfrac{\varphi, \Pi \vdash \Lambda, \psi(\alpha)}{\varphi, \Pi \vdash \Lambda, \forall x\, \psi(x)}\,\forall_r}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut} \quad \overset{\text{cut}}{\longrightarrow} \quad \dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \quad \dfrac{(\pi_2[\alpha \backslash \beta])}{\varphi, \Pi \vdash \Lambda, \psi(\beta)}\,\text{cut}}{\dfrac{\Gamma, \Pi \vdash \Delta, \Lambda, \psi(\beta)}{\Gamma, \Pi \vdash \Delta, \Lambda, \forall x\, \psi(x)}\,\forall_r}$$

Gentzen's proof [43] shows a subtly different result, namely that this is true in a different calculus where we replace the cut inference by a mix inference:

$$\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi, \ldots, \varphi} \quad \dfrac{(\pi_2)}{\varphi, \ldots, \varphi, \Pi \vdash \Lambda}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{mix}$$

However it will be important that we have the result directly on the calculus $\mathsf{LK}(T)$ (i.e., with cut instead of mix) and using the reduction rules shown in Figures 2.2 to 2.7 because the grammars and Herbrand sequents of proofs will be preserved under those reduction rules.

Lemma 2.2.1 ([43]). *Let π_1 and π_2 be cut- and induction-free $\mathsf{LK}(T)$-proofs of the sequents $\Gamma \vdash \Delta, \varphi$ and $\varphi, \Pi \vdash \Lambda$, resp. Then there exists a cut- and induction-free $\mathsf{LK}(T)$-proof π^* such that:*

$$\dfrac{\dfrac{(\pi_1)}{\Gamma \vdash \Delta, \varphi} \quad \dfrac{(\pi_2)}{\varphi, \Pi \vdash \Lambda}}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\mathit{cut} \quad \overset{\text{cut}}{\longrightarrow} \pi^*$$

Proof. See [47, Corollary 2.2] or [26, Theorem 3 and Lemma 16] for versions of this lemma in inessentially different versions of LK. The most significant difference is that our calculus has a theory inference T, but this is an unproblematic addition because we have a reduction for a cut on two T-inferences in Figure 2.3. □

Theorem 2.2.1 ([43]). *Let π be an induction-free $\mathsf{LK}(T)$-proof, then there is an induction-free $\mathsf{LK}(T)$-proof π^* such that $\pi \overset{\text{cut}}{\longrightarrow} \pi^*$.*

Proof. Iteratively eliminate each top-most cut using Lemma 2.2.1. □

We also extend the cut-reduction relation to proofs with induction using an extended reduction relation $\overset{\text{ind}}{\longrightarrow} \supset \overset{\text{cut}}{\longrightarrow}$ as described in Figure 2.8. Whenever the main term of an induction inference is a constructor application, then we can unfold that induction to several cuts and induction inferences whose main terms are the recursive occurrences of the constructor term. This procedure allows us to reduce (some) induction inferences to nested cuts, thereby eliminating them. Clearly we cannot eliminate all induction inferences since the addition of the induction rule is not conservative over first-order logic. For example, the following $\mathsf{LK}(\emptyset)$-proof with a single induction inference proves a first-order statement that is not valid in first-order logic:

$$\dfrac{\dfrac{}{P(0) \vdash P(0)}\,\text{ax} \quad \dfrac{\dfrac{\dfrac{}{P(v) \vdash P(v)}\,\text{ax} \quad \dfrac{}{P(s(v)) \vdash P(s(v))}\,\text{ax}}{P(v), P(v) \to P(s(v)) \vdash P(s(v))}\,{\to_l}}{P(v), \forall x\,(P(x) \to P(s(x))) \vdash P(s(v))}\,{\forall_l}}{P(0), \forall x\,(P(x) \to P(s(x))) \vdash P(\alpha)}\,\text{ind}}{P(0), \forall x\,(P(x) \to P(s(x))) \vdash \forall x\,P(x)}\,{\forall_r}$$

It is nevertheless possible to use such a reduction to eliminate induction from proofs of existential statements, as Gentzen has showed [41, 42].

Example 2.2.1. Let us consider induction-reduction on an example:

$$\dfrac{\begin{matrix}(\pi_1) \\ \vdash P(\mathsf{nil})\end{matrix} \quad \begin{matrix}(\pi_2) \\ P(\alpha_2) \vdash P(\mathsf{cons}(\alpha_1, \alpha_2))\end{matrix}}{\vdash P(\mathsf{cons}(a, \mathsf{nil}))}\,\text{ind}_{\mathsf{list}}$$

In the first step, this proof reduces to a cut on an induction inference with a smaller main term:

$$\cdots \overset{\text{ind}}{\longrightarrow} \dfrac{\dfrac{\begin{matrix}(\pi_1) \\ \vdash P(\mathsf{nil})\end{matrix} \quad \begin{matrix}(\pi_2) \\ P(\alpha_2) \vdash P(\mathsf{cons}(\alpha_1, \alpha_2))\end{matrix}}{\vdash P(\mathsf{nil})}\,\text{ind}_{\mathsf{list}} \quad \begin{matrix}(\pi_2[\alpha_1 \backslash a, \alpha_2 \backslash \mathsf{nil}]) \\ P(\mathsf{nil}) \vdash P(\mathsf{cons}(a, \mathsf{nil}))\end{matrix}}{\vdash P(\mathsf{cons}(a, \mathsf{nil}))}\,\text{cut}$$

In the second step, we can eliminate the remaining induction inference:

$$\cdots \overset{\text{ind}}{\longrightarrow} \dfrac{\begin{matrix}(\pi_1) \\ \vdash P(\mathsf{nil})\end{matrix} \quad \begin{matrix}(\pi_2[\alpha_1 \backslash a, \alpha_2 \backslash \mathsf{nil}]) \\ P(\mathsf{nil}) \vdash P(\mathsf{cons}(a, \mathsf{nil}))\end{matrix}}{\vdash P(\mathsf{cons}(a, \mathsf{nil}))}\,\text{cut}$$

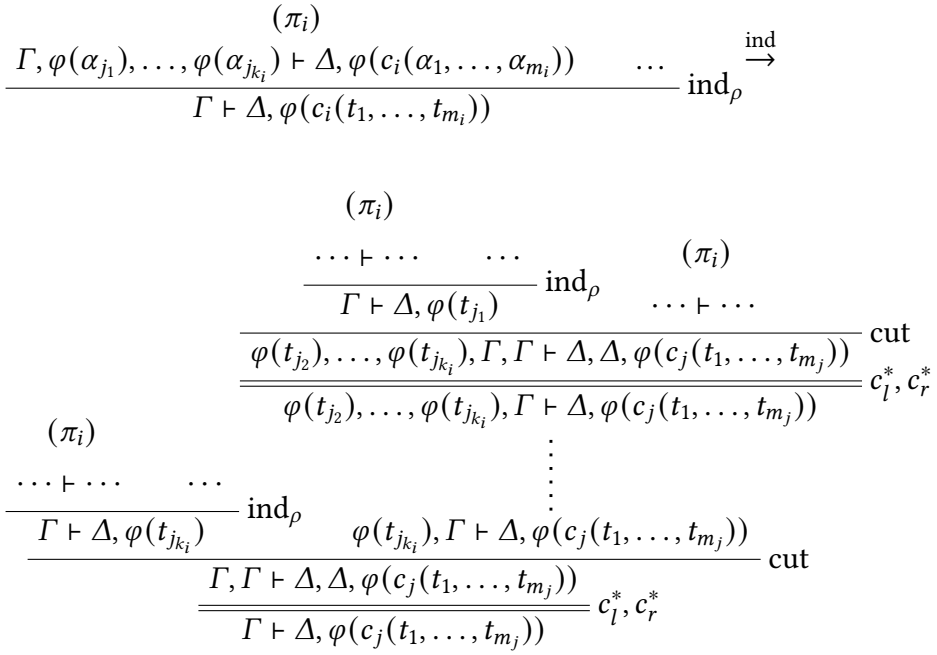


Figure 2.8: Induction-reduction rule.

2.3 Cut-free proofs and tree languages

Cut-free proofs directly contain the quantifier-free instances that constitute a Herbrand sequent. For technical reasons, we only consider proofs of Σ_1-sequents—these are sequents of prenex formulas that are universally quantified in the antecedent and existentially quantified in the succedent. Without these restrictions, we would need to handle eigenvariables or Skolem terms in the instances as well as restrictions that ensure their correct use. This would necessitate a more complicated formalism, such as for example expansion trees [71]. However the restriction to Σ_1-sequents is not a particularly significant restriction: we can always prenexify and Skolemize formulas as well as proofs in first-order logic, resulting in a proof of a Σ_1-sequent.

Theorem 2.3.1. *Let $\Gamma \vdash \Delta$ be a Σ_1-sequent, and π a cut- and induction-free $\mathsf{LK}(T)$-proof of $\Gamma \vdash \Delta$. Then there exist quantifier-free instances Γ' and Δ' of Γ and Δ, resp., such that $\Gamma' \vdash \Delta'$ is a T-tautology.*

Proof. For a detailed proof, see Buss [16]. We first permute the quantifier inferences in the proof down as far as possible. Since the end-sequent is prenex, the resulting proof can be divided into two parts: the lower part which only consists of quantifier inferences (as well as the structural rules of contraction and weakening), and the upper part which does not contain any quantifier rules. The sequent in the middle is a quantifier-free sequent of instances of the end-sequent. And furthermore it is a T-tautology since there is a proof above it. □

Example 2.3.1. Consider the $\mathsf{LK}(\emptyset)$-proof shown in Figure 2.9 with the end-sequent $\forall y\, P(0, y), \forall x \forall y\, (P(x, f(y)) \to P(s(x), y)) \vdash P(s^2(0), c)$. Following the proof of Theorem 2.3.1 we can extract an Herbrand sequent from this proof. The proof above is already of the form required after the rule permutations: the part below the lowest $\to_l$-inference only consists of $\forall_l$ and c_l-inferences and there are no quantifier inferences above it. Hence the conclusion of the lowest $\to_l$-inference is a Herbrand sequent:

$$\begin{aligned} & P(0, f^2(c)), \\ & P(0, f^2(c)) \to P(s(0), f(c)), \\ & P(s(0), f(c)) \to P(s^2(0), c) \\ \vdash\ & P(s^2(0), c) \end{aligned}$$

2.4 Term encoding of formulas

To connect proofs and grammars, as well as Herbrand sequents and languages, we need to convert between formulas and terms. This is because Herbrand sequents contain formulas, and tree grammars generate a set of terms. For instance, recall the Herbrand sequent from Example 2.3.1 (we write $f^2(c)$ as a convenient abbreviation for $f(f(c))$):

$$\begin{aligned} & P(0, f^2(c)), \\ & P(0, f^2(c)) \to P(s(0), f(c)), \\ & P(s(0), f(c)) \to P(s^2(0), c) \\ \vdash\ & P(s^2(0), c) \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(0,f^2(c)) \vdash P(0,f^2(c))}\,\text{ax} \quad \dfrac{\dfrac{}{P(s(0),f(c)) \vdash P(s(0),f(c))}\,\text{ax} \quad \dfrac{}{P(s^2(0),c) \vdash P(s^2(0),c)}\,\text{ax}}{P(s(0),f(c)), P(s(0),f(c)) \to P(s^2(0),c) \vdash P(s^2(0),c)}\,{\to_l}}{P(0,f^2(c)), P(0,f^2(c)) \to P(s(0),f(c)), P(s(0),f(c)) \to P(s^2(0),c) \vdash P(s^2(0),c)}\,{\to_l}}{P(0,f^2(c)), P(0,f^2(c)) \to P(s(0),f(c)), \forall y\,(P(s(0),f(y)) \to P(s^2(0),y)) \vdash P(s^2(0),c)}\,{\forall_l}}{P(0,f^2(c)), P(0,f^2(c)) \to P(s(0),f(c)), \forall x \forall y\,(P(x,f(y)) \to P(s(x),y)) \vdash P(s^2(0),c)}\,{\forall_l}}{P(0,f^2(c)), \forall y\,(P(0,f(y)) \to P(s(0),y)), \forall x \forall y\,(P(x,f(y)) \to P(s(x),y)) \vdash P(s^2(0),c)}\,{\forall_l}}{P(0,f^2(c)), \forall x \forall y\,(P(x,f(y)) \to P(s(x),y)), \forall x \forall y\,(P(x,f(y)) \to P(s(x),y)) \vdash P(s^2(0),c)}\,{c_l}}{P(0,f^2(c)), \forall x \forall y\,(P(x,f(y)) \to P(s(x),y)) \vdash P(s^2(0),c)}\,{\forall_l}}{\forall y\,P(0,y), \forall x \forall y\,(P(x,f(y)) \to P(s(x),y)) \vdash P(s^2(0),c)}\,{\forall_l}$$

Figure 2.9: Proof used in Example 2.3.1.

We want to associate to this Herbrand sequent a set of terms L. One potential choice would be to treat the predicate symbols and logical connectives of H as function symbols, and negate the formulas in the succedent:

$$\begin{aligned}&\{P(0, f^2(c)),\\ &\ P(0, f^2(c)) \to P(s(0), f(c)),\\ &\ P(s(0), f(c)) \to P(s^2(0), c),\\ &\ \neg P(s^2(0), c)\}\end{aligned}$$

However this representation is large and computationally unwieldy, so we encode formulas instances as term by introducing a new function symbol for each formula:

Definition 2.4.1. Let $\forall \overline{x_1}\ \varphi_1, \ldots, \forall \overline{x_m}\ \varphi_m \vdash \exists \overline{x_{m+1}}\ \varphi_{m+1}, \ldots, \exists \overline{x_n}\ \varphi_n$ be a Σ_1-sequent. Then we associate to each formula φ_i a fresh function symbol r_i (whose type is such that $r_i(\overline{x_i})$ has type o).

- The formula instance $\varphi_i[\overline{x_i}\backslash\overline{t}]$ *encodes* to the term $r_i(\overline{t})$, which we write as $E(\varphi_i[\overline{x_i}\backslash\overline{t}]) = r_i(\overline{t})$.
- Terms of the form $r_i(\overline{t})$ for some i where $\overline{t}$ does not contain r_j for any j are called *decodable*

Example 2.4.1. The formula instance $P(s(0), f(c)) \to P(s^2(0), c)$ encodes to the term $E(P(s(0), f(c)) \to P(s^2(0), c)) = r_2(s(0), c)$. The term $r_2(s^9(c), f(s(0)))$ is decodable, but $r_2(r_1, 0)$ and $r_{27}(0)$ are not.

We can now extend this term encoding to sequents as well.

Definition 2.4.2. Let $\Gamma \vdash \Delta$ be a Σ_1-sequent.

- If $\Gamma' \vdash \Delta'$ is a sequent of formula instances of $\Gamma \vdash \Delta$, then $\Gamma' \vdash \Delta'$ encodes to the set of terms $E(\Gamma' \vdash \Delta') = \{E(\varphi) \mid \varphi \in \Gamma' \cup \Delta'\}$.
- A set of terms L is called decodable if all $t \in L$ are decodable.
- Given a decodable set of terms L, we define $D(L)$ as the unique sequent such that $E(D(L)) = L$.

Example 2.4.2. The Herbrand sequent of Example 2.3.1 encodes to the following language: $E(H) = \{r_1(f^2(c)), r_2(0, f(c)), r_2(s(0), c), r_3\}$

We define the language of a cut-free proof to be the encoded Herbrand sequent:

Definition 2.4.3. Let π be a cut- and induction-free $\mathsf{LK}(T)$-proof of a Σ_1-sequent $\Gamma \vdash \Delta$. Then we define the Herbrand language $L(\pi) = E(\Gamma' \vdash \Delta')$ where $\Gamma' \vdash \Delta'$ is the Herbrand sequent as in Theorem 2.3.1.

Example 2.4.3. For the $\mathsf{LK}(\emptyset)$-proof π from Example 2.3.1, we have $L(\pi) = \{r_1(f^2(c)), r_2(0, f(c)), r_2(s(0), c), r_3\}$.

2.5 Vectorial totally rigid acyclic tree grammars

In this section we will define a special class of tree grammars called vectorial totally rigid acyclic tree grammars, or VTRATGs for short. These are different from regular tree grammars (as presented for example in [23]) in two ways: nonterminals are *vectors*, and the derivation relation is highly restricted. The terminology of rigidity goes back to a similar restriction on the derivation relation which was introduced with rigid tree automata [60].

Non-terminals are special variables. We will write $\mathcal{T}(\Sigma \cup X \cup N)$ for the set of terms containing constants (including function symbols) from a signature Σ, variables from X, and nonterminals from N; the set of constants is always disjoint from the set of variables. We write $f/n \in \Sigma$ if the function symbol f has arity n. Non-terminals are special variables, and variables are nullary function symbols.

Definition 2.5.1. Let Σ be a set of function symbols with arity. A VTRATG G is given by a tuple $G = (N, \Sigma, P, A)$:

1. $A \in N$ is the start symbol.

2. N is a finite set of nonterminal vectors. A nonterminal vector is a finite sequence of nonterminals, and a nonterminal is a nullary function symbol. The nonterminals need to be pairwise distinct.

3. Σ is a finite signature such that $\Sigma \cap N = \emptyset$.

4. P is a finite set of vectorial productions. A vectorial production is a pair $\overline{B} \to \overline{t}$, where $\overline{B} \in N$ is a nonterminal vector and $\overline{t} = (t_1, \ldots, t_k)$ is a vector of terms of the same length as as $\overline{B}$, and t_i has the same type as B_i for all $i \leq k$.

5. there exists a strict linear order $\prec$ on the nonterminal vectors such that $\overline{B} \prec \overline{C}$ whenever $\overline{B} \to \overline{t} \in P$ for some $\overline{t}$ and j, k such that t_j contains the nonterminal C_k.

The last condition for the productions expresses the requirement that G is acyclic.

Example 2.5.1. With N, Σ, P defined as follows, $G = (N, \Sigma, P, A)$ is a VTRATG:

$$\begin{aligned} N &= \{(A), (B_1, B_2)\}, \\ \Sigma &= \{f/3, c/0, d/0, e/0\}, \\ P &= \{(A) \to (f(B_1, B_2, B_2)), && (p_1) \\ &\quad (A) \to (f(B_2, B_1, B_1)), && (p_2) \\ &\quad (B_1, B_2) \to (c, d) && (p_3) \\ &\quad (B_1, B_2) \to (d, e)\}. && (p_4) \end{aligned}$$

We will often omit the parentheses for nonterminal vectors that consist of only a single nonterminal, i.e. we write A instead of (A). In addition, we typically only specify the productions of a VTRATG, leaving A, Σ, N implicit. We would then just define G to be the VTRATG with the productions:

$$\begin{aligned} A &\to f(B_1, B_2, B_2) \mid f(B_2, B_2, B_1) \\ (B_1, B_2) &\to (c, d) \mid (d, e) \end{aligned}$$

We will compute the language generated by G in the next section, in Example 2.6.7.

Sometimes we will consider VTRATGs where every nonterminal vector has length one. These non-vectorial VTRATGs are called TRATGs (totally rigid acyclic tree grammars):

Definition 2.5.2. A TRATG is a VTRATG $G = (N, \Sigma, P, A)$ such that all nonterminals vectors $\overline{B} \in N$ are of length $|\overline{B}| = 1$.

There are several reasonable ways to measure the size of a grammar, e.g. we could count the number of symbols in a textual representation. However, for our applications the most natural measurement that we will use by default is the number of productions. The number of productions will correspond the number of quantifier inferences in the simple proof described by the VTRATG (as we will see in Lemma 2.7.4). Counting the numbers of productions is also the size measure used by Bucher [15] in descriptional complexity.

Definition 2.5.3. Let $G = (N, \Sigma, P, A)$ be a VTRATG. Its size $|G| = |P|$ is the number of its productions.

2.6 Derivations in VTRATGs

So far we have only defined VTRATGs, but not their language, that is, the set of terms generated by a VTRATG. A term is generated by a VTRATG if there is a *derivation* of the term. There are several equivalent ways to define derivations (and hence languages of VTRATGs):

($\overset{1}{\Rightarrow}$) A derivation is a finite sequence of terms, such that in every step a nonterminal is replaced by the right-hand side of a production, using at most one production for every nonterminal vector.

($\overset{2}{\Rightarrow}$) A derivation is a finite sequence of terms, such that in every step a nonterminal is replaced by the right-hand side of a production, fulfilling a certain rigidity condition. (This definition only works for TRATGs.)

($\overset{3}{\Rightarrow}$) A derivation is a finite subset of the productions, containing at most one production for every nonterminal vector. The derived term is then given by an iterated substitution using the productions.

Let us begin with the definition using $\overset{1}{\Rightarrow}$: we first define a single-step derivation relation $\overset{1}{\Rightarrow}_p$, which describes replacing a single nonterminal using the

production p. This single-step relation is then extended to the rigid derivation relation $\overset{1}{\Rightarrow}^{r}_{G}$, which is a finite sequence of $\overset{1}{\Rightarrow}_{p}$ steps, using at most one production per nonterminal vector.

Definition 2.6.1. Let $G = (N, \Sigma, P, A)$ be a VTRATG, $p = \overline{B} \to \overline{s}$ a production, and t_1, t_2 terms. Then $t_1 \overset{1}{\Rightarrow}_{p} t_2$ iff there exists a position $q \in \mathrm{Pos}(t_1)$ such that $t_1|_q = B_i$ and $t_2 = t_1|_p[s_i]$ for some i. We define $t_1 \overset{1}{\Rightarrow}_{G} t_2$ iff there exists a production $p \in P$ such that $t_1 \overset{1}{\Rightarrow}_{G} t_2$.

Example 2.6.1 (continuing Example 2.5.1).

$$A \overset{1}{\Rightarrow}_{p_1} f(B_1, B_2, B_2) \overset{1}{\Rightarrow}_{p_3} f(c, B_2, B_2) \overset{1}{\Rightarrow}_{p_3} f(c, B_2, d) \overset{1}{\Rightarrow}_{p_3} f(c, d, d)$$

A set of pairs X is called a partial function if $b = b'$ for all $(a, b) \in X$ and $(a, b') \in X$. Since productions are by definition pairs, a set of productions is a partial function if and only if it contains at most one production for every nonterminal vector.

Example 2.6.2 (continuing Example 2.5.1). $\{p_1, p_3\}$ is a partial function, but $\{p_3, p_4\}$ is not.

Definition 2.6.2. Let $G = (N, \Sigma, P, A)$ be a VTRATG, and $s, t \in \mathcal{T}(\Sigma \cup N)$. Then $s \overset{1}{\Rightarrow}^{r}_{G} t$ iff there exists a sequence $s = t_1 \overset{1}{\Rightarrow}_{p_1} t_2 \overset{1}{\Rightarrow}_{p_2} \ldots \overset{1}{\Rightarrow}_{p_n} t_n = t$ such that the set $\{p_1, \ldots, p_n\} \subseteq P$ is a partial function.

Such a sequence of terms is called a derivation.

Example 2.6.3 (continuing Example 2.5.1). $A \overset{1}{\Rightarrow}^{r}_{G} f(c, d, d)$ since the set of productions $\{p_1, p_3\}$ used in the sequence in Example 2.6.1 is a partial function. Using Definition 2.6.2, we will be able to compute $L(G) = \{t \in \mathcal{T}(\Sigma) \mid A \overset{1}{\Rightarrow}^{r}_{G} t\} = \{f(c, d, d), f(d, c, c), f(d, e, e), f(e, d, d)\}$. Note that $A \overset{1}{\not\Rightarrow}^{r}_{G} f(c, e, e)$, since we would need to use two different productions for $\overline{B}$.

The size of a $\overset{1}{\Rightarrow}$-derivation is bounded by the size of the term and the number of nonterminal vectors in the grammar:

Lemma 2.6.1. *Let $G = (N, \Sigma, P, A)$ be a VTRATG, and $s, t \in \mathcal{T}(\Sigma \cup N)$. Then $n \leq |\mathrm{Pos}(t)||N|$ for any sequence $s = t_1 \overset{1}{\Rightarrow}_{p_1} t_2 \overset{1}{\Rightarrow}_{p_2} \ldots \overset{1}{\Rightarrow}_{p_n} t_n = t$ with $p_i \in P$ for all i.*

Proof. Each of the steps replaces a nonterminal with its right-hand side at a single position. There are at most $|\mathrm{Pos}(t)|$ such positions. It is possible that we have multiple steps that replace at the same position, for example with productions of the form $B \to C$. However due to the acyclicity of G, this can only happen $|N|$ times per position. □

For TRATGs, we can use a slightly more interesting side condition on the sequence of terms $t_1, \ldots, t_n$ in Definition 2.6.2. Instead of characterizing the set of used productions as a partial function, we can give a side condition on the equality of subterms of $t = t_n$: if a nonterminals occurs twice as $t_i|_q = t_{i'}|_{q'}$, then both occurrences expand to the same subterm $t|_q = t|_{q'}$ in t:

Definition 2.6.3. Let $G = (N, \Sigma, P, A)$ be a TRATG, and $s, t \in \mathcal{T}(\Sigma \cup N)$. Then $s \overset{2}{\underset{G}{\Rightarrow}}{}^{r} t$ iff there exists a sequence $s = t_1 \overset{1}{\Rightarrow}_{p_1} t_2 \overset{1}{\Rightarrow}_{p_2} \ldots \overset{1}{\Rightarrow}_{p_n} t_n = t$ such that $p_i \in P$ for all $1 \leq i \leq n$, and: for any $1 \leq i, i' \leq n$ and positions q, q' where $t_i|_q = t_{i'}|_{q'}$ is a nonterminal, we have $t|_q = t|_{q'}$.

The side condition on the sequence in Definition 2.6.3 is called total rigidity and is named after a corresponding notion for regular tree automata [61].

Example 2.6.4. Let G be the TRATG with the productions $p_1 = A \to f(B, B)$, $p_2 = B \to c$ and $p_3 = B \to d$. In the sequence $A \overset{1}{\Rightarrow}_{p_1} f(B, B) \overset{1}{\Rightarrow}_{p_2} f(c, B) \overset{1}{\Rightarrow}_{p_2} f(c, c)$, the nonterminal B occurs three times as $B = t_2|_1 = t_2|_2 = t_3|_2$. The subterms of $f(c, c)|_1 = c$ and $f(c, c)|_2 = c$ are equal, and hence $A \overset{2}{\underset{G}{\Rightarrow}}{}^{r}$. As a counterexample, consider the sequence $A \overset{1}{\Rightarrow}_{p_1} f(B, B) \overset{1}{\Rightarrow}_{p_2} f(c, B) \overset{1}{\Rightarrow}_{p_3} f(c, d)$. It does not fulfill the rigidity condition in Definition 2.6.3: we have $B = t_2|_1 = t_3|_2$ but $f(c, d)|_1 = c \neq f(c, d)|_2 = d$.

Remark 2.6.1. We cannot use this rigidity condition for derivations in VTRATGs as this would result in a different notion of language. Consider the VTRATG G with the productions $A \to f(B, C)$ and $(B, C) \to (d, d) \mid (e, e)$. Then the sequence $A \Rightarrow^2_G f(B, C) \Rightarrow^2_G f(d, C) \Rightarrow^2_G f(d, e)$ fulfills the rigidity condition, but it uses two different productions for the nonterminal vector (B, C) and hence derives a term $f(d, e) \notin L(G)$.

The third way to define derivations, $\overset{3}{\Rightarrow}$, directly characterizes the used productions. We associate to every partial function $P' \subseteq P$ of the produc-

tions a substitution $\sigma_{P'}$. The derivable terms are then the images of the start nonterminal A under these substitutions.

Definition 2.6.4. Let $G = (N, \Sigma, P, A)$ be a VTRATG, and $P' \subseteq P$ a partial function. We can order the nonterminal vectors on the left-hand sides of P' as $\mathrm{dom}(P') = \{\overline{B_1} \prec \overline{B_2} \prec \cdots \prec \overline{B_n}\}$ (where $\prec$ is as in Definition 2.5.1). Then we define $\sigma_{P'} = [\overline{B_1}\backslash\overline{t_1}] \cdots [\overline{B_n}\backslash\overline{t_n}]$ where $\overline{t_i}$ is the unique vector of terms such that $\overline{B_i} \to \overline{t_i} \in P'$ for $1 \leq i \leq n$.

Example 2.6.5 (continuing Example 2.5.1). The nonterminal vectors are ordered as $A \prec \overline{B}$ and we have $\sigma_{\{p_1,p_3\}} = [A\backslash f(B_1, B_2, B_2)][B_1\backslash c, B_2\backslash d]$.

Remark 2.6.2. The reason we use partial functions and not just functions in Definition 2.6.4 is because the VTRATG might contain a nonterminal vector $\overline{B}$ such that there is no production of the form $\overline{B} \to \ldots$. There are nontrivial examples of such VTRATGs: consider for example the VTRATG G with the productions $A \to C$ and $(B, C) \to (D, e)$ (where D is a nonterminal vector, which clearly does not have any associated productions). Then $L(G) = \{e\}$. The term e has the $\Rightarrow_G^3$-derivation $\{A \to C, (B, C) \to (D, e)\}$. We cannot make this partial function total since there is no production for D.

Lemma 2.6.2. *Let $G = (N, \Sigma, P, A)$ be a VTRATG, and $P' \subseteq P$ a partial function. Then $C_j\sigma_{P'} = s_j\sigma_{P'}$ for any j and $\overline{C} \to \overline{s} \in P'$.*

Proof. With $\overline{B_1} \prec \cdots \prec \overline{B_n}$ as in Definition 2.6.4, let i be such that $\overline{C} = \overline{B_i}$ and $s_j = t_{i,j}$. Then:

$$\begin{aligned}
C_j\sigma_{P'} &= C_j[\overline{B_1}\backslash\overline{t_1}] \cdots [\overline{B_n}\backslash\overline{t_n}] \\
&= C_j[\overline{B_2}\backslash\overline{t_2}] \cdots [\overline{B_n}\backslash\overline{t_n}] \\
&\vdots \\
&= C_j[\overline{B_i}\backslash\overline{t_i}] \cdots [\overline{B_n}\backslash\overline{t_n}] \\
&= s_j[\overline{B_{i+1}}\backslash\overline{t_{i+1}}] \cdots [\overline{B_n}\backslash\overline{t_n}] \\
&= s_j[\overline{B_i}\backslash\overline{t_i}] \cdots [\overline{B_n}\backslash\overline{t_n}] \\
&\vdots \\
&= s_j[\overline{B_1}\backslash\overline{t_1}] \cdots [\overline{B_n}\backslash\overline{t_n}] \\
&= s_j\sigma_{P'}
\end{aligned}$$

The substitutions $[\overline{B_k}\backslash\overline{t_k}]$ are the identity on C_j for $k < i$, and the identity on s_j for $k \leq i$, since $\overline{B_k}$ and $\mathrm{FV}(C_j)$ (resp., $\mathrm{FV}(s_j)$) are disjoint. □

Definition 2.6.5. Let $G = (N, \Sigma, P, A)$ be a VTRATG, and $t \in \mathcal{T}(\Sigma)$. Then $A \overset{3}{\Rightarrow}_G t$ iff there exists a partial function $P' \subseteq P$ such that $t = A\sigma_{P'}$.

Example 2.6.6 (continuing Example 2.6.5). $A \overset{3}{\Rightarrow}_G f(c,d,d)$ since $A\sigma_{\{p_1,p_3\}} = f(c,d,d)$.

Lemma 2.6.3. *Let P' be a set of productions that is a partial function, and $t \overset{1}{\Rightarrow}_p s$ where $p \in P'$. Then $t\sigma_{P'} = s\sigma_{P'}$.*

Proof. Let $\overline{B} \to \overline{r} \in P'$ and j, q be such that $t|_q = B_j$ and $t|_q[r_j] = s$. It suffices to show that $t|_q\sigma_{P'} = s|_q\sigma_{P'}$ since that is the only common position at which the two terms differ. This is the case iff $B_j\sigma_{P'} = s_j\sigma_{P'}$, which is true by Lemma 2.6.2. □

Theorem 2.6.1. *Let $G = (N, \Sigma, P, A)$ be a VTRATG, and $t \in \mathcal{T}(\Sigma)$ a term. Then the following are equivalent:*

1. $A \overset{1}{\Rightarrow}^{r}_G t$
2. $A \overset{3}{\Rightarrow}_G t$

Proof. $1 \Rightarrow 2$. By Definition 2.6.2, there exists a sequence $A = t_1 \overset{1}{\Rightarrow}_{p_1} t_2 \overset{1}{\Rightarrow}_{p_2} \ldots \overset{1}{\Rightarrow}_{p_n} t_n = t$ such that the set $P' = \{p_1, \ldots, p_n\} \subseteq P$ is a partial function. By Lemma 2.6.3, $t_i\sigma_{P'} = t_{i+1}\sigma_{P'}$ for any $1 \leq i \leq n$ and hence $A\sigma_{P'} = t_n\sigma_{P'} = t$.

$2 \Rightarrow 1$. By Definition 2.6.5, there exists a partial function $P' \subseteq P$ such that $t = A\sigma_{P'} = [\overline{B_1}\backslash\overline{s_1}] \cdots [\overline{B_n}\backslash\overline{s_n}]$. Define a sequence $t_1 \cdots t_n$ that performs this substitution using $\overset{1}{\Rightarrow}$-steps with the productions in P'. □

Theorem 2.6.2. *Let $G = (N, \Sigma, P, A)$ be a TRATG, and $t \in \mathcal{T}(\Sigma)$ a term. Then the following are equivalent:*

1. $A \overset{1}{\Rightarrow}^{r}_G t$
2. $A \overset{2}{\Rightarrow}^{r}_G t$

3. $A \overset{3}{\Rightarrow}_G t$

Proof. We have already shown $1 \Leftrightarrow 3$ in Theorem 2.6.1.

$1 \Rightarrow 2$. A $\overset{1}{\Rightarrow}$-derivation already fulfills the total rigidity condition of Definition 2.6.5 since $t_i|_q\sigma_{P'} = t|_q$ for any i and position q by Lemma 2.6.3.

$2 \Rightarrow 3$. By Definition 2.6.3, there exists a sequence $A = t_1 \overset{1}{\Rightarrow}_{p_1} t_2 \overset{1}{\Rightarrow}_{p_2} \dots \overset{1}{\Rightarrow}_{p_n} t_n = t$ such that $t|_q = t|_{q'}$ whenever $t_i|_q = t_{i'}|_{q'}$ is a nonterminal. Hence we can define a substitution ρ in such a way that $t_i|_q\rho = t|_q$ for all i and positions $q \in \mathrm{Pos}(t_i)$. Let $P' \subseteq \{p_1, \dots, p_n\}$ be a partial function that contains a production for every nonterminal occurring in $t_1, \dots, t_n$. Let $N = \{B_1 \prec B_2 \prec \cdots \prec B_k\}$. We now show that $B_j\rho = B_j\delta_{P'}$ for every j by reverse induction on j, that is, starting with $j = k$. Let $B_j \to s_j \in P'$. Then there is an i such that $t_i \overset{1}{\Rightarrow}_{B_j \to s_j} t_{i+1}$ and hence $B_j\rho = s_j\rho$. We also have $B_j\sigma_{P'} = s_j\sigma_{P'}$ since $B_j \to s_j \in P'$. Now $s_j\rho = s_j\sigma_{P'}$ by the induction hypothesis since s_j only contains nonterminals C such that $B_j \prec C$. Thus $B_j\rho = B_j\sigma_{P'}$ by transitivity. Finally we have $A\sigma_{P'} = A\rho = t$ and hence $A \overset{3}{\Rightarrow} t$. □

Theorems 2.6.1 and 2.6.2 show that all the three ways to define derivations are equivalent in the sense that they allows us to derive the same set of terms. Formally, we pick $\overset{3}{\Rightarrow}$ as the official definition of derivation:

Definition 2.6.6. Let $G = (N, \Sigma, P, A)$ a VTRATG. A derivation δ in G is a subset $\delta \subseteq P$ of the productions such that δ is a partial function and $A\sigma_{P'} \in \mathcal{T}(\Sigma)$. The derivation δ derives the term $A\sigma_{P'}$.

We will often implicitly use the derivation δ as a substitution: that is, we write $A\delta$ as an abbreviation for $A\sigma_\delta$.

Definition 2.6.7. Let $G = (N, \Sigma, P, A)$ be a VTRATG. The language generated by G is the set of derivable terms $L(G) = \{t \in \mathcal{T}(\Sigma) \mid A \overset{3}{\Rightarrow}_G t\}$.

Example 2.6.7 (continuing Example 2.5.1). We can compute the language of G as follows: $L(G) = \{f(c,d,d), f(d,e,e), f(d,c,c), f(e,d,d)\}$.

The languages of VTRATGs are finite. In fact the language of a VTRATG contains at most exponentially more terms than the number of productions in the VTRATG:

Lemma 2.6.4. *Let $G = (N, \Sigma, P, A)$ be a VTRATG, and for every $\overline{B} \in N$ define $P_{\overline{B}} = \{p \in P \mid \exists \overline{t}\ p = \overline{B} \to \overline{t}\}$. Then $|L(G)| \leq \prod_{\overline{B} \in N, P_{\overline{B}} \neq \emptyset} |P_{\overline{B}}|$.*

Proof. By counting the derivations: observe that if $\delta \subseteq \delta'$ for two derivations δ, δ' in G, then $A\delta = A\delta'$. Hence we can assume that a derivation δ is defined for all nonterminal vectors $\overline{B}$ such that $P_{\overline{B}} \neq \emptyset$. Hence we need to count the number of subsets $\delta \subseteq P$ that are functions with domain $\{\overline{B} \in N \mid P_{\overline{B}} \neq \emptyset\}$. This number is exactly the stated bound. □

If a VTRATG contains a production for every nonterminal vector, then we can also give another equivalent definition of $L(G)$:

Lemma 2.6.5. *Let $G = (N, \Sigma, P, A_0)$ be a VTRATG with the nonterminals $N = \{\overline{A_0} \prec \overline{A_1} \prec \cdots \prec \overline{A_n}\}$ such that for every i there is a production $\overline{A_i} \to \overline{t} \in P$ for some t. Then:*

$$L(G) = \{A_0[\overline{A_0}\backslash\overline{s_0}][\overline{A_1}\backslash\overline{s_1}]\cdots[\overline{A_n}\backslash\overline{s_n}] \mid \forall i\ \overline{A_i} \to \overline{s_i} \in P\}$$

Proof. The substitution is $\sigma_{P'}$ where $P' \subseteq P$ such that P' is a total function. Thus the right-hand side is included in $L(G)$. Since $A\delta = A\delta'$ for derivations $\delta \subseteq \delta'$ and there is a production for every nonterminal vector, we can extend any derivation to a total function. Therefore $L(G)$ is also included in the right-hand side. □

2.7 Grammars for simple proofs

We will now look at the class of proofs described by VTRATGs. The main restriction is that the cut formulas in the proofs may not contain quantifier alternations. In addition, we require that the cut formulas are prenex. Formally, we call this class of proofs the simple proofs:

Definition 2.7.1. A *simple proof* is an induction-free $\mathsf{LK}(T)$-proof such that:

1. All cut formulas are of the form $\forall \overline{x}\ \varphi(\overline{x})$ or $\exists \overline{x}\ \varphi(\overline{x})$ where $\varphi(\overline{x})$ is quantifier-free.
2. The end-sequent is a Σ_1-sequent.

3. Whenever a formula with strong quantifiers occurs in the proof then it is immediately preceded by an $\forall_r$ or $\exists_l$ inference.

4. Every strong quantifier inference ($\forall_r$, $\exists_l$) has a different eigenvariable, none of the eigenvariables occur in the end-sequent.

5. No weakening inferences are applied to quantified formulas, unless the weakened formula is an ancestor of a cut-formula or a formula in the end-sequent.

Only the first restriction is significant. Restricting the logical complexity of the cut formulas has an effect on proof size. While we can of course always reduce cuts of higher complexity to the cuts allowed in simple proofs, this incurs a size increase.

The second restriction on the end-sequent simplifies the theory since we do not have to deal with eigenvariables from the end-sequent. We can always Skolemize and prenexify the end-sequent, producing an equi-satisfiable sequent.

The restriction on the strong quantifiers. It only has a local effect on the cut inferences because the only strong quantifiers in the proof occur in the cuts. It means that the cuts are of the following form:

$$\dfrac{\dfrac{\Gamma \vdash \Delta, \varphi(\overline{\alpha})}{\Gamma \vdash \Delta, \forall \overline{x}\, \varphi(\overline{x})}\,\forall_r^* \qquad \forall \overline{x}\, \varphi(\overline{x}), \Pi \vdash \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\,\text{cut}$$

That is, the cut inference always occurs together with the $\forall_r$-block (or $\exists_l$-block) as an inseparable package. This also simplifies the correspondence between proofs and grammars, because now there is only one vector of eigenvariables for each cut. The following cut is hence forbidden:

$$\dfrac{\dfrac{\dfrac{\psi \vdash \varphi(\alpha)}{\psi \vdash \forall x\, \varphi(x)}\,\forall_r \qquad \dfrac{\theta \vdash \varphi(\beta)}{\theta \vdash \forall x\, \varphi(x)}\,\forall_r}{\psi \wedge \theta \vdash \forall x\, \varphi(x)}\,\wedge_l \qquad \forall x\, \varphi(\overline{x}) \vdash \chi}{\psi \wedge \theta \vdash \chi}\,\text{cut}$$

The restriction on the eigenvariables is a purely technical one, known as "regularity" in the literature. (We will avoid this terminology here to avoid

confusion with another interesting kind of regularity, namely that a family of instance proofs comes from a proof with induction.) Requiring the eigenvariables to be different means that we can reuse them as the nonterminals in the VTRATG that corresponds to the proof. We can always rename the eigenvariables to fulfill this condition.

The last restriction on weakening inferences on quantified formulas ensures that the number of quantifier inferences in the proof corresponds to the number of productions in the grammar (up to a constant factor). We can always ensure this restriction by permuting the weakening inferences downwards, this only decreases the size of the proof. Another way to satisfy this restriction is to always instantiate all quantifiers of a formula at once (as if the quantifier block was a single quantifier), with no other inferences in between. An example of a forbidden weakening is the following:

$$\dfrac{\dfrac{\Gamma \vdash \Delta}{\forall y\, \varphi(c, y), \Gamma \vdash \Delta}\, w_l}{\forall x \forall y\, \varphi(x, y), \Gamma \vdash \Delta}\, \forall_l$$

Lemma 2.7.1. *Let π be a simple proof and $\pi \xrightarrow{\text{cut}} \pi'$, then π' is also a simple proof (up to a potential renaming of eigenvariables).*

Proof. Each of the restrictions is preserved under cut-reduction, except for the condition on the eigenvariables. This may be violated in a reduction of a contraction inference, which duplicates a subproof. □

In Section 2.4 we introduced the isomorphism between a subset of terms and formula instances. We called the terms that encode formula instances decodable (Definition 2.4.1). We now extend this notion to VTRATGs:

Definition 2.7.2. Let S be a Σ_1-sequent, and $G = (N, \Sigma, A, P)$ be a VTRATG. Then G is called decodable iff all the following are true:

- A has type o
- For every production $A \to t \in P$, the right-hand side t is decodable.
- For every production $\overline{B} \to \overline{t} \in P$ where $\overline{B} \neq A$, the right-hand side $\overline{t}$ does not contain r_i for any i.

Example 2.7.1. The VTRATG with the productions $\{A \to r_1(B) \mid r_2(f(B)), B \to c \mid d\}$ is decodable, the one with the productions $\{A \to B, B \to r_2(c)\}$ is not (even though its language $\{r_2(c)\}$ would be decodable).

One consequence of this definition is that the language of a decodable VTRATG is decodable:

Lemma 2.7.2. *Let G be a decodable VTRATG, then $L(G)$ is decodable as well.*

We can now assign to every simple proof a VTRATG containing the data of the quantifier inferences:

Definition 2.7.3. Let π be a simple proof. We define the VTRATG $G(\pi)$ with the start symbol A containing the following productions:

- For every quantified cut with the eigenvariables $\overline{\alpha}$ and a weak quantifier instance of the cut-formula with the terms $\overline{t}$, we add the production $\overline{\alpha} \to \overline{t}$: (and analogously for existential formulas)

$$\dfrac{\dfrac{\vdots}{\cdots \vdash \cdots, \varphi(\overline{\alpha})}}{\cdots \vdash \cdots, \forall \overline{x}\, \varphi(\overline{x})}\,\forall^*_r \qquad \dfrac{\dfrac{\vdots}{\varphi(\overline{t}), \cdots \vdash \cdots}}{\forall \overline{x}\, \varphi(\overline{x}), \cdots \vdash \cdots}\,\forall^*_l, \cdots \quad \vdots \quad \forall \overline{x}\, \varphi(\overline{x}), \cdots \vdash \cdots}{\cdots \vdash \cdots}\,\text{cut}$$

- For every instance $\varphi(\overline{t})$ of a formula of the end-sequent, we add the production $A \to E(\varphi(\overline{t}))$: (and analogously for existential formulas)

$$\dfrac{\dfrac{\vdots}{\varphi(\overline{t}), \cdots \vdash \cdots}}{\forall \overline{x}\, \varphi(\overline{x}), \cdots \vdash \cdots}\,\forall^*_l, \cdots \quad \vdots \quad \forall \overline{x}\, \varphi(\overline{x}), \cdots \vdash \cdots$$

Example 2.7.2. Consider the following $\mathsf{LK}(\emptyset)$-proof of the sequent $\forall x\, P(x) \vdash P(f(c))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(f(\alpha)) \vdash P(f(\alpha))}\,\text{ax}}{\forall x\, P(x) \vdash P(f(\alpha))}\,\forall_l}{\forall x\, P(x) \vdash \forall x\, P(f(x))}\,\forall_r \quad \dfrac{\dfrac{}{P(f(c)) \vdash P(f(c))}\,\text{ax}}{\forall x\, P(f(x)) \vdash P(f(c))}\,\forall_l}{\forall x\, P(x) \vdash P(f(c))}\,\text{cut}$$

The VTRATG $G(\pi)$ as defined by Definition 2.7.3 then has the following productions:

$$A \to r_1(f(\alpha)) \mid r_2$$
$$\alpha \to c$$

The language it generates is $L(G(\pi)) = \{r_1(f(c)), r_2\}$. This language decodes to the Herbrand sequent $P(f(c)) \vdash P(f(c))$.

It is easy to see that $G(\pi)$ is decodable:

Lemma 2.7.3. *Let π be a simple proof. Then $G(\pi)$ is decodable.*

The size of $G(\pi)$ is proportional to the number of quantifier inferences $|\pi|_q$, where the constant factor only depends on the end-sequent and the maximum size of a non-terminal vector in $G(\pi)$. The difference is due to non-terminal vectors: one production like $(B_1, B_2) \to (t, s)$ typically corresponds to two quantifier inferences in the proof. The bound of the following theorem can be made precise by using more sophisticated size measures [32].

Lemma 2.7.4. *Let π be a simple proof, and $G(\pi) = (N, \Sigma, P, A)$. Then $|G(\pi)| - D \leq |\pi|_q \leq |G(\pi)| \cdot C \cdot \max_{B \in N} |B|$, where C is the maximum number of quantifiers in a formula of the end-sequent of π, and D is the number of quantifier-free formulas in the end-sequent.*

Let us conclude by stating the main theorem connecting simple proofs and their corresponding VTRATGs, namely that cut-reduction is compatible with language generation:

Lemma 2.7.5 ([46])**.** *Let π, π' be simple proofs such that $\pi \overset{\text{cut}}{\longrightarrow} \pi'$, then $L(G(\pi)) \supseteq L(G(\pi'))$. If $\pi \overset{\text{ne}}{\to} \pi'$, then $L(G(\pi)) = L(G(\pi'))$.*

Corollary 2.7.1. *Let π be a simple proof, then $L(G(\pi))$ is a T-tautology.*

This correspondence has since been extended to a larger class of proofs, with no restrictions on the quantifier complexity of the cut formulas [2]. However as the cuts become more complicated, the grammars used to describe the quantifier inferences need to become more complicated as well. The class of grammars used for the general case is a special form of higher-order recursion schemes.

Theorem 2.7.1 ([2, Lemma 7.2])**.** *For every induction-free proof π of a Σ_1-sequent such that all cut formulas in π are prenex there is a higher-order recursion scheme $G(\pi)$, where this function G has the property that $L(G(\pi)) \supseteq L(G(\pi'))$ whenever $\pi \overset{\text{cut}}{\longrightarrow} \pi'$.*

2.8 Grammars for simple induction proofs

2.8.1 Simple induction problems

We can also assign grammars to another class of proofs, namely some proofs containing induction of the following kind of sequents:

Definition 2.8.1. A *simple induction problem* is a sequent $\Gamma \vdash \forall x\, \varphi(x)$ where Γ is a list of universally quantified prenex formulas, $\varphi(x)$ quantifier-free, and the quantifier $\forall x$ ranges over an inductive sort ρ.

Example 2.8.1. $\forall y\, P(0, y), \forall x \forall y\, (P(x, f(y)) \to P(s(x), y)) \vdash \forall x\, P(x, c)$

For technical reasons, we only consider the case of a single universal quantifier in the conclusion. We can still treat problems that would naturally be stated using multiple quantifiers by instantiating all but one quantifier with fresh constants: e.g. for commutativity we get the simple induction problem $\Gamma \vdash \forall x\, x + c = c + x$.

Given such a simple induction problem $\Gamma \vdash \forall x\, \varphi(x)$, we will consider instance problems $\Gamma \vdash \varphi(t)$ for terms t. If these terms are built from constructors, then we will be able to unfold a proof with induction of $\Gamma \vdash \forall x\, \varphi(x)$ into a proof with cuts. The following definition makes this notion concrete:

Definition 2.8.2. Let ρ be an inductive type with constructors $c_1, \ldots, c_n$. The set of *constructor terms* of type ρ is the smallest set of terms containing for each constructor c_i all terms $c_i(r_1, \ldots, r_{i_n})$ whenever it contains all r_j where j is a recursive occurrence in c_i. A *free constructor term* is a constructor term where all subterms of a type other than ρ are pairwise distinct fresh constants. We denote the set of free constructor terms by C.

Example 2.8.2. For natural numbers, the terms $0, s(0), s(s(0))$ are free constructor terms, but $s(x)$ is not; all constructor terms are already free constructor terms. If we consider lists of natural numbers with the constructors nil and cons, then $\text{nil}, \text{cons}(a_1, \text{nil}), \text{cons}(a_1, \text{cons}(a_2, \text{nil}))$ are free constructor terms, but $\text{cons}(x + x, \text{cons}(x, \text{nil}))$ is a constructor term that is not a free constructor term.

Definition 2.8.3. Let $\Gamma \vdash \forall x\ \varphi(x)$ be a simple induction problem, and t a free constructor term of type ρ. Then $\Gamma \vdash \varphi(t)$ is the *instance problem for the parameter* t.

Example 2.8.3. The Example 2.3.1 that we considered earlier was an instance problem for Example 2.8.1.

2.8.2 Simple induction proofs

We can now define the class of simple induction proofs, these consist of a single induction followed by a cut. Note that the end-sequents of the proofs π_i and π_c are T-tautologies.

Definition 2.8.4. Let ρ be an inductive type, $\Gamma \vdash \forall x\ \varphi(x)$ a simple induction problem, $\psi(x, w, \overline{y})$ a quantifier-free formula, $\Gamma_1, \ldots, \Gamma_n, \Gamma_c$ quantifier-free instances of Γ, and $\overline{t_{i,j,k}}, \overline{u_k}$ term vectors. Then a *simple induction proof* is a proof π of the following form, where $\pi_1, \ldots, \pi_n, \pi_c$ are cut-free proofs:

$$\dfrac{\dfrac{\begin{array}{ccc}(\pi_1') & & (\pi_n') \\ \vdots & \cdots & \vdots \\ \Gamma \vdash \forall \overline{y}\ \psi(\alpha, \alpha, \overline{y}) & & \end{array}\ \text{ind}_\rho \qquad \dfrac{(\pi_c)}{\Gamma_c, \psi(\alpha, \alpha, \overline{u_k}), \cdots \vdash \varphi(\alpha)}}{\Gamma, \forall \overline{y}\ \psi(\alpha, \alpha, \overline{y}) \vdash \varphi(\alpha)}\ \forall_l^*, w_l^* \quad \text{cut}, c_l^*}{\Gamma \vdash \varphi(\alpha)}\ \forall_r$$

$$\text{where} \quad \pi_i' \quad = \quad \dfrac{\dfrac{(\pi_i)}{\Gamma_i, \psi(\alpha, v_{i,i_l}, \overline{t_{i,i_l,k}}), \cdots \vdash \psi(\alpha, c_i(\overline{v_i}), \overline{\gamma})}}{\Gamma, \forall \overline{y}\, \psi(\alpha, v_{i,i_l}, \overline{y}), \cdots \vdash \forall \overline{y}\, \psi(\alpha, c_i(\overline{v_i}), \overline{y})}\, \forall_l^*, w_l^*, c_l^*, \forall_r^*$$

Example 2.8.4. We consider a simple induction proof of Example 2.8.1 with the induction formula $\forall y\, \psi(\alpha, v, y)$ where $\psi(\alpha, v, y) = P(v, \gamma)$. Formally the proof contains the following instances and terms:

$$\begin{aligned} \Gamma_2 &= \{P(v, f(\gamma)) \to P(s(v), \gamma)\} \\ \Gamma_1 &= \{P(0, \gamma)\} \\ \Gamma_c &= \emptyset \\ t_{2,1,1} &= f(\gamma) \\ u_1 &= c \end{aligned}$$

Lemma 2.8.1. *Let $\Gamma \vdash \forall x\, \varphi(x)$ be a simple induction problem. If there exists a simple induction proof π for the simple induction problem, then for every t there exists a first-order proof π_t for the instance problem with parameter t.*

Proof. Inspecting Definition 2.8.4, there is a subproof π' of π with the end-sequent $\Gamma \vdash \varphi(\alpha)$. By substitution we obtain a proof $\pi'' = \pi'[\alpha \backslash t]$ of $\Gamma \vdash \varphi(t)$. This proof π'' contains a single induction inference with the main term t. Since t is a free constructor term, we can unfold the induction inference in π'' in at most $|t|$ steps using the $\overset{\text{ind}}{\to}$ reduction, yielding $\pi'' \overset{\text{ind}}{\to} \pi_t$. □

2.8.3 Induction grammars

Just as sets of terms describe the quantifier inferences in proofs of the sequents of the instance problems (via their decoding to Herbrand sequents), and VTRATG describe the quantifier inferences in the instance proofs with cuts, we use *induction grammars* to describe the quantifier inferences in the simple induction proof.

Definition 2.8.5. An *induction grammar* $G = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$ consists of:

1. the start nonterminal τ of type o,
2. a nonterminal α whose type ρ is an inductive sort,

3. a family of nonterminal vectors $(\overline{\nu_c})_c$, such that for each constructor c of the inductive sort ρ the term $c(\overline{\nu_c})$ is well-typed,

4. a nonterminal vector $\overline{\gamma}$, and

5. a set of vectorial productions P, where each production is of the form $\tau \to t[\alpha, \overline{\nu_i}, \overline{\gamma}]$ or $\overline{\gamma} \to \overline{t}[\alpha, \overline{\nu_i}, \overline{\gamma}]$ for some i.

Example 2.8.5. The induction grammar corresponding the simple induction proof in Example 2.8.4 has the following productions:

$$\tau \to r_1(\gamma) \mid r_2(\nu, \gamma) \mid r_3$$
$$\gamma \to f(\gamma) \mid c$$

An induction grammar generates a family of languages: each constructor term induces a language.

Definition 2.8.6. A *family* $(L_t)_{t \in I}$ *of languages* is a function from a set of free constructor terms I to languages.

We will not directly define derivations and the generated language for induction grammars. Instead we will define an instantiation operation that results in a VTRATG, and define the language of the induction grammar as the language of the VTRATG obtained via instantiation. The instantiation operation depends on a constructor term r as parameter, in the same way as the instance problem $\Gamma \vdash \varphi(r)$ uses a constructor term. Instantiation of an induction grammar into a VTRATG closely mirrors how the induction rule is unfolded into a series of nested cuts in Lemma 2.8.1.

Definition 2.8.7. Let $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ be an induction grammar, and r a constructor term of the same type as α. The *instance grammar* $I(G, r) = (\tau, N, P')$ is a VTRATG with nonterminal vectors $N = \{\tau\} \cup \{\overline{\gamma_s} \mid s \trianglelefteq r\}$ and productions $P' = \{p' \mid \exists p \in P \ (p \rightsquigarrow p')\}$. The instantiation relation $p \rightsquigarrow p'$ is defined as follows:

- $\tau \to t[\alpha, \overline{\nu_i}, \overline{\gamma}] \ \rightsquigarrow \ \tau \to t[r, \overline{s}, \overline{\gamma_{c_i(\overline{s})}}]$ for $c_i(\overline{s}) \trianglelefteq r$
- $\overline{\gamma} \to \overline{t}[\alpha] \ \rightsquigarrow \ \overline{\gamma_s} \to \overline{t}[r]$ for $s \trianglelefteq r$

- $\overline{\gamma} \to \overline{t}[\alpha, \overline{v_i}, \overline{\gamma}] \;\leadsto\; \overline{\gamma_{s_j}} \to \overline{t}[r, \overline{s}, \overline{\gamma_{c_i(\overline{s})}}]$ for $c_i(\overline{s}) \trianglelefteq r$,
  where j is a recursive occurrence in c_i

Example 2.8.6. Let us instantiate the induction grammar in Example 2.8.5 with the parameter $s(s(0))$. The instance grammar $I(G, s(s(0)))$ has the nonterminals τ, γ_0, $\gamma_{s(0)}$, and $\gamma_{s(s(0))}$, and contains all productions on the right-hand side (we use the abbreviation $p \rightsquigarrow p'_1 \mid p'_2$ for $p \rightsquigarrow p'_1 \wedge p \rightsquigarrow p'_2$).

$$\begin{aligned}
\tau \to r_1(\gamma) \quad &\rightsquigarrow \quad \tau \to r_1(\gamma_0) \mid \tau \to r_1(\gamma_{s(0)}) \mid \tau \to r_1(\gamma_{s(s(0))}) \\
\tau \to r_2(\nu, \gamma) \quad &\rightsquigarrow \quad \tau \to r_2(0, \gamma_{s(0)}) \mid \tau \to r_2(s(0), \gamma_{s(s(0))}) \\
\gamma \to f(\gamma) \quad &\rightsquigarrow \quad \gamma_0 \to f(\gamma_{s(0)}) \mid \gamma_{s(0)} \to f(\gamma_{s(s(0))}) \\
\gamma \to c \quad &\rightsquigarrow \quad \gamma_0 \to c \mid \gamma_{s(0)} \to c \mid \gamma_{s(s(0))} \to c
\end{aligned}$$

We can now define the language in terms of the instance grammar. There is a different language for each constructor term. Alternatively, we can also think of the induction grammar generating a family of languages.

Definition 2.8.8. Let $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ be an induction grammar, and t a constructor term of the same type as α. Then we define the language for the parameter t as $L(G, t) = L(I(G, t))$.

Example 2.8.7. The induction grammar in Example 2.8.5 produces the following language for the parameter $s(s(0))$, which decodes to a tautology:

$$\begin{aligned}
L(G, s(s(0))) = \{&r_1(c), r_1(f(c)), r_2(f(f(c)), \\
&r_2(0, c), r_2(0, f(c)), r_2(s(0), c), \\
&r_3\}
\end{aligned}$$

In Example 2.3.1 we have already seen an instance proof of the sequent of our running example. Its Herbrand sequent uses only a subset of the instances here. We have $E(H) \subseteq L(G, s(s(0)))$ for the encoded set of terms we computed in Example 2.4.2.

Recall that $L(G(\pi), t)$ is a set of terms that encodes a Herbrand sequent for the instance problem with parameter t. The computation of the Herbrand sequent on the term-level using grammars closely mirrors the Herbrand sequents obtained via induction-unfolding and cut-elimination on the level of

proofs (that is, $L(\pi_t^*)$): the language generated by the grammar is a superset of the language extracted from the proof. That $L(G(\pi), t)$ is (in general) a strict superset of $L(\pi_t^*)$ (and not exactly equal) is a subtlety introduced by weakening inferences in π: quantifier inferences may be deleted when reducing weakening inferences during cut-elimination. But the grammar still generates these deleted terms, since it intentionally abstracts away from the propositional reasoning of the proof.

Theorem 2.8.1. *Let π be a simple induction proof and t an instance term. Then $L(G(\pi), t) \supseteq L(\pi_t^*)$.*

$$\begin{array}{ccccc} \pi & \xrightarrow{\text{ind}} & \pi_t & \xrightarrow{\qquad\text{cut}\qquad} & \pi_t^* \\ \downarrow & & \downarrow & & \downarrow \\ G(\pi) & \longmapsto & G(\pi_t) & \longmapsto & L(G(\pi), t) \supseteq L(\pi_t^*) \end{array}$$

Proof. Unfolding the induction in π yields a proof π_t without induction as in Lemma 2.8.1. The instantiation relation in Definition 2.8.7 is chosen so that $I(G(\pi), t) \supseteq G(\pi_t)$. Then $L(G(\pi), t) = L(I(G(\pi), t)) \supseteq L(G(\pi_t)) \supseteq L(\pi_t^*)$ for any π_t^* such that $\pi_t \overset{\text{cut}}{\rightarrow} \pi_t^*$ by Lemma 2.7.5. □

In particular, Theorem 2.8.1 tells us that induction grammars extracted from proofs produce tautological languages for every instance:

Definition 2.8.9. A family of languages $(L_t)_{t\in I}$ is called (T-)*tautological* if L_t is T-tautological for every t. An induction grammar G is called *tautological* if $(L(G, t))_{t\in C}$ is tautological, i.e., the language $L(G, t)$ is T-tautological for every constructor term t.

Corollary 2.8.1 (of Theorem 2.8.1)**.** *Let π be a simple induction proof. Then the induction grammar $G(\pi)$ is tautological.*

Similar to Lemma 2.7.4 for the VTRATG of a simple proof, the quantifier complexity of the simple induction proof π is related to the number of productions $|G(\pi)|$ in the induction grammar:

Lemma 2.8.2. *Let π be a simple induction proof, and $G(\pi) = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$. Then $|G(\pi)| - D \leq |\pi|_q \leq C \cdot |G(\pi)| \cdot |\overline{\gamma}|$, where C and D are constant that only depends on the end-sequent of π.*

2.9 Reversing cut- and induction-elimination

In the previous sections we have seen that we can use grammars to compute Herbrand sequents for proofs. For a proof with Π_1-cuts we can extract a VTRATG whose language then decodes to a Herbrand sequent. For a simple induction proof we can extract an induction grammar that for every parameter generates a language that decodes to a Herbrand sequent.

An interesting application is to reverse this process. On the level of the reduction relations of cut- and induction-elimination it is virtually hopeless to reverse the reduction, even going back one step there are infinitely many proofs that reduce in one step to the current one. Undoing the reduction of a weakening inference, we would even need to choose an arbitrary subproof.

The concept that connects grammars and proofs is the one of formula equations, which will collect the conditions necessary to produce a proof for a given grammar. That is, we will assign to a grammar G a formula equation Φ_G such that Φ_G has a quantifier-free solution iff G is the grammar of a proof. We will discuss these induced formula equations in more detail in Chapter 5, as well as practically effective algorithms to solve them.

Definition 2.9.1. A *formula equation* $\exists X_1 \dots \exists X_n \, \Psi$ is a formula where Ψ is a first-order formula and $X_1, \dots, X_n$ are second-order predicate variables. A *solution* modulo the theory T is a substitution σ of the predicate variables $X_1, \dots, X_n$ to first-order formulas such that T entails $\Psi\sigma$.

A solution is called quantifier-free if $X_1\sigma, \dots, X_n\sigma$ are all quantifier-free.

Example 2.9.1. The FE $\Psi = \exists X \ ((P(a) \to X(a)) \wedge (X(b) \to P(b)))$ has the solution $[X \backslash \lambda x \, P(x)]$ modulo any theory T. The predicate variable X could stand for the quantifier-free matrix of a cut formula, solving the formula equations then amounts to finding a suitable matrix for the cut formula.

On the level of grammars, the reversal of cut- and induction-elimination is feasible in practice. For simple proofs, this process of *cut-introduction* was introduced in [51]. Starting from a Herbrand sequent of a cut-free proof π^* encoded as a term language L, we first find a VTRATG G such that $L(G) \supseteq L$. As the second step we need to find a simple proof π such that $G(\pi) = G$. This is equivalent to solving the induced formula equation Φ_G. (Remember that

solution of the formula equation consists of the quantifier-free matrices of the cut formulas in the simple proof). Given a VTRATG G whose language is tautological (which it is because $G \supseteq L = L(\pi^*)$), we can always solve the formula equation Φ_G, as we will see in Theorem 5.2.1. There are a number of techniques to post-process these formulas to obtain short and human-readable cut-formulas [48, 36]. In practice this algorithm can produce interesting cut formulas, such as automatically synthesizing a lemma stating the transitivity of the order in a lattice, given a proof from only axioms for the meet operation [36].

We can also use reverse the process of induction-elimination [31]. Here we start with a family $(L_i)_{i \in I}$ of Herbrand sequents, one for the instance problem of each parameter. First, we find a induction grammar G such that $\forall i \, L(G, i) \supseteq L_i$. Here we need to solve a more complicated formula equation in the second step. The solution to this formula equation is the quantifier-free matrix of the induction formula. This problem is much harder than the corresponding problem in cut-introduction, and for some grammars there may not even exist such a matrix (even though the language of the grammar is tautological for each parameter, see Theorem 5.4.1 [31]).

We do not have to start with proofs obtained by induction-elimination either, the languages L_i could also be produced by an automated theorem prover. This combination produces an automated inductive theorem prover.

2.10 Regularity and reconstructability

On a theoretical level, we can also view the reversal of induction-elimination of the preceding Section 2.9 as a way to recover the induction grammar of a given simple induction proof from the Herbrand sequents of its induced instance proofs, and finding a solution to the formula equation amounts to recovering the induction formula. In this section we will study this problem on a theoretical level, namely in how far it is decidable whether a family of languages comes from induction-elimination, and whether we can the induction-reversal on the level of grammars is computable. It turns out that this reversal of induction-elimination is computable (in the limit). The interesting

question that we will hence address in the rest of this thesis is how to do this in a feasible way in practice. We assume that $T \models \varphi$ is decidable for quantifier-free formulas φ.

We will consider two kinds of regularity of families of languages in this section: one is whether the family can come from an induction grammar (grammatical regularity), and the other is whether the family can come from a simple induction proof. Let us first study the problem whether a family of tautological languages can actually come from a simple induction proof, i.e., if it is covered by the language of an induction grammar.

Definition 2.10.1. Let $(L_t)_{t\in I}$ be a family of T-tautological languages. Then $(L_t)_{t\in I}$ is *regular* if and only if $\exists \pi\ \forall t\ L_t \subseteq L(\pi_t^*)$, and *grammatically regular* if and only if $\exists G\ \forall t\ L_t \subseteq L(G, t)$.

Every regular family $(L_t)_{t\in I}$ is also grammatically regular by Theorem 2.8.1. Grammatical regularity is a statement purely on the level of formal languages, it does not imply the existence of an induction formula. However, if there is any simple induction proof for the problem at all, then grammatical regularity and regularity are equivalent:

Theorem 2.10.1. *Let $(L_t)_{t\in I}$ be a grammatically regular family. Then $(L_t)_{t\in I}$ is regular if and only if there exists a simple induction proof for the problem.*

Proof. If the family is regular then there exists a simple induction proof by definition. For the other direction, assume that G is an induction grammar covering the family and π is a simple induction proof with induction invariant φ. Consider the induction grammar $G' = G \cup G(\pi)$ (taking the union of the productions). Since φ is a solution for Φ_G, it is easy to see that it is also a solution for $\Phi_{G'}$. Hence we can construct a simple induction proof π' with $G(\pi') = G'$ by Theorem 5.3.1, which witnesses the regularity of $(L_t)_{t\in I}$. □

For finite families, grammatical regularity is decidable but regularity is not (which we will see in Theorem 2.10.4):

Theorem 2.10.2. *The set of grammatically regular families $(L_t)_{t\in I}$ with finite I is decidable.*

Proof. First we show that grammatical regularity is decidable. Note that whenever there is a induction grammar G such that $L(G,t) \supseteq L_t$ for all $t \in I$, there is also one that contains only generalizations of subterms in $(L_t)_{t\in I}$—hence there is a straightforward upper bound on the symbolic complexity of G (i.e., the number of bits required to represent G) and we can simply iterate through all induction grammars of smaller symbolic complexity to find it. If we can find a covering induction grammar, then the family is grammatically regular, otherwise not. □

To show that regularity is not decidable, we first note the unsurprising result that proof by simple induction proof is undecidable:

Theorem 2.10.3 ([31])**.** *The set of sequents that have a simple induction proof is computably enumerable (but not decidable).*

Theorem 2.10.4. *The set of regular families $(L_t)_{t\in I}$ with finite I is computably enumerable (but not decidable).*

Proof. To show computable enumerability, we just enumerate all possible simple induction proofs and check whether $L(\pi_t^*) \supseteq L_t$ for all $t \in I$, all of which are computable operations. For undecidability, set $I = \emptyset$ and use Theorem 2.10.3. □

Let us now turn to algorithms on infinite families $(L_t)_{t\in C}$. Formally, we assume that the family is given as an oracle that returns L_t when given $t \in C$.

Theorem 2.10.5. *For families $(L_t)_{t\in C}$, grammatical regularity and regularity are both undecidable.*

Proof. Fix a regular family $(L_t)_{t\in C}$. Any algorithm that decides (grammatical) regularity returns a result in finite time and hence only accesses a finite subfamily $(L_t)_{t\in I}$ for $I \subseteq C$ finite. Hence it necessarily returns the same result for any other family that extends $(L_t)_{t\in I}$.

An easy example for a family that is not regular is one that grows too fast—by straightforward counting we see that $|L(G,t)| \leq |P|^{|t|}$ where $|P|$ is the number of productions in G, thus any family that grows faster cannot be (grammatically) regular. Hence we can extend $(L_t)_{t\in I}$ to a family $(L'_t)_{t\in C}$

which is not grammatically regular, but which the algorithm claims to be (grammatically) regular. □

While regularity is undecidable, we can approximate the decision problem in a computable way. Namely it is computable in the limit. We can give an algorithm whose output will converge to a covering grammar or simple induction proof as we give it more languages from the infinite family.

Theorem 2.10.6 (learnability in the limit)**.** *There is an algorithm that takes a family $(L_t)_{t\in C}$ as input, and produces a sequence of induction grammars $(G_n)_{n\geq 0}$ with the following property: if $(L_t)_{t\in C}$ is grammatically regular, then there exists an $N \geq 0$ such that $G_n = G_N$ for all $n \geq N$, and $L(G_N, t) \supseteq L_t$ for all t.*

(The sequence $(G_n)_{n\geq 0}$ is eventually constant if and only if $(L_t)_{t\in C}$ is grammatically regular.)

Proof. Enumerate the free constructor terms as $C = \{t_1, t_2, \dots\}$. Now for $i \geq 0$ compute an induction grammar H_i such that $L(H_i, t_j) \supseteq L_{t_j}$ for $j \leq i$ and H_i is of minimal symbolic complexity. As in Theorem 2.10.4, we can iterate through all induction grammars of a certain size to find H_i. The output of the algorithm is then $G_i = \bigcup_{j\leq i} H_i$ where the union operation on grammars is defined as the union of the sets of productions.

If $(L_t)_{t\in C}$ is grammatically regular, i.e., there exists a covering induction grammar G, then the symbolic complexity of H_i is bounded by the symbolic complexity of G for all i. Hence there are only finitely many possibilities for H_i, and also for G_i. Since $(G_i)_i$ is monotonically increasing, it must be eventually constant with value G_N. We have $L(G_N, t_j) = L(G_{N+j}, t_j) \supseteq L(H_{N+j}, t_j) \supseteq L_{t_j}$ by construction. □

Unfortunately the construction in Theorem 2.10.6 is only guaranteed to result in a covering induction grammar. It is possible that this induction grammar is not the grammar of any simple induction proof (because there is no possible induction formula).

Theorem 2.10.7 ([31])**.** *There exists a regular family $(L_t)_{t\in C}$ and an induction grammar G such that $L(G, t) \supseteq L_t$ for all t, but there is no simple induction proof π with $G = G(\pi)$.*

Therefore we have to take a different approach to handle regularity, namely we actually have to enumerate the simple induction proofs:

Theorem 2.10.8 (reconstructability in the limit)**.** *There is an algorithm that takes a family $(L_t)_{t\in C}$ as input, and produces a sequence $(\pi_n)_{n\geq 0}$, where each π_n is either a simple induction proof or $\emptyset$. The output has the following property: if $(L_t)_{t\in C}$ is regular, then there exists an $N \geq 0$ such that $\pi_n = \pi_N \neq \emptyset$ for all $n \geq N$, and $L((\pi_N)_t^*) \supseteq L_t$ for all t.*

(The sequence $(\pi_n)_{n\geq 0}$ is eventually constant if and only if $(L_t)_{t\in C}$ is regular.)

Proof. Enumerate all simple induction proofs as $\psi_0, \psi_1, \ldots$. Define $\tilde{\pi}_n$ as the first ψ_j which proves the current simple induction problem with $j \leq n$, or $\emptyset$ if none exists. By Theorem 2.10.7 it is possible that $L((\tilde{\pi}_n)_t^*) \not\supseteq L_t$, we will fix this as in Theorem 2.10.1. Let G_n be as in Theorem 2.10.6; define π_n as a simple induction proof with $G(\pi_n) = G_n \cup G(\tilde{\pi}_n)$ if $\tilde{\pi}(n) \neq \emptyset$, and $\emptyset$ otherwise. If $(L_t)_{t\in C}$ is regular, then $(\tilde{\pi}_n)_{n\geq 0}$ and $(G_n)_{n\geq 0}$ are the desired eventually constant sequences. □

The simple induction proof π obtained by reconstruction is not unique. It can be different from the original proof in two important aspects: first, the induction grammar $G(\pi)$ can be different—e.g., it can contain extra productions. Second, the induction formula may be different—e.g., it could be any propositionally equivalent formula.

3 Decision problems on grammars

We have seen in Lemma 2.7.5 and Theorem 2.8.1 that cut- and induction-elimination correspond to the language generation process of grammars. When reversing cut- and induction-elimination on the level of grammars, the first step is hence to reverse the language generation. We will investigate the algorithmic side of this problem in Chapter 4, that is, how to effectively compute a grammar that covers a given set of terms.

Finding such covering grammars raises interesting questions: if we have a grammar, how difficult is it to decide whether it actually covers the set of terms? Maybe we want to change the grammar a little, how difficult is it to figure out whether it still generates the same language as the old grammar, or to decide whether it generates a superset? In this chapter we will therefore first investigate the computational complexity of these and related problems on several classes of grammars.

Table 3.1 shows an overview of the obtained complexity results. Many of the problems turn out to be surprisingly hard, e.g. even the problem of deciding whether a VTRATG generates a given term is NP-complete. The main open problem is the complexity of minimal cover, deciding whether there exists a covering grammar whose size is less than a given bound. Except for VTRATGs (where the problem is trivial due to the used size measure), we could only show that the problem is in NP. It remains open whether any of these problems are NP-hard as well (Open Problems 3.7.1, 3.11.1, 3.12.1 and 3.12.2).

The complexity results for TRATGs were previously published as [30]. For other models these complexity questions have been studied as well by other authors: e.g. regular tree automata in [23, Section 1.7], regular expressions, finite automata, and context-free grammars in [90, 59].

	VTRATG	TRATG	tw $\leq k$	Ind	Ind ($\lvert\gamma\rvert \leq k$)
Membership	NP-complete (Theorem 3.2.1)	NP-complete (Theorem 3.2.1)	P (Theorem 3.11.2)	NP-complete (Theorem 3.12.2)	P (Theorem 3.12.1)
Emptiness	coNP-complete (Theorem 3.3.1)	P (Theorem 3.3.2)	P (Theorem 3.11.3)	PSPACE-complete (Theorem 3.12.4)	P (Theorem 3.12.5)
Containment	Π_2^P-complete (Theorem 3.4.1)	Π_2^P-complete (Theorem 3.4.1)	coNP-complete (Theorem 3.11.4)	undecidable (Theorem 3.12.6)	undecidable (Theorem 3.12.6)
Disjointness	coNP-complete (Theorem 3.5.1)	coNP-complete (Theorem 3.5.1)	coNP-complete (Theorem 3.11.5)	undecidable (Theorem 3.12.7)	undecidable (Theorem 3.12.7)
Equivalence	Π_2^P-complete (Theorem 3.6.1)	Π_2^P-complete (Theorem 3.6.1)	coNP-complete (Theorem 3.11.6)	undecidable (Theorem 3.12.8)	undecidable (Theorem 3.12.8)
Minimization	NP-complete (Theorem 3.8.1)	NP-complete (Theorem 3.8.1)	NP-complete (Theorem 3.11.7)	NP-complete (Theorem 3.12.9)	NP-complete (Theorem 3.12.10)
Cover	P (Corollary 3.7.1)	NP-complete? (Open Problem 3.7.1)	NP-complete? (Open Problem 3.11.1)	NP-complete? (Open Problem 3.12.1)	NP-complete? (Open Problem 3.12.2)
n-Cover	P (Lemma 3.7.1)	NP-complete (Theorem 3.7.2)	NP-complete (Theorem 3.11.9)	n/a	n/a

Table 3.1: Complexity of decision problems on VTRATGs, TRATGs, VTRATGs with dependency graph of bounded treewidth, induction grammars, and induction grammars where $\overline{\gamma}$ has a bounded length.

3.1 Computational complexity and the polynomial hierarchy

Let us first review some basic and well-known notions of complexity theory; the reader is referred to textbooks such as [77] for more details. Formally, a computational (decision) problem is defined by the set of accepting input values, encoded as binary words:

Definition 3.1.1. A *computational problem* is a set of words $L \subseteq \{0,1\}^*$.

We often call a set of words a language. There are famously many equivalent ways to define computability, so let us use Turing machines to define when a function (between binary words) computable.

Definition 3.1.2. A function $f : \{0,1\}^* \to \{0,1\}^*$ is called *computable* if there is a Turing machine such that for any given input $a \in \{0,1\}^*$ it produces the output $f(a)$. The function f is called *polynomial-time computable* if there additionally exists a polynomial p such that the number of steps of the Turing machine on input a is bounded by $p(|a|)$ where $|a|$ is the length of a. The class of polynomial-time computable languages is called P.

We can then extend the notion of computability to sets, by saying that a set is computable if its characteristic function is computable:

Definition 3.1.3. A set $L \subseteq \{0,1\}^*$ is computable if its characteristic function $\chi_L : \{0,1\}^* \to \{0,1\}$ is computable, where $\chi_L(a) = 1$ iff $a \in L$. The set L is polynomial-time computable if χ_L is polynomial-time computable.

It seems plausible that some languages are easier to compute than others. One way to make this intuition concrete is by studying certain kinds of reductions from one language to another. If L_1 can be reduced to L_2, then L_1 is in some sense simpler than L_2. There are various kinds of reductions that we can study, typically a reduction consists of a program that can use L_2 in a restricted way to compute L_1. Notions of reduction can then vary along several axes: first, in how they can use the other language. The program could use L_2 like a constant-time function call (a so-called oracle), this gives us Turing reductions.

Or it could translate an input $a \in L_1$ to an input $f(a) \in L_2$, these are called many-to-one reductions. On a second axis, they can also differ by resource limitations imposed on the program. For example, we can require the program to terminate in polynomial time.

Definition 3.1.4. Let $A, B \subseteq \{0,1\}^*$. A *polynomial-time many-to-one reduction* from A to B is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $A = f^{-1}(B)$. We then say the A is polynomial-time many-to-one reducible to B, and write $A \leq_P B$.

The condition $A = f^{-1}(B)$ means that we can use f to translate questions about the set A to questions about B, because we have $x \in A$ iff $f(x) \in B$ for all x. The polynomial-time hierarchy was first introduced to show that universality of regular expressions is hard for every level of the polynomial-time hierarchy [70]. The lowest level of the hierarchy are the polynomial-time computable sets, the further levels are built by iterating the $\mathsf{NP}(\cdot)$ and $\mathsf{coNP}(\cdot)$ operations:

Definition 3.1.5. Let $C \subseteq \mathcal{P}(\{0,1\}^*)$ be a class of languages. The class $\mathsf{NP}(C)$ is the class of languages accepted by non-deterministic polynomial-time Turing machines with oracles from C. A non-deterministic Turing machine with accepts a word $w \in \{0,1\}^*$ if there exists an execution with input w where the machine outputs 1. Such a Turing machine has polynomial-time runtime if there exists a fixed polynomial which bounds the number of steps taken for every execution and for every input.

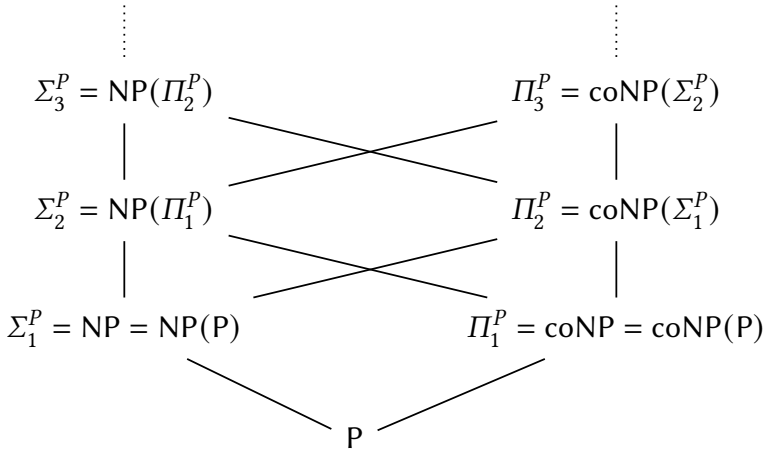
The class $\mathsf{coNP}(C)$ consists of the sets for which the non-deterministic Turing machine outputs 1 on *every* execution. Equivalently, we define it as the complements of $\mathsf{NP}(C)$:

Definition 3.1.6. The class $\mathsf{coNP}(C) = \{\{0,1\}^* \setminus L \mid L \in \mathsf{NP}(C)\}$ consists of the complements of sets in $\mathsf{NP}(C)$.

Definition 3.1.7. The polynomial hierarchy is defined recursively by $\Sigma_0^P = \Pi_0^P = \mathsf{P}$, and $\Sigma_{n+1}^P = \mathsf{NP}(\Pi_n^P)$ as well as $\Pi_{n+1}^P = \mathsf{coNP}(\Sigma_n^P)$. We define $\mathsf{NP} = \Sigma_1^P$ and $\mathsf{coNP} = \Pi_1^P$.

Each of the levels of the polynomial hierarchy is contained in the every level above it:

Lemma 3.1.1. *$\Sigma_i^P \subseteq \Sigma_j^P$, $\Sigma_i^P \subseteq \Pi_j^P$, $\Pi_i^P \subseteq \Sigma_j^P$, and $\Pi_i^P \subseteq \Pi_j^P$ for all $i < j$.*



It is an open problem whether the inclusions in Lemma 3.1.1 are strict. Hence if we want to show that a given $L \subseteq \{0,1\}^*$ is complicated, we cannot hope to prove that e.g. $L \in \mathsf{NP} \setminus \mathsf{P}$. Instead we can only show that it is as hard as any other set in NP, as measured by the $\leq_P$ reduction:

Definition 3.1.8. Let $C \subseteq \mathcal{P}(\{0,1\}^*)$ be a class of languages. Then a set $L \in \{0,1\}^*$ is called C-hard if $A \leq_P L$ for every $A \in C$. If additionally $L \in C$, then L is called C-complete.

The classical NP-complete problem is propositional satisfiability:

Problem 3.1.1 (SAT)**.**
Given a propositional formula F, is F satisfiable?

Theorem 3.1.1. *SAT is* NP*-complete.*

There are analogous Σ_n^P- and Π_n^P-complete problems, which determine the truth of quantified Boolean formulas (QBFs). QBFs allow in addition to the propositional connectives also the quantifiers $\forall, \exists$ over the Booleans. The canonical Π_n^P-complete problem consists of deciding the truth of QBFs in Π_n, that is, closed prenex QBFs whose quantifier prefix has n blocks alternating

between blocks of universal and existential quantifiers, starting with universal quantifiers (and analogously for Σ_n^P and Σ_n-QBFs).

For illustration, we can express the satisfiability of the formula $a \wedge b \vee c$ as the truth of the QBF $\exists a \exists b \exists c\ (a \wedge b \vee c)$. Truth of closed QBFs is easily seen to be decidable: we can recursively replace $\forall x\, \varphi(x)$ by $\varphi(\top) \wedge \varphi(\bot)$, and $\exists x\, \varphi(x)$ by $\varphi(\top) \vee \varphi(\bot)$, resulting in a formula without quantifiers or variables that can be immediately evaluated.

Problem 3.1.2 (Π_n-TQBF)**.**
Given a closed QBF $\psi = \forall \overline{x_1} \exists \overline{x_2} \forall \overline{x_3} \ldots Q \overline{x_n}\ \varphi$ where φ is quantifier-free, is ψ true?

Problem 3.1.3 (Σ_n-TQBF)**.**
Given a closed QBF $\psi = \exists \overline{x_1} \forall \overline{x_2} \exists \overline{x_3} \ldots Q \overline{x_n}\ \varphi$ where φ is quantifier-free, is ψ true?

Lemma 3.1.2. *Π_n-TQBF is Π_n^P-complete and Σ_n-TQBF is Σ_n^P-complete for all n.*

We can hence intuitively see the complexity class Π_n^P as the class of problems that have a description like $\forall x_1 \exists x_2 \ldots Q x_n\ \varphi(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n \in \{0,1\}^*$ and $\varphi(x_1, \ldots, x_n)$ is polynomially-time computable. For instance, the typical problem in Π_2^P has the form $\forall x \exists y\ \varphi(x, y)$; for every x we can non-deterministically pick a "certificate" y and then check it in polynomial time.

3.2 Membership

Let us first determine the complexity of the membership problem for VTRATGs, that is, whether a grammar generates a given term. The term in this problem is represented as a DAG instead of a tree since this is required for the reductions in the following sections. The complexity of the problem would however be unchanged if we represent the term as a tree instead.

Problem 3.2.1 (VTRATG-Membership)**.**
Given a VTRATG G and a term t represented as a DAG, is $t \in L(G)$?

Problem 3.2.2 (TRATG-Membership)**.**
Given a TRATG G and a term t represented as a DAG, is $t \in L(G)$?

Clearly VTRATG-Membership is in NP, since we can guess a derivation (which is a subset of the productions whose size is clearly bounded by the size of G) and check whether $t \in L(G)$ in polynomial time. In Definition 2.5.3, we defined the size of a VTRATG as the number of its production. In order to avoid confusion, we use the term symbolic size to refer to the size of a representation in bits; polynomial-time hence means that the runtime is bounded by a polynomial in the symbolic size of the input. Terms are always represented as DAGs, unless noted otherwise. Given sets A and B, we write $A \leq_P B$ if there exists a polynomial-time many-to-one reduction from A to B. For the NP-hardness of VTRATG-Membership, we will show that 3SAT $\leq_P$ VTRATG-Membership:

Problem 3.2.3 (3SAT)**.**
Given a propositional formula F in 3CNF, is F satisfiable?

For simplicity, we assume that the formula F is represented in the following syntax: the propositional variables occurring in F are exactly $\mathtt{x}_1, \ldots, \mathtt{x}_m$, and there are exactly n clauses, each containing at most 3 literals. Literals are either $\mathtt{x}_j$ or $\mathtt{neg}(\mathtt{x}_j)$ for some $1 \leq j \leq m$. Clauses are written $\mathtt{or}(l_1, l_2, l_3)$ where l_i is `true`, `false`, or a literal for $i \in \{1, 2, 3\}$. The formula F is then $\mathtt{and}(c_1, \ldots, c_n)$ where c_i is a clause for each $1 \leq i \leq n$. We can now define a TRATG $\mathrm{Sat}_{n,m}$ that will encode 3SAT for n clauses and m variables.

Definition 3.2.1. Let $n, m > 0$. We define the TRATG $\mathrm{Sat}_{n,m}$ with the following productions:

$$
\begin{aligned}
\mathrm{A} &\to \mathtt{and}(\mathrm{Clause}_1, \ldots, \mathrm{Clause}_n) \\
\mathrm{Clause}_i &\to \mathtt{or}(\mathrm{True}_i, \mathrm{Any}_{i,1}, \mathrm{Any}_{i,2}) \\
\mathrm{Clause}_i &\to \mathtt{or}(\mathrm{Any}_{i,1}, \mathrm{True}_i, \mathrm{Any}_{i,2}) \\
\mathrm{Clause}_i &\to \mathtt{or}(\mathrm{Any}_{i,1}, \mathrm{Any}_{i,2}, \mathrm{True}_i) \\
\mathrm{Any}_{i,k} &\to \mathtt{x}_1 \mid \mathtt{neg}(\mathtt{x}_1) \mid \cdots \mid \mathtt{x}_m \mid \mathtt{neg}(\mathtt{x}_m) \mid \mathtt{false} \mid \mathtt{true} \\
\mathrm{True}_i &\to \mathrm{Value}_1 \mid \cdots \mid \mathrm{Value}_m \mid \mathtt{true} \\
\mathrm{Value}_j &\to \mathtt{x}_j \mid \mathtt{neg}(\mathtt{x}_j)
\end{aligned}
$$

In fact $\mathrm{Sat}_{n,m}$ generates exactly the satisfiable CNFs. Rigidity provides the main ingredient for this construction: since we can use at most one production

per nonterminal, we can "synchronize" across different clauses and make sure that we assign consistent values to the propositional variables. Which value we assign to the x_j variable is determined by the choice of the production for the Value_j nonterminal. It is clear that $\text{Sat}_{n,m}$ can be computed in polynomial time depending on m and n.

Lemma 3.2.1. *Let $n, m > 0$, and F be a propositional formula in 3CNF with exactly n clauses and at most m variables. Then $F \in L(\text{Sat}_{n,m})$ iff F is satisfiable.*

Proof. Let δ be a derivation of F in $\text{Sat}_{n,m}$. We define an interpretation I_δ such that $I_\delta(\mathsf{x}_j) = 1$ iff $\text{Value}_j \to \mathsf{x}_j \in \delta$, for all $j \leq m$. Now we need to show that indeed $I_\delta \models F$. By case analysis on the chosen productions, we have $\text{Value}_\text{j} \in \text{dom}(\sigma_\delta)$ and $I_\delta \models \text{Value}_j\sigma_\delta$ for all $1 \leq j \leq m$, then $I_\delta \models \text{True}_i\sigma_\delta$ and $I_\delta \models \text{Clause}_i\sigma_\delta$ for all $1 \leq i \leq n$, and hence $I_\delta \models F$ since $F = \text{A}\sigma_\delta$.

On the other hand, let $F = \mathsf{and}(c_1, \ldots, c_n)$ and $I \models F$ be a satisfying interpretation. We construct a derivation δ with $\text{A} \to \mathsf{and}(\text{Clause}_1, \ldots, \text{Clause}_n) \in \delta$, $\text{Value}_j \to \mathsf{x}_j \in \delta$ if $I \models \mathsf{x}_j$, and $\text{Value}_j \to \mathsf{neg}(\mathsf{x}_j) \in \delta$ if $I \not\models \mathsf{x}_j$.

For $1 \leq i \leq n$, we define the productions for Clause_i, $\text{Any}_{i,1}$, $\text{Any}_{i,2}$, and True_i depending on which literal is true in the i-th clause. Let $\mathsf{or}(l_1, l_2, l_3)$ be the i-th clause, and let j be such that $l_1 = \mathsf{x}_j$ or $l_1 = \mathsf{neg}(\mathsf{x}_j)$. If $I \models l_1$, we let $\text{Clause}_i) \to \mathsf{or}(\text{True}_i, \text{Any}_{i,1}, \text{Any}_{i,2}) \in \delta$, $\text{True}_i \to \text{Value}_j \in \delta$, $\text{Any}_{i,1} \to l_2 \in \delta$ and $\text{Any}_{i,2} \to l_3 \in \delta$. The cases where the second or third literal are true are handled analogously. We can see that this set δ is then indeed a derivation of F in $\text{Sat}_{n,m}$. □

Theorem 3.2.1. *The TRATG-Membership and VTRATG-Membership problems are* NP*-complete.*

Proof. By reduction from 3SAT using Lemma 3.2.1. □

3.3 Emptiness

In Theorem 3.2.1 we have seen that the membership problem for VTRATGs is NP-complete. An even more basic property about VTRATGs is emptiness, that is: does the grammar not generate any terms at all?

Problem 3.3.1 (VTRATG-Emptiness)**.**
Given a VTRATG G, is $L(G) = \emptyset$?

This property is not trivial: if there are no productions for some nonterminal, then the language is not necessarily empty. And it is not sufficient to remove the production-less nonterminal, as it might occur in an unused part of a vector:

Example 3.3.1. In the following VTRATG, there is no production for the nonterminal vector C. However the language $L(G) = \{d\}$ is non-empty since the production $A \to B_1$ only contains B_1 and not B_2.

$$A \to B_1$$
$$(B_1, B_2) \to (d, C)$$

Clearly VTRATG-Emptiness is in coNP, since we can guess a subset of the productions (whose size is clearly bounded by the size of G) and check whether it is a derivation in polynomial time. For the coNP-hardness of VTRATG-Emptiness, we will show a reduction from the complement of 3SAT.

Theorem 3.3.1. *VTRATG-Emptiness is* coNP*-complete.*

Proof. By reduction from the complement of 3SAT. Let $\varphi = \mathsf{and}(c_1, \ldots, c_n)$ with $c_i = \mathrm{or}(l_{i,1}, \ldots, l_{i,3})$ and $l_{i,k} \in \{\mathsf{x}_{j_{i,k}}, \mathsf{neg}(\mathsf{x}_{j_{i,k}})\}$. We want decide whether φ is unsatisfiable. Define $p_{i,j} = 1$ if $l_{i,k} = \mathsf{x}_{j_{i,k}}$ and $p_{i,j} = 2$ if $l_{i,k} = \mathsf{neg}(\mathsf{x}_{j_{i,k}})$; this number $p_{i,j}$ indicates whether the literal is negated and will be used as an index in $\mathrm{Value}_{j,p_{i,k}}$. We define the VTRATG G with the following productions:

$$\mathrm{A} \to \mathsf{and}(\mathrm{Clause}_1, \ldots, \mathrm{Clause}_n)$$
$$\mathrm{Clause}_i \to \mathtt{or}(\mathrm{Value}_{j_{i,1},q_1}, \mathrm{Value}_{j_{i,2},q_2}, \mathrm{Value}_{j_{i,3},q_3})$$
$$(\text{for each } i \le n \text{ and } q_1, q_2, q_3 \in \{1, 2\}$$
$$\text{such that } q_1 = p_{i,1} \vee q_2 = p_{i,2} \vee q_3 = p_{i,3})$$
$$(\mathrm{Value}_{j,1}, \mathrm{Value}_{j,2}) \to (\mathrm{True}, \mathrm{False}) \mid (\mathrm{False}, \mathrm{True}) \quad (\text{for each } j \le m)$$
$$\mathrm{True} \to \mathtt{true}$$

It remains to show that $L(G) \neq \emptyset$ iff there is a satisfying assignment for φ. If δ is a derivation in G, then for every $j \le m$, either $\mathrm{Value}_{j,1}$ or $\mathrm{Value}_{j,2}$ (but not

both) can occur in δ, depending on which production was chosen for $\overline{\text{Value}_j}$. If the first production was chosen, then $\text{Value}_{j,2}$ can not occur in δ, since False has no productions (and vice versa for the other production).

Define an assignment I such that $I \models \mathsf{x}_j$ if $\text{Value}_{j,1}$ occurs in δ, and $I \not\models \mathsf{x}_j$ if $\text{Value}_{j,2}$ occurs in δ. We have chosen the $p_{i,j}$ in such a way that $I \models l_{i,j}$ if $\text{Value}_{j,p_{i,j}}$ occurs in δ. We will now show that $I \models \varphi$: δ contains the production $\text{Clause}_i \to \mathsf{or}(\text{Value}_{j_{i,1},q_1}, \text{Value}_{j_{i,2},q_2}, \text{Value}_{j_{i,3},q_3})$ for some $q_1, q_2, q_3 \in \{1, 2\}$. If $q_j = p_{i,j}$, then $\text{Value}_{i,p_{i,j}}$ occurs in δ and $I \models l_{i,j}$ and hence $I \models c_i$. Since this holds for every clause c_i, we have $I \models \varphi$.

If on the other hand I is a satisfying assignment, then we get a derivation δ by using the production $\overline{\text{Value}_j} \to (\text{True}, \text{False})$ if $I \models \mathsf{x}_j$ and $\overline{\text{Value}_j} \to (\text{True}, \text{False})$ if $I \not\models \mathsf{x}_j$. For each clause, we pick the production for Clause_i depending on which literal evaluates to true. □

For TRATGs, we can decide emptiness much easier—in polynomial time:

Problem 3.3.2 (TRATG-Emptiness)**.**
Given a TRATG G, is $L(G) = \emptyset$?

Theorem 3.3.2. *TRATG-Emptiness* $\in \mathsf{P}$.

Proof. Let $G = (N, \Sigma, P, A)$. Define the set of "productive" nonterminals $\tilde{N}$ by $B \in \tilde{N}$ iff there exists a production $B \to t \in P$ such that $C \in \tilde{N}$ for every nonterminal C that occurs in t. This set $\tilde{N}$ is computable in polynomial time. We only need to show $L(G) \neq \emptyset$ iff $A \in \tilde{N}$.

Assume $A \in \tilde{N}$. Then we have for every nonterminal in $\tilde{N}$ a production satisfying the conditions above. This set of productions is a derivation of some term in $L(G)$: it contains at most one production per nonterminal, and the derived term contains no nonterminal: whenever a nonterminal C appears in the derivation from a production $B \to t$ (i.e., where C occurs in t), then C will be replaced later on (since $C \in \tilde{N}$).

If on the other hand δ is a derivation of some term in $L(G)$, then $\tilde{N}$ contains all nonterminals occurring in δ, including A. (This is the difference to VTRATGs: if a nonterminal occurs in the derivation of a TRATG, then the right-hand side of the production and all of its nonterminals occur as well. In

a derivation of a VTRATG, it is possible that only a part of the right-hand side appears in the derivation.) □

3.4 Containment

We will now consider the problem of determining whether the generated languages of two (V)TRATGs are contained in one another:

Problem 3.4.1 (VTRATG-Containment)**.**
Given VTRATGs G_1 and G_2, is $L(G_1) \subseteq L(G_2)$?

Problem 3.4.2 (TRATG-Containment)**.**
Given TRATGs G_1 and G_2, is $L(G_1) \subseteq L(G_2)$?

We will show that VTRATG-Containment is complete for $\Pi_2^P = \mathsf{coNP}(\mathsf{NP})$ by showing Π_2-TQBF $=_P$ Π_2-3TQBF $\leq_P$ VTRATG-Containment. We use a syntactic variant of Π_n-TQBF where the matrix is restricted to formulas in 3CNF, to apply our results from the earlier Section 3.2.

Problem 3.4.3 (Π_2-3TQBF)**.**
Given a closed QBF $\forall \mathsf{y}_1 \cdots \forall \mathsf{y}_k \exists \mathsf{x}_1 \cdots \exists \mathsf{x}_m\ F$ where F is in 3CNF, is F true?

A Π_2-QBF is true if and only if the propositional matrix F is satisfiable for all instances of the universal quantifiers. Theorem 3.2.1 showed that we can encode satisfiability as TRATG-Membership. It only remains to encode the instantiation of the universal quantifiers as a TRATG. We will thus define a TRATG that generates exactly the instances of the QBF where we substitute the universal variables by either `true` or `false`:

Definition 3.4.1. Let $F = \mathsf{and}(c_1, \ldots, c_n)$ be a propositional formula in 3CNF. We define the TRATG $\mathrm{Inst}_{F,\bar{\mathsf{y}}}$ with the following productions:

$$\mathrm{A} \longrightarrow F[\mathsf{y}_1 \backslash \mathrm{Y}_1, \ldots, \mathsf{y}_k \backslash \mathrm{Y}_k]$$
$$\mathrm{Y}_j \longrightarrow \texttt{true} \mid \texttt{false} \quad \text{for } j \leq k$$

Lemma 3.4.1. *Let $Q = \forall \mathsf{y}_1 \cdots \forall \mathsf{y}_k \exists \mathsf{x}_1 \cdots \exists \mathsf{x}_m\ F$ be a closed QBF such that $F = \mathsf{and}(c_1, \ldots, c_n)$ is in 3CNF. Then Q is true iff $L(\mathrm{Inst}_{F,\bar{\mathsf{y}}}) \subseteq L(\mathrm{Sat}_{n,k+m})$.*

Proof. The QBF Q is true if and only if every instance $F[\mathtt{y}_1 \backslash v_1, \ldots, \mathtt{y}_k \backslash v_k]$ is satisfiable where $v_1, \ldots, v_k \in \{\mathtt{true}, \mathtt{false}\}$. The TRATG $\mathrm{Inst}_{F,\overline{\mathtt{y}}}$ generates exactly these instances. Hence Q is true iff $L(\mathrm{Inst}_{F,\overline{\mathtt{y}}}) \subseteq L(\mathrm{Sat}_{n,m})$ by Lemma 3.2.1. □

Theorem 3.4.1. *TRATG-Containment and VTRATG-Containment are Π_2^P-complete.*

Proof. VTRATG-Containment is in Π_2^P because we can guess a derivation δ of a term t in G_1, and then check whether $t \in L(G_2)$. Each derivation can be produced in polynomial time, and checking whether $t \in L(G_2)$ is in NP per Theorem 3.2.1. For hardness, we reduce the Π_2^P-complete Π_2-TQBF to TRATG-Containment via Lemma 3.4.1. □

3.5 Disjointness

The complexity of the disjointness problem follows straightforwardly from (the complement of) VTRATG-Membership.

Problem 3.5.1 (VTRATG-Disjointness)**.**
Given VTRATGs G_1 and G_2, is $L(G_1) \cap L(G_2) = \emptyset$?

Problem 3.5.2 (TRATG-Disjointness)**.**
Given TRATGs G_1 and G_2, is $L(G_1) \cap L(G_2) = \emptyset$?

Theorem 3.5.1. *TRATG-Disjointness and VTRATG-Disjointness are* coNP*-complete.*

Proof. We first show that VTRATG-Disjointness is in coNP. For all derivations δ_1 of a term t_1 in G_1 and δ_2 of a term t_2 in G_2, we check that $t_1 \neq t_2$: as usual we can generate a derivation in polynomial time, and checking equality of terms is polynomial as well. Hardness follows via a reduction from the complement of TRATG-Membership: $t \notin L(G)$ if and only if $L(G) \cap L(G_t) = \emptyset$, where G_t is a TRATG such that $L(G_t) = \{t\}$ (such as $G_t = (\{A\}, \Sigma, P, A)$ with $P = \{\mathrm{A} \to t\}$). □

3.6 Equivalence

The complexity of equivalence follows from VTRATG-Containment via a union operation on VTRATGs.

Problem 3.6.1 (VTRATG-Equivalence)**.**
Given VTRATGs G_1 and G_2, is $L(G_1) = L(G_2)$?

Problem 3.6.2 (TRATG-Equivalence)**.**
Given TRATGs G_1 and G_2, is $L(G_1) = L(G_2)$?

Definition 3.6.1. Let $G_1 = (N_1, \Sigma_1, P_1, \mathrm{A}_1), G_2 = (N_2, \Sigma_2, P_2, \mathrm{A}_2)$ be VTRATGs such that $N_1 \cap N_2 = \emptyset$. Then $G_1 \cup G_2 = (N_1 \cup N_2, \Sigma_1 \cup \Sigma_2, P', \mathrm{A}_1)$, where $P' = P_1 \cup P_2 \cup \{\mathrm{A}_1 \to \mathrm{A}_2\}$.

It is easy to see that $L(G_1 \cup G_2) = L(G_1) \cup L(G_2)$. We will also use the notation $G_1 \cup G_2$ for VTRATGs that share nonterminals: we then implicitly rename the nonterminals in one TRATG so that they are disjoint.

Lemma 3.6.1. *Let G_1 and G_2 be VTRATGs. Then $L(G_1 \cup G_2) = L(G_1) \cup L(G_2)$.*

Proof. Let $G_1 = (N_1, \Sigma_1, P_1, \mathrm{A}_1)$ and $G_2 = (N_2, \Sigma_2, P_2, \mathrm{A}_2)$, and assume without loss of generality that $N_1 \cap N_2 = \emptyset$. Let δ be a derivation of t in $G_1 \cup G_2$. If $\mathrm{A}_1 \to \mathrm{A}_2 \in \delta$, then $\delta \cap P_2$ is a derivation of t in G_2. Otherwise, $\mathrm{A}_1 \to \mathrm{A}_2 \notin \delta$ and $\delta \cap P_1$ is a derivation of t in G_1. Hence $L(G_1 \cup G_2) \subseteq L(G_1) \cup L(G_2)$.

For the reverse inclusion, let first δ be a derivation of t in G_1. Then δ is a derivation of t in $G_1 \cup G_2$. Now let δ be a derivation of t in G_2, then $\delta \cup \{\mathrm{A}_1 \to \mathrm{A}_2\}$ is a derivation of t in $G_1 \cup G_2$. Hence $L(G_1) \cup L(G_2) \subseteq L(G_1 \cup G_2)$. □

Theorem 3.6.1. *VTRATG-Equivalence and TRATG-Equivalence are Π_2^P-complete.*

Proof. We first show that it is in Π_2^P via a reduction to VTRATG-Containment: $L(G_1) = L(G_2)$ if and only if $L(G_1) \subseteq L(G_2)$ as well as $L(G_2) \subseteq L(G_1)$—and Π_2^P is closed under intersection. Hardness follows by a reduction from TRATG-Containment: $L(G_1) \subseteq L(G_2)$ iff $L(G_1) \cup L(G_2) = L(G_2)$. This is equivalent to $L(G_1 \cup G_2) = L(G_2)$, an instance of TRATG-Equivalence. □

3.7 Minimal cover

3.7.1 Minimal cover for terms

We saw in Chapter 2 that cut-elimination of proofs with Π_1-cuts corresponds to the computation of the language of VTRATGs. To reverse cut-elimination we need to find a grammar that covers a given language. There are trivial solutions to find such a VTRATG. For example, given a finite set of terms L we might return a VTRATG with one nonterminal A and productions $A \longrightarrow t$ for every $t \in L$. However such trivial grammars correspond to trivial cuts. If we want to introduce interesting structure into proofs, we need to impose extra conditions on the grammar. A good condition is that the grammar should be small. Small grammars correspond to small proofs (see Lemmas 2.7.4 and 2.8.2), and it is a reasonable hypothesis that small proofs typically contain interesting structure. In the case of induction there is an even more significant size difference: we want to find a finite grammar that generates an infinite and unbounded family of sets of terms. Thus we are looking for grammars of minimal size, that is, with the least number of productions.

The finite set L corresponds to a tautological set of instances. Since supersets of tautological disjunctions are tautological as well, we require that the TRATG covers the given finite set of terms, as opposed to $L(G) = L$. The covering condition $L(G) \supseteq L$ is similar to (but different from) the one imposed on cover automata [17, 18]: there an automaton A is sought such that $L(A) \supseteq L$, but in addition it is required that $L(A) \setminus L$ consists only of words longer than any word in L. Another related notion is that of the grammatical complexity of a finite language as defined in [15] and studied further in [13, 3, 14, 98]: the grammatical complexity of a finite language L is defined as the minimal number of productions of a grammar G with $L(G) = L$. In this way, each class of grammars gives rise to a measure of descriptional complexity.

As a first attempt, we might formalize the problem as a decision problem on VTRATGs:

Problem 3.7.1 (VTRATG-Cover)**.**
Given a finite set of terms L and $k \geq 0$, is there a VTRATG G such that $|G| \leq k$ and $L(G) \supseteq L$?

However this formulation of the problem turns out to be easily solvable, since VTRATGs always allow optimal compression:

Theorem 3.7.1 ([32]). *Let L be a finite set of terms, and $l_0, \ldots, l_n$ be natural numbers such that $|L| \leq \prod_i l_i$. Then there exists a VTRATG G of size $|G| = \sum_i l_i$ such that $L(G) = L$.*

For example, Theorem 3.7.1 shows that for every set of terms L of size $|L| \leq 2^n$ there exists a covering VTRATG of size $2n$. Unfortunately, the VTRATG constructed in Theorem 3.7.1, while small, does not correspond to a proof with interesting lemmas.

Corollary 3.7.1. *VTRATG-Cover* $\in$ P

Proof. Assume that $G = (N, \Sigma, P, A)$ is a covering VTRATG. For $\overline{B} \in N$, define $P_{\overline{B}}$. Then $|L| \leq |L(G)| \leq \prod_{\overline{B} \in N} |P_{\overline{B}}|$, and $|G| = |P| = \sum_{\overline{B} \in N} |P_{\overline{B}}|$. That is, the minimal covering VTRATG is of the form given by Theorem 3.7.1. We only need to find a minimal decomposition, that is, a tuple $(l_0, \ldots, l_n)$ such that $\sum_i l_i \leq |L|$, $\prod_i l_i \geq |L|$, and $\sum_i l_i \leq |L|$ is minimal. Furthermore we only need to consider decompositions such that $\prod_i l_i < 2|L|$: if we have a decomposition $\bar{l}$ with $\prod_i l_i \geq 2|L|$ then we decrement one of l_i until the product is less than $2|L|$.

We can now recursively compute the set of sums and products of decompositions: $A = \{(\sum_i l_i, \prod_i l_i) \mid \sum l_i, \prod_i l_i \leq 2|L|\}$ with $(m, k) \in A$ iff $m = k$ or $(m_1 + m_2, k_1 k_2) \in A$ for some $m_1, m_2 < m$ and $k_1, k_2 < k$. This set can be computed in time bounded by a polynomial in $|L|$. The minimal size of a covering VTRATG is then $\min\{m \mid \exists k \geq |L| \colon (m, k) \in A\}$. □

Hence we consider the decision version of minimal cover for TRATGs instead of VTRATGs:

Problem 3.7.2 (TRATG-Cover).
Given a finite set of terms L and $k \geq 0$, is there a TRATG G such that $|G| \leq k$ and $L(G) \supseteq L$?

TRATG-Cover is in NP, since potential TRATGs G have at most $|L|$ productions, hence at most $|L|$ nonterminals, and the symbolic size of each production is bounded by the symbolic size of L.

Open Problem 3.7.1. Is TRATG-Cover NP-complete?

So far a solution to Open Problem 3.7.1 remains elusive. However we will show that the following version of TRATG-Cover with a fixed parameter is NP-complete:

Problem 3.7.3 (TRATG-n-Cover)**.**
Given a finite set of terms L and $k \geq 0$, is there a TRATG $G = (N, \Sigma, P, \mathrm{A})$ such that $|N| \leq n$, $|P| \leq k$, and $L(G) \supseteq L$?

Theorem 3.7.2. *TRATG-n-Cover is* NP*-complete for $n \geq 2$.*

The proof of NP-hardness in the following Section 3.7.2 only requires regular grammars for words. A similar approach to prove lower bounds for TRATGs using regular grammars for words was already successfully used in descriptional complexity [32].

For grammars on words, the shortest grammar problem—given a string and size, is there a context-free grammar of at most this size generating just the singleton containing the string—is known to be NP-complete [92, 19] and several approximation algorithms are known. The most important difference between the classical setting and ours is that a TRATG does not compress a single string or tree but a finite set of trees. We compress with regard to the number of production rules as opposed to, e.g., the size of the grammar as a binary string. Moreover, we cover the input language instead of reproducing it exactly.

Just as Problem 3.7.1 turned out to be polynomial-time computable due to trivial optimally compressing VTRATG of Theorem 3.7.1, the restricted problem analogous to TRATG-n-Cover for VTRATGs is polynomial-time computable as well:

Problem 3.7.4 (VTRATG-n-Cover)**.**
Given a finite set of terms L and $k \geq 0$, is there a VTRATG $G = (N, \Sigma, P, \mathrm{A})$ such that $|N| \leq n$, $|P| \leq k$, and $L(G) \supseteq L$?

Lemma 3.7.1. *VTRATG-n-Cover* $\in$ P.

Proof. Similar to Corollary 3.7.1. For every m, the set $A_m = \{(\sum_{i\leq m} l_i, \prod_{i\leq m} l_i) \mid \sum_{i\leq m} l_i, \prod_{i\leq m} l_i \leq 2|L|\}$ is polynomial-time computable. Observe that there exists such a covering VTRATG iff there exists a tuple $(l_0, \ldots, l_m)$ with $m \leq n$, $\sum_i l_i \leq k$, and $\prod_i l_i \geq |L|$, this is the case iff $\exists m \leq n\ \exists p \geq |L|\ \exists s \leq k\ (p, s) \in A_m$. □

3.7.2 Minimal cover for words

It is well known that we can represent words as trees by using unary function symbols instead of letters. We may hence represent the word `hello` as the tree $\mathtt{h}(\mathtt{e}(\mathtt{l}(\mathtt{l}(\mathtt{o}(\epsilon)))))$, for example. Under this correspondence, TRATGs with only unary functions and constants correspond exactly to acyclic regular grammars:

Definition 3.7.1. A regular grammar $G = (N, \Sigma, P, \mathrm{A})$ is a tuple consisting of a start symbol $\mathrm{A} \in N$, a finite set of nonterminals N, a finite set of letters Σ such that $N \cap \Sigma = \emptyset$, and a finite set of productions $P \subseteq N \times \Sigma^*(N \cup \{\epsilon\})$. We call G *acyclic* if there exists a strict linear order $\prec$ on the nonterminals such that $\mathrm{B} \prec \mathrm{C}$ whenever $\mathrm{B} \to w\mathrm{C} \in P$ for some $w \in \Sigma^*$.

The one-step derivation relation $\Rightarrow_G$ is defined by $w\mathrm{B} \Rightarrow_G wv$ for $\mathrm{B} \to v \in P$ and $w \in \Sigma^*$. The language $L(G) = \{w \in \Sigma^* \mid \mathrm{A} \Rightarrow_G^* w\}$ then consists of the derivable words. We write $|G| = |P|$ for the number of productions. In the case of acyclic regular grammars, the derivations and the generated language correspond exactly to those for TRATGs. Note that there is no need to require rigidity for derivations here: derivations in acyclic regular grammars can never use more than one production per nonterminal. Rigidity only plays a role for terms, where we can have multiple parallel occurrences of a nonterminal.

Problem 3.7.5 (Regular-Cover)**.**
Given a finite set of words L and $k \geq 0$, is there an acyclic regular grammar G such that $|G| \leq k$ and $L(G) \supseteq L$?

Lemma 3.7.2. *Regular-Cover $\leq_P$ TRATG-Cover.*

Proof. Treat words as terms with unary function symbols and vice versa. □

Regular-Cover corresponds to TRATG-Cover, and is in NP. Furthermore, if Regular-Cover is NP-hard then so is TRATG-Cover. However the precise complexity of Regular-Cover is open, similar to the case for terms we conjecture it to be NP-complete.

Open Problem 3.7.2. Is Regular-Cover NP-complete?

Clearly, Open Problem 3.7.2 implies Open Problem 3.7.1. In the rest of this section, we will show that the restriction of Regular-Cover to a bounded number of nonterminals is NP-hard.

Problem 3.7.6 (Regular-n-Cover)**.**
Given a finite set of words L and $k \geq 0$, is there an acyclic regular grammar $G = (N, \Sigma, P, \mathrm{A})$ such that $|N| \leq n$, $|P| \leq k$, and $L(G) \supseteq L$?

We will show the NP-hardness of Regular-n-Cover in several steps: first we reduce 3SAT to Regular-2-Cover using an intermediate problem where we can not only specify words that must be generated by the grammar, but also productions that must be included:

Problem 3.7.7 (Regular-2-Cover-Extension)**.**
Given a finite set of words L, an acyclic regular grammar $G = (N, \Sigma, P, \mathrm{A})$ such that $N = \{\mathrm{A}, \mathrm{B}\}$, w contains B whenever $\mathrm{A} \longrightarrow w \in P$, and $k \geq 0$, is there a superset $P' \supseteq P$ of the productions such that $|P'| \leq |P| + k$, and $L(G') \supseteq L$ where $G' = (N, \Sigma, P', \mathrm{A})$?

Lemma 3.7.3. *3SAT $\leq_P$ Regular-2-Cover-Extension.*

Proof. Consider a propositional formula in CNF with m clauses $C_1, \ldots, C_m$ and n variables (called $x_1, \ldots, x_n$). The number of variables n will be used as the k parameter in Regular-2-Cover-Extension. Assume without loss of generality that $x_j \vee \neg x_j$ is a clause in this formula for all $j \leq n$. We will encode the literals as unary natural numbers. First we define the natural numbers $a_j = 2j$ and $b_j = 2j + 1$ that correspond to x_j and $\neg x_j$, resp. The number $c = 2n + 1$ is their upper bound.

Let $L = \{\mathtt{s}^c \mathtt{o}_{l,i} \mid i \leq m, l \leq 2n\}$ and $\Sigma = \{\mathtt{o}_{l,i} \mid l \leq 2n, i \leq n\} \cup \{\mathtt{s}\}$. Furthermore, define the acyclic regular grammar $G = (N, \Sigma, P, \mathrm{A})$ where

$N = \{\mathrm{A}, \mathrm{B}\}$, and the productions P are the following:

$$\begin{aligned} \mathrm{B} &\to \mathsf{s}^{c-a_j}\mathsf{o}_{l,i} && \text{for } x_j \in C_i \text{ and } l \leq 2n \\ \mathrm{B} &\to \mathsf{s}^{c-b_j}\mathsf{o}_{l,i} && \text{for } \neg x_j \in C_i \text{ and } l \leq 2n \end{aligned}$$

It remains to show that F is satisfiable iff there exists a set of productions $P' \supseteq P$ such that $|P'| \leq |P| + n$ and $L(G') \supseteq L$ where $G' = (N, \Sigma, P', \mathrm{A})$.

Left-to-right direction. Let $I \models F$ be a satisfying interpretation, then we construct an acyclic regular grammar G' by adding the following productions to G. We clearly have $|G'| = |G| + n$, and also $L(G') \supseteq L$.

$$\mathrm{A} \to \mathsf{s}^{a_j}\mathrm{B} \quad \text{if } I \models x_j \qquad\qquad \mathrm{A} \to \mathsf{s}^{b_j}\mathrm{B} \quad \text{if } I \not\models x_j$$

Right-to-left direction. Let $G' = (N, \Sigma, P', \mathrm{A})$ such that $P' \supseteq P$, $L(G') \supseteq L$, and $|P'| \leq |P| + n$. By symmetry, all productions from the nonterminal A are of the form $\mathrm{A} \to \mathsf{s}^r\mathrm{B}$ for some $r \geq 0$. Otherwise there would be an $i \leq m$ such that all constants $\mathsf{o}_{l,i}$ appear in productions from A, and we would have at least $2n$ new productions.

Let δ be a derivation of $\mathsf{s}^c\mathsf{o}_{l,i} \in L$ in G'. Then δ uses the production $\mathrm{A} \to \mathsf{s}^r\mathrm{B}$ for some r, and we have that $r = a_j$ and $x_j \in C_i$ or $r = b_j$ and $\neg x_j \in C_i$. Since $x_j \vee \neg x_j \in F$ for all $j \leq n$, we have $\mathrm{A} \to \mathsf{s}^{a_j}\mathrm{B} \in P'$ or $\mathrm{A} \to \mathsf{s}^{b_j}\mathrm{B} \in P'$ for all $j \leq n$. Because there are at most n such productions, we cannot have both $\mathsf{s}^{a_j}\mathrm{B} \in P'$ and $\mathsf{s}^{b_j}\mathrm{B} \in P'$.

Now define an interpretation I such that $I \models x_j$ iff $\mathrm{A} \to \mathsf{s}^{a_j}\mathrm{B} \in P'$. Note that $I \not\models x_j$ iff $\mathrm{A} \to \mathsf{s}^{b_j}\mathrm{B} \in P'$, and hence I is a model for F by the previous paragraph. □

Lemma 3.7.4. *Regular-2-Cover-Extension* $\leq_P$ *Regular-2-Cover.*

Proof. Let L, $G = (N, \Sigma, P, \mathrm{A})$, $N = \{\mathrm{A}, \mathrm{B}\}$, and k be as in the definition of Regular-2-Cover-Extension. We set $m = |L| + |G|$. Without loss of generality, assume that $L \neq \emptyset$. Take $4m$ fresh letters $\mathsf{a}_1, \ldots, \mathsf{a}_{3m}, \mathsf{b}_1, \ldots, \mathsf{b}_m$. We extend the language L to L':

$$\begin{aligned} L' = L &\cup \{\mathsf{a}_i w \mid \mathrm{B} \to w \in P,\ i \leq 3m\} \\ &\cup \{w\mathsf{b}_j \mid \mathrm{A} \to w\mathrm{B} \in P,\ j \leq m\} \\ &\cup \{\mathsf{a}_i\mathsf{b}_j \mid i \leq 3m, j \leq m\} \end{aligned}$$

We need to construct an acyclic regular grammar $G' = (N, \Sigma', P', \mathrm{A})$ with $\Sigma' = \Sigma \cup \{\mathsf{a}_1, \ldots, \mathsf{a}_{3m}, \mathsf{b}_1, \ldots, \mathsf{b}_m\}$ such that $L(G') \supseteq L'$ and $|G'| \leq |G| + k + 4m$ if and only if there exists an acyclic regular grammar $G'' = (N, \Sigma, P'', \mathrm{A})$ such that $P \subseteq P''$, $|G''| \leq |G| + k$, and $L(G') \supseteq L$.

Left-to-right direction. Let us first assume that such a grammar G' exists. We can assume that $|G'| \leq |L| + |G| + 4m = 5m$ since there exists a covering grammar of that size with the productions $\{\mathrm{A} \to w \mid w \in L\} \cup P \cup \{\mathrm{A} \to \mathsf{a}_i \mid i \leq 3m\} \cup \{\mathrm{B} \to \mathsf{b}_j \mid j \leq m\}$.

Without loss of generality, assume that the letters a_i occur only as the first letter of productions of the nonterminal A. We can drop any production that contains a_i in the middle since a_i can only occur at the beginning of words in L'. For the same reason we can then replace each production $\mathrm{B} \to \mathsf{a}_i w$ by $\mathrm{A} \to \mathsf{a}_i w$.

Furthermore, we can assume that G' is symmetric in the new symbols, that is, $\mathrm{A} \to \mathsf{a}_i w \in P'$ if and only if $\mathrm{A} \to \mathsf{a}_j w \in P'$ for all $i, j \leq 3m$ and words w. Otherwise pick an $i \leq 3m$ such that $W = \{w \mid \mathrm{A} \to \mathsf{a}_i w \in P'\}$ is of minimal size. We then remove all productions containing an a_j for some $j \leq 3m$, and replace them by the productions $\mathrm{A} \to \mathsf{a}_j w$ for $w \in W$. The resulting grammar still covers L' since L' is symmetric under permutation of the letters a_i.

Now for every $i \leq 3m$ there is at most one production of the form $\mathrm{A} \to \mathsf{a}_i w$ for some w—if there were two, then by a_i-symmetry we would have $2 \cdot 3m = 6m$ productions, exceeding the previously obtained upper bound of $5m$ productions. This also implies that $\mathrm{B} \to \mathsf{b}_j \in P'$ for all $j \leq m$. By a symmetry argument for b_j there are no other productions that contain b_j. We also have that $\mathrm{B} \to w \in P'$ whenever $\mathrm{B} \to w \in P$.

We can now construct the acyclic regular grammar G'' by removing all productions from G' that contain one of the new symbols a_i or b_j. We have $L(G'') \supseteq L$ since $L(G') \supseteq L$ and because productions that contain the new symbols cannot be used in derivations of words in L. Furthermore, $|G''| \leq |G'| - 4m = |G| + k$. It remains to show that $\mathrm{A} \to w\mathrm{B} \in P'$ whenever $\mathrm{A} \to w\mathrm{B} \in P$ for some $w \in \Sigma^*$. Recall that L' contains $w\mathsf{b}_j$ for all $j \leq m$. However the only occurrence of b_j in P' is in the production $\mathrm{B} \to \mathsf{b}_j \in P'$. Hence $\mathrm{A} \to w\mathrm{B} \in P'$.

Right-to-left direction. In the other case, we assume that such a grammar G'' exists, and need to construct G'. We obtain this grammar G' by adding the

productions $A \to a_i B$ and $B \to b_j$ for all $i \leq 3m$ and $j \leq m$. Then G' has $4m$ more productions than G'' and covers L'. □

Lemma 3.7.5. *Regular-n-Cover* $\leq_P$ *Regular-*$(n+1)$*-Cover.*

Proof. Let L be a finite set of words, $k \geq 0$, and define $m = 2k$. Assume that $|L| \geq 1$, $k \geq 2$, and $n \geq 1$—otherwise we can directly compute the answer. Take $m+1$ fresh letters $a_1, \ldots, a_m, b$. We define $L' = \{a_1, \ldots, a_m\} \cdot (L \cup \{b\})$. It remains to show that there exists an acyclic regular grammar G with n nonterminals such that $|G| \leq k$ and $L(G) \supseteq L$ if and only if there exists an acyclic regular grammar G' with $n+1$ nonterminals such that $|G'| \leq k+m+1$ and $L(G') \supseteq L'$.

Left-to-right direction. Let $G = (N, \Sigma, P, B)$ be an acyclic regular grammar such that $|N| = n$, $|G| \leq k$, and $L(G) \supseteq L$. We set $G' = (N \cup \{A\}, \Sigma \cup \Sigma', P \cup P', A)$ where A is a fresh nonterminal, $\Sigma' = \{a_1, \ldots, a_m, b\}$, and $P' = \{A \to a_i B \mid i \leq m\} \cup \{B \to b\}$. Clearly $|N \cup \{A\}| = n+1$, $|G'| = |G| + m + 1 \leq k+m+1$, and $L(G') \supseteq L'$.

Right-to-left direction. Let $G' = (N, \Sigma, P', A)$ be an acyclic regular grammar such that $|N| = n+1$, $|G| \leq k+m+1$, and $L(G') \supseteq L'$. Via the same argument as used in the proof of Lemma 3.7.4 and noting that $2m > k+m$, we can assume that there exists a nonterminal $B \neq A$ such that all productions from A are of the form $A \to a_i B$ for some $i \leq m$. Let $P \subseteq P'$ be the set of productions whose left side is not A and whose right side is not b. The acyclic regular grammar $G = (N \setminus \{A\}, \Sigma, P, B)$ then has the desired properties. □

Theorem 3.7.3. *Regular-n-Cover is* NP*-complete for* $n \geq 2$*.*

Proof. By reduction from 3SAT using Lemmas 3.7.3 to 3.7.5. □

Proof (of Theorem 3.7.2, the NP*-completeness of TRATG-n-Cover).* Follows directly from Theorem 3.7.3 by representing words as terms and vice versa. □

3.8 Minimization

Problem 3.8.1 (VTRATG-Minimization)**.**
Given a VTRATG $G = (N, \Sigma, P, A)$, a set of terms L such that $L(G) \supseteq L$, and

$k \geq 0$, is there a subset $P' \subseteq P$ of the productions such that $|P'| \leq k$ and $L(G') \supseteq L$ where $G' = (N, \Sigma, P', \mathrm{A})$?

This optimization problem will play a central role in the MaxSAT algorithm to find minimal covering VTRATGs that we will see in Section 4.3. We will now show that VTRATG-Minimization is NP-complete via reduction from Set Cover.

Problem 3.8.2 (Set Cover)**.**
Given a finite set X, a finite collection $C \subseteq \mathcal{P}(X)$ of subsets such that $\bigcup C = X$, and $k \geq 0$, is there a sub-collection $C' \subseteq C$ such that $\bigcup C = X$ and $|C'| \leq k$?

Theorem 3.8.1. *VTRATG-Minimization is* NP*-complete.*

Proof. The problem is in NP because we can check $|P'| \leq k$ in polynomial time, and reduce $L(G') \supseteq L$ to VTRATG-Membership. Hardness follows by reduction from Set Cover: we pick a fresh nonterminal A, set $N = \{\mathrm{A}\} \cup C$ (treating the subsets as nonterminals), $L = X$, and $P = \{\mathrm{A} \to U \mid U \in C\} \cup \{U \to x \mid x \in U \in C\}$. A subset $P' \subseteq P$ of the productions of with $k + |X|$ elements then directly corresponds to a sub-collection C' of such that $|C'| \leq k$: for every $x \in X$, there is at least one production $U \to x \in P'$ for some $U \in C$. These are $|X|$ productions, there are hence at most k productions of the form $\mathrm{A} \to U$ for some U, which yield the sub-collection $C' = \{U \mid \mathrm{A} \to U \in P'\}$. □

3.9 Decision problems on Herbrand disjunctions

Each of these decision problems on VTRATGs corresponds to a decision problem on simple proofs: for instance, VTRATG-Membership decides whether a formula is contained in the Herbrand disjunction of a non-erasing cut-normal form.

As we have seen in Lemma 2.7.5, in general, cut-elimination can decrease the language of the grammar of a proof. This is due to the reduction of weakening inferences, which can delete parts of the proof and hence remove instances. However, without reduction of weakenings, the language is preserved by cut-elimination:

Definition 3.9.1 (non-erasing cut reduction, [53])**.** $\overset{\text{ne}}{\longrightarrow}$ is the cut-reduction relation without reduction on weakening inferences.

Theorem 3.9.1 ([53])**.** *Let π, π' be simple proofs. If $\pi \overset{\text{ne}}{\longrightarrow} \pi'$, then $L(G(\pi)) = L(G(\pi'))$.*

If π is a simple proof, and π^* a non-erasing cut-normal form such that $\pi \overset{\text{ne}}{\longrightarrow}^* \pi^*$, then $L(\pi^*) = L(G(\pi))$. Any simple proof has a $\overset{\text{ne}}{\longrightarrow}$-normal form. If in addition, π only contains weakening inferences introducing quantifier-free formulas, then π^* does not contain quantified cuts. Given a $\overset{\text{ne}}{\longrightarrow}$-normal form π^* of a proof π with only quantifier-free weakening, we can hence directly extract a Herbrand disjunction $H(\pi^*)$ such that $H(\pi^*) = L(G(\pi^*)) = L(G(\pi))$.

Problem 3.9.1 (H-Membership)**.**
Given a simple proof π with only quantifier-free weakening and a formula φ, is there a $\overset{\text{ne}}{\longrightarrow}$-normal form $\pi \overset{\text{ne}}{\longrightarrow}^* \pi^*$ such that $\varphi \in H(\pi^*)$?

Problem 3.9.2 (H-Containment)**.**
Given a simple proofs π_1, π_2 with only quantifier-free weakening, are there $\overset{\text{ne}}{\longrightarrow}$-normal forms $\pi_i \overset{\text{ne}}{\longrightarrow} \pi_i^*$ for $i \in \{1, 2\}$ such that $H(\pi_1^*) \subseteq H(\pi_2^*)$?

Problem 3.9.3 (H-Disjointness)**.**
Given a simple proofs π_1, π_2 with only quantifier-free weakening, are there $\overset{\text{ne}}{\longrightarrow}$-normal forms $\pi_i \overset{\text{ne}}{\longrightarrow} \pi_i^*$ for $i \in \{1, 2\}$ such that $H(\pi_1^*) \cap H(\pi_2^*) = \emptyset$?

Problem 3.9.4 (H-Equivalence)**.**
Given a simple proofs π_1, π_2 with only quantifier-free weakening, are there $\overset{\text{ne}}{\longrightarrow}$-normal forms $\pi_i \overset{\text{ne}}{\longrightarrow} \pi_i^*$ for $i \in \{1, 2\}$ such that $H(\pi_1^*) = H(\pi_2^*)$?

The restriction of only allowing weakening on quantifier-free corresponds to a restriction on the productions on the VTRATG: for every nonterminal vector $\overline{B}$ there has to be at least one production $\overline{B} \to \overline{t}$.

Lemma 3.9.1. *There is a formula $\varphi(x)$ containing x such that: let G be a VTRATG such that there is a production for every nonterminal vector and every nonterminal vector is nonempty. Then there is a simple proof π_G of $\vdash \exists x\, \varphi(x)$ such that π_G does not contain quantified weakening and $L(\pi_G^*) = \{r_1(t) \mid t \in L(G)\}$ for any $\overset{\text{ne}}{\longrightarrow}$-normal form π_G^*. This proof π_G is polynomially-time computable from G.*

Proof. Choose $\varphi(x) := R(x) \to R(x)$ for a relation symbol R. We could also use $\top$ instead of $\varphi(x)$, but then the terms would not appear in the formulas. Let $A \prec \overline{B_1} \prec \cdots \prec \overline{B_n}$ be the nonterminal vectors of G. The proof π_G then has n cuts, each with the formula $\phi_i = \forall \overline{x_i} \neg \bigvee_j \varphi(x_{i,j})$ where $\overline{x_i}$ is a variable vector of the same size and types as the nonterminal vector $\overline{B_i}$. For the cut with the cut formula ϕ_i we will use the eigenvariable vector $\overline{\alpha_i}$. We define a sequence of proofs π_i for $1 \leq i \leq n+1$. For $i \leq n$, π_i is defined as follows where $\overline{B_i} \to \overline{t_1}, \ldots, \overline{B_i} \to \overline{t_m}$ are all the productions for the nonterminal vector $\overline{B_i}$:

$$\dfrac{\dfrac{\dfrac{(\pi_{i+1})}{\vdash \exists x\, \varphi(x)}}{\vdash \neg \bigvee_j \varphi(\alpha_{i,j}), \exists x\, \varphi(x)}\, w_r}{\vdash \phi_i, \exists x\, \varphi(x)}\, \forall_r^* \qquad \dfrac{\dfrac{\dfrac{\dfrac{R(t_{1,1}) \vdash R(t_{1,1})}{\vdash \varphi(t_{1,1})}\, \to_r}{\vdash \bigvee_j \varphi(t_{1,j})}\, w_r, \vee_r}{\neg \bigvee_j \varphi(t_{1,j}) \vdash}\, \neg_l^*}{\neg \bigvee_j \varphi(t_{1,j}), \ldots, \neg \bigvee_j \varphi(t_{1,j}) \vdash}\, w_l^* \qquad \dfrac{}{\phi_i \vdash}\, \forall_l^*, c_l^*$$

$$\psi_1, \ldots, \psi_{i-1} \vdash \exists x\, \varphi(x)\ \text{cut}$$

We define π_{n+1} as follows where $A \to t_1 \mid \cdots \mid t_m$ are all the productions for the nonterminal A:

$$\dfrac{\dfrac{\dfrac{R(t_1) \vdash R(t_1)}{\vdash \varphi(t_1)}\, \to_r}{\vdash \varphi(t_1), \ldots, \varphi(t_m)}\, w_r^*}{\vdash \exists x\, \varphi(x)}\, \exists_r^*, c_r^*$$

Finally we set $\pi_G = \pi_1$. By looking at π_G, we have $G(\pi_G) \simeq G$, i.e., G has the production $A \to t$ iff $G(\pi_G)$ has the production $A \to r_1(t)$. We then have $L(\pi_G^*) = L(G(\pi_G)) = \{r_1(t) \mid t \in L(G)\}$ by Theorem 3.9.1. □

A similar (and less artificial) construction is also present in [46, Theorem 25], however that construction gives a proof of a larger sequent $\varphi_1, \ldots, \varphi_n \vdash \exists x\, R_1(x)$ where the formula φ_i depends on the productions for the nonterminal B_i and the instances of φ_i encode the terms derivable from B_i. The Herbrand sequent of the proof hence contains more terms than the language of the grammar, and for example $L(G_1) \subseteq L(G_2)$ would in general not be equivalent to $H(\pi_{G_1}^*) \subseteq H(\pi_{G_2}^*)$ where π_{G_1}, π_{G_2} are defined as in [46].

Theorem 3.9.2. *The following complexity results hold:*

- *H-Membership is* NP*-complete*
- *H-Containment is* Π_2^P*-complete*
- *H-Disjointness is* coNP*-complete*
- *H-Equivalence is* Π_2^P*-complete*

Proof. Hardness follows from the corresponding Theorems 3.2.1, 3.4.1, 3.5.1 and 3.6.1 for VTRATGs via Lemma 3.9.1, since we have $H(\pi_G^*) = E(L(\pi_G^*)) = E(L(G(\pi_G))) = \{r_1(t) \mid t \in L(G)\} \simeq L(G)$ and all the VTRATGs used in the hardness proofs have the property that every nonterminal vector has at least one production. □

3.10 The treewidth measure on graphs

Many problems on graphs are easier to solve on trees. For example coloring a graph using a given number of colors (such that no two adjacent vertices have the same color) is a well-known NP-complete problem; however it is trivial to color a tree using just two colors. We could expect that such problems can also be easily solved on graphs that are in some sense "close" to being a tree. The notion of treewidth makes this intuition concrete. The treewidth of a graph is a natural number that indicates how close it is to a tree: non-empty trees (as well as forests) have treewidth 1, and graphs with larger tree widths are less like trees. The concept of treewidth was originally introduced under the name "dimension" in [10] to study tractable subclasses of optimization problems, and later called treewidth in [84].

Definition 3.10.1 (tree decomposition)**.** Let $G = (V, E)$ be an undirected graph. Then a pair $(T, (X_t)_{t \in V_T})$ of a tree $T = (V_T, E_T)$ together with a function $X \colon V_T \to \mathcal{P}(V)$, which assigns to every node of the tree a "bag" of vertices of G, is called a tree decomposition of G if:

1. For any vertex $v \in V$, there exists a vertex $t \in V_T$ such that $v \in X_t$.
2. For any edge $e = \{v_1, v_2\} \in E$, there exists a vertex $t \in V_T$ such that $e \subseteq X_t$.

3. Let $t_1 \dots t_n$ be a path in T, and $v \in X_{t_1} \cap X_{t_n}$. Then $v \in X_{t_i}$ for any t_i on the path.

The width of a tree decomposition T is the maximum of $|X_t| - 1$ for $t \in V_T$.

Definition 3.10.2 (treewidth)**.** Let G be an undirected graph. Then $\mathsf{tw}(G)$ is the minimum width of a tree decomposition of G.

The treewidth of a graph is 1 if and only if it is a forest. The treewidth is 0 if and only if the graph has no edges. For every $k \geq 0$, there is a polynomial-time algorithm that computes a tree decomposition of width $\leq k$, if such a decomposition exists [6]. If k is not fixed, the problem of determining whether a given graph has treewidth k is NP-complete [6]. Let us first prove some basic facts about treewidth.

Lemma 3.10.1. *Let $n \geq 1$ and let K_n be the complete graph on n vertices. Then $\mathsf{tw}(K_n) = n - 1$.*

Proof. It is easy to see that $\mathsf{tw}(K_n) \leq n - 1$ using a tree decomposition with tree consisting of a single vertex. For $\mathsf{tw}(K_n) \geq n - 1$, let $(T, (X_t)_{t \in V_T})$ be a tree decomposition of K_n. We need to show that this tree decomposition has at least width $n - 1$. If T contains a leaf v connected to T via the edge $\{v, v'\}$, such that $X_v \neq V_{K_n}$, then we can make a smaller tree decomposition by removing v: let $i \in V_{K_n} \setminus X_v$. In order to show that $(T \setminus \{v\}, X)$ is still a tree decomposition, we need to show that for every edge $\{w, w'\} \in E_{K_n}$, we still have a node $\tilde{v} \in V_T \setminus \{v\}$ such that $\{w, w'\} \subseteq X_{\tilde{v}}$. Since (T, X) is a tree decomposition there exist $v_1, v_2 \in V_T$ such that $\{w, i\} \subseteq X_{v_1}$ and $\{w', i\} \subseteq X_{v_2}$. If $\{w, w'\} \subseteq X_v$ of the removed node, then $\{w, w'\} \subseteq X_{v'}$ since the adjacent node lies on the paths to v_1, v_2. □

Lemma 3.10.2. *Let G, H be graphs such that G is a subgraph of H. Then $\mathsf{tw}(G) \leq \mathsf{tw}(H)$.*

Proof. Every tree decomposition of H induces a tree decomposition of G of the same width or less, by restricting the X_v sets to the vertices of G. □

3.11 The case of bounded treewidth

The NP-hardness result of VTRATG-Membership in Theorem 3.2.1 was based on the observation that we can encode Boolean satisfiability as the TRATG $\mathrm{Sat}_{n,m}$. This TRATG makes use of the fact that derivations in VTRATGs can only use a single production for each nonterminal, and we can hence "synchronize" across many productions to choose a consistent interpretation. There is a very relevant natural class of VTRATGs where this synchronization occurs in only a very limited amount, namely instance grammars:

Example 3.11.1. Consider the following induction grammar G:

$$\begin{aligned} \tau &\to r(\gamma) \\ \gamma &\to c \mid f(\gamma) \end{aligned}$$

Then the instance grammar $I(G, s^n(0))$ has the following form:

$$\begin{aligned} \tau &\to r(\gamma_0) \mid \cdots \mid r(\gamma_n) \\ \gamma_0 &\to c \mid f(\gamma_1) \\ \gamma_1 &\to c \mid f(\gamma_2) \\ &\vdots \\ \gamma_{n-1} &\to c \mid f(\gamma_n) \\ \gamma_n &\to c \end{aligned}$$

Note that the productions for each nonterminal γ_i only contain the "next" nonterminal γ_{i+1}. The treewidth will hence be small. Concretely, the treewidth will be at most 2 for each instance grammars $I(G, s^n(0))$.

In this section we will show that the notion of treewidth captures how much of this kind of synchronization can occur across nonterminals in a VTRATG: if the treewidth is bounded, then the membership problem will be computable in polynomial time.

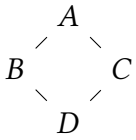
For VTRATGs we consider the treewidth of a so-called dependency graph of the grammar. (For TRATGs, the directed version of this dependency graph is acyclic by Definition 2.5.1, since it embeds into the linear order $\prec$ of the VTRATG.) Given a set of nonterminal vectors $N' \subseteq N$, we write $\bigcup N' = \{B_i \mid \overline{B} \in N'\}$ for the set of all nonterminals contained in N_a.

Definition 3.11.1 (dependency graph)**.** Let $G = (N, \Sigma, P, A)$ be a TRATG. Then the dependency graph $D(G) = (\bigcup N, E)$ has the nonterminals as vertices, and two nonterminals B, C are adjacent if there exists a production $\overline{B}' \to t \in P$ and j such that t contains C and $B = B'_i$ for some i, or if $B = D_i$ and $C = D_j$ for some nonterminal vector $\overline{D}$ and $i \neq j$.

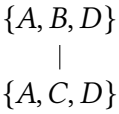
Example 3.11.2. Consider the TRATG G with the productions:

$$A \to f(B, C)$$
$$B \to D$$
$$C \to D$$
$$D \to c \mid d$$

This TRATG G has the following dependency graph $D(G)$:

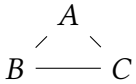


The dependency graph $D(G)$ has treewidth $\mathrm{tw}(D(G)) = 2$, witnessed by the following tree decomposition of width 2 (labelling each vertex t in the tree decomposition with its associated set X_t):



Intuitively, the nonterminal D occurs in both the productions for B and for C. Hence the two nonterminals B and C need to "synchronize" the result for D. In the tree decomposition this causes D to be in every X_t set.

Example 3.11.3. Let G be the VTRATG with the productions $A \to f(B, C)$ and $(B, C) \to (c, d)$. Then G has the following dependency graph:



The dependency graph has treewidth $\mathrm{tw}(D(G)) = 2$, as witnessed by the tree decomposition $\{A, B, C\}$ (consisting of a single vertex and no edges).

Example 3.11.4. Bounding $\mathrm{tw}(D(G))$ restricts the amount of synchronization, but not the size of the generated language. The generated language $L(G)$ can still be exponentially larger, even if $\mathrm{tw}(D(G)) = 1$ as in the following grammar of size $2n + 1$ whose language has size $|L(G)| = 2^n$:

$$\begin{aligned} A &\to f(B_1) \mid g(B_1) \\ B_1 &\to f(B_2) \mid g(B_2) \\ &\vdots \\ B_{n-1} &\to f(B_n) \mid g(B_n) \\ B_n &\to c \end{aligned}$$

Lemma 3.11.1. *Let $G = (N, \Sigma, P, A)$ be a VTRATG. Then $|\overline{B}| \leq \mathrm{tw}(D(G)) + 1$ for all $\overline{B} \in N$.*

Proof. Let $\overline{B} = (B_1, \ldots, B_n)$. Then B_i and B_j are adjacent in $D(G)$ for every $i \neq j$, so $D(G)$ contains an isomorphic copy of K_n as a subgraph. If $n = 0$, then trivially $n \leq \mathrm{tw}(D(G)) + 1$. Otherwise, $|\overline{B}| = n = \mathrm{tw}(K_n) + 1 \leq \mathrm{tw}(D(G)) + 1$ by Lemmas 3.10.1 and 3.10.2. □

There is a good reason why we define the dependency graph on nonterminals and not on the nonterminal vectors. Call the analogously defined dependency graph on nonterminal vectors $D'(G)$, i.e., the vertices of $D'(G)$ are the nonterminal vectors, and two nonterminal vectors $\overline{B}$ and $\overline{C}$ are adjacent iff there exists a production $\overline{B} \to \bar{t}$ such that $\bar{t}$ contains C_j for some j. Then for every VTRATG G there is a (polynomial-time computable) VTRATG G' such that $L(G) = L(G')$ and $\mathrm{tw}(D'(G')) = 1$. That is, even for bounded $\mathrm{tw} \circ D'$ the membership problem is already NP-complete by Theorem 3.2.1. The construction of this VTRATG G' is straightforward: let G have the nonterminal vectors $A_0 \prec \overline{A_1} \prec \cdots \prec \overline{A_n}$ (and without loss of generality, let there be at least one production for every nonterminal vector). Then G' has the nonterminal vectors $B_0 \prec \overline{B_1} \prec \cdots \prec \overline{B_n}$, where $\overline{B_i} = (\overline{B_{i,i}}, \ldots, \overline{B_{i,n}})$ such that $i \geq 1$ and $|\overline{B_{i,j}}| = |\overline{A_j}|$. That is, $\overline{B_i}$ consists of copies of the nonterminal vectors $\overline{A_i}, \ldots, \overline{A_n}$,

stacked on each other. A production $\overline{A_i} \to \bar{t}[\overline{A_{i+1}}, \ldots, \overline{A_n}]$ in G is translated to a production $\overline{B_i} \to (t[\overline{B_{i+1,i+1}}, \ldots \overline{B_{i+1,n}}], \overline{B_{i+1,i+1}}, \ldots, \overline{B_{i+1,n}})$. Clearly $D'(G')$ is a tree and hence $\mathrm{tw}(D'(G')) = 1$. This construction even preserves the size of the VTRATG $|G'| = |G|$. Defining the dependency graph on nonterminals also allows us to bound the length of a nonterminal vector by the treewidth.

3.11.1 Membership

Bounding the treewidth of the dependency graph restricts the amount of synchronization that can occur between nonterminals in a VTRATG. This restriction allows us to decide the membership problem in polynomial time if the treewidth is bounded.

Problem 3.11.1 ($(\mathrm{tw} \leq k)$-Membership)**.**
Given a term t and a VTRATG G such that $\mathrm{tw}(D(G)) \leq k$, is $t \in L(G)$?

For the rest of this subsection, fix a term t, a VTRATG $G = (N, \Sigma, P, A)$, and a tree decomposition $T = (V_T, E_T)$ of $D(G)$ with bags X_v for $v \in V_T$. We will construct a polynomial-time decision procedure for $(\mathrm{tw} \leq k)$-Membership. Choose a root $v_0 \in V_T$ of the tree decomposition such that $A \in X_{v_0}$ and orient the edges in T away from the root. In this way, every vertex $v \in V_T$ induces a subtree T_v containing all vertices reachable from v.

The decision procedure will be implemented by recursion on the tree decomposition: starting with the root of the decomposition, we determine all terms generated by the nonterminals X_v. The possible assignments from nonterminals to terms are captured by the following definition. Given a set of nonterminals $M \subseteq \bigcup N$, we define the set $\tilde{M} \subseteq N$ of all nonterminal vectors that contain some nonterminal in M, i.e., $\tilde{M} = \{\overline{B} \in N \mid \exists i\, B_i \in M\}$.

Definition 3.11.2. Let $v \in V_T$ be a node in the tree decomposition. A *partial assignment for* v is a pair (f, p) such that:

1. $f \subseteq X_v \times \mathrm{st}(t)$ is a partial function,
2. $p \subseteq P$ is a partial function such that $\mathrm{dom}(p) \subseteq \tilde{X_v}$,
3. for every production $\overline{B} \to \bar{s} \in p$ and i such that $B_i \in \mathrm{dom}(f)$:

a) $f(B_i) \preceq s_i$

b) $f(C) = C(f(B_i)/s_i)$ for every nonterminal C such that C occurs in s_i

Example 3.11.5 (continuing Example 3.11.2). Let v be the vertex in the tree decomposition with $X_v = \{A, B, D\}$, and let $t = f(c, d)$ be the term where we want to decide $t \in L(G)$. Then $(\{(A, f(c, d)), (B, c), (D, c)\}, \{A \to f(B, C), B \to D, D \to c\})$ is a partial assignment for v.

In a partial assignment, the terms $f_a(B)$ are the terms that we want to parse (where $B \in \bigcup N$ is a nonterminal), that is, where we want to find a derivation δ such that $B\delta = f_a(B)$. It is important that the assignment from nonterminals to terms is partial: consider a term $f(c)$ and a VTRATG with the productions $A \to f(B) \mid f(C), B \to c, C \to d$. Here, there is no subterm $s \trianglelefteq f(c)$ such that C expands to s.

Furthermore, the partial assignment fixes the productions used in the derivation. This is due to another reason: we might have a term $f(c, c)$, a VTRATG with the productions $A \to f(B, C) \mid f(C, B), B \to c, C \to d$ and a tree decomposition $\{A, B\} - \{A\} - \{A, C\}$. If we did not include the production for A in the partial assignment, then the partial assignments for $\{A, B\}$ and $\{A, C\}$ could disagree about the production for A. If two partial assignments agree on the productions and terms for the common nonterminals, then we call them compatible:

Definition 3.11.3. Let (f, p) and (f', p') be partial assignments for $v, v' \in V_T$, resp. The partial assignments are *compatible* iff $f \restriction (X_v \cap X_{v'}) = f' \restriction (X_v \cap X_{v'})$ and $p \restriction (\tilde{X}_v \cap \tilde{X}_{v'}) = p' \restriction (\tilde{X}_v \cap \tilde{X}_{v'})$.

We now give a characterization of $L(G)$ in terms of partial assignments on the tree decomposition. In a sense, we split up a derivation into small overlapping parts, and each node v in the tree decomposition stores the part about the nonterminals X_v:

Theorem 3.11.1. *$t \in L(G)$ iff there exists a function a that assigns to every vertex $v \in V_T$ of the tree decomposition a partial assignment $a(v)$ for v in such a way that $a(v)$ and $a(v')$ are compatible for all adjacent v and v', and $(a(v))_1(A) = t$ for some v.*

Proof. First, assume that $t \in L(G)$ and δ is a derivation of t in G. Then we define $a(v) = (f_v, p_v)$ as follows: $f_v(B) = B\delta$ if $B \in X_v$ and $B\delta \trianglelefteq t$, and undefined otherwise. For productions, we set $\overline{C} \to \overline{s} \in p_v$ iff $\overline{C} \in \tilde{X_v}$ and $\overline{C} \to \overline{s} \in \delta$. Clearly $a(v)$ is a partial assignment for every v, and all of these partial assignments are compatible.

On the other hand, let a be a function as required by the theorem. Let f and p be partial functions such that $(a(v))_1 = f \restriction X_v$ and $(a(v))_2 = p \restriction \tilde{X_v}$ for all $v \in V_T$. The partial function f exists, as the function values in the partial assignments need to agree for adjacent vertices and all vertices where X_v contains a given nonterminal are connected in T since T is a tree decomposition. We clearly have $f(A) = t$. Furthermore, given $B \in N$ and i, j, all vertices v such that X_v contains B_i or B_j are connected (by definition of the dependency graph). Hence the partial function p exists as well. From f and p we can now define a derivation δ using the set of productions p. We now have $f(B) = B\delta$ for every nonterminal $B \in \bigcup N$ and hence δ is a derivation of t in G. □

Example 3.11.6 (continuing Example 3.11.2). Let $t = f(c, c)$. Then $t \in L(G)$ as witnessed by the derivation using the productions $A \to f(B, C), B \to D, C \to D, D \to c$. The corresponding function a is given as follows (where v_0, v_1 are the vertices such that $X_{v_0} = \{A, B, D\}$ and $X_{v_1} = \{A, C, D\}$):

$$a(v_0) = (\{(A, f(c, c)), (B, c), (D, c)\}, \{A \to f(B, C), B \to D, D \to c\})$$
$$a(v_1) = (\{(A, f(c, c)), (C, c), (D, c)\}, \{A \to f(B, C), C \to D, D \to c\})$$

We can now use the characterization of Theorem 3.11.1 to show that (tw $\leq k$)-Membership is computable in polynomial time.

Theorem 3.11.2. (tw $\leq k$)-*Membership* $\in$ P *for all* k.

Proof. We compute for every $v \in V_T$ the set A_v of partial assignments for v such that $b \in A_v$ iff there exists a function a that assigns to every $v' \in V_{T_v}$ a partial assignment for v' such that $a(v) = b$ and $a(v')$ and $a(v'')$ are compatible whenever v' and v'' are adjacent. This set can be computed recursively: given a partial assignment b for v, $b \in A_v$ iff for every child v' of v there exists a compatible partial assignment $b' \in A_{v'}$. There are only polynomially many partial assignments for every v, so the recursive algorithm runs in polynomial

time. By Theorem 3.11.1, $t \in L(G)$ if and only if there exists a $(f, p) \in A_v$ with $f(A) = t$. □

3.11.2 Emptiness

We can also use Theorem 3.11.1 to effectively decide the emptiness problem for VTRATGs with dependency graphs of bounded treewidth:

Problem 3.11.2 ($(\mathsf{tw} \leq k)$-Emptiness)**.**
Given a VTRATG G such that $\mathsf{tw}(D(G)) \leq k$, is $L(G) = \emptyset$?

Theorem 3.11.3. $(\mathsf{tw} \leq k)$*-Emptiness* $\in \mathsf{P}$ *for all* k.

Proof. Compute the sets A_v of partial assignments as in the proof of Theorem 3.11.2. We have $t \in L(G)$ iff there exists a $(f, p) \in A_v$ with $f(A) = t$. Hence $L(G) \neq \emptyset$ iff there exists a $(f, p) \in A_v$ such that $f(A)$ is defined. □

3.11.3 Containment

Having determined the complexity of membership, we can now turn towards containment. In contrast to the unconstrained case where containment was Π_2^P-complete (Theorem 3.4.1), for bounded treewidth it will turn out to be only coNP-complete:

Problem 3.11.3 ($(\mathsf{tw} \leq k)$-Containment)**.**
Given VTRATGs G_1, G_2 such that $\mathsf{tw}(D(G_1)) \leq k$ and $\mathsf{tw}(D(G_2)) \leq k$, is $L(G_1) \subseteq L(G_2)$?

Theorem 3.11.4. $(\mathsf{tw} \leq k)$*-Containment is* coNP*-complete for* $k \geq 1$.

Proof. The problem is in coNP since for any t (whose symbolic size we can bound by a polynomial in the symbolic size of G_1), we can check in polynomial time whether $t \notin L(G_1)$ or $t \in L(G_2)$. Hardness follows via reduction from the complement of 3SAT. Hence let φ be a formula in 3CNF with the variables $\overline{x}$. We choose $G_1 = \mathrm{Inst}_{\varphi,\overline{x}}$, which generates exactly the substitution instances of φ where we replace the variables by either true or false. We clearly have $\mathsf{tw}(D(G_1)) = 1$.

For the reduction we want to find a TRATG G_2 such that $L(G_1) \subseteq L(G_2)$ iff φ is unsatisfiable. Ideally $L(G_2)$ would contain exactly the false substitution instances of φ, however this grammar would likely have a dependency graph of unbounded treewidth since we need to choose the same substitution for every clause. Therefore we use a slightly different grammar G_2 that generates false formulas of the same "shape" as φ, in particular it generates all false instances of φ:

$$
\begin{aligned}
A &\to \mathsf{and}(C_1, \ldots, C_{i-1}, F, C_{i+1}, \ldots, C_n) \quad \text{(for each } i \leq n\text{)} \\
C_i &\to \mathtt{or}(D_{i,1}, D_{i,2}, D_{i,3}) \\
D_{i,j} &\to \mathtt{false} \mid \mathtt{neg}(\mathtt{true}) \mid \mathtt{true} \mid \mathtt{neg}(\mathtt{false}) \quad \text{(for each } i \leq n \text{ and } j \leq 3\text{)} \\
F &\to \mathtt{or}(G_1, G_2, G_3) \\
G_j &\to \mathtt{false} \mid \mathtt{neg}(\mathtt{true}) \quad \text{(for each } j \leq 3\text{)}
\end{aligned}
$$

Each nonterminal C_i expands to a substitution instance of a clause. The nonterminal F expands to such instances as well, but only to those which are semantically false. We clearly have $\mathrm{tw}(D(G_2)) = 1$. Since F expands to a false clause, every formula in $L(G_2)$ is equivalent to false. So if $L(G_1) \subseteq L(G_2)$, then φ is unsatisfiable. On the other hand, $L(G_2)$ contains every false substitution instance of φ: if the instance is false, then at least one conjunct of the top-level and is false and we can construct a derivation in G_2 using the corresponding production for A. So we have a reduction from the complement of 3SAT; it is polynomial-time since the grammars G_1, G_2 can be computed in polynomial time from φ. □

3.11.4 Disjointness

Interestingly, bounding the treewidth of the dependency graph does not change the complexity of the disjointness problem. In both the unconstrained and the treewidth-bounded case, the problem is coNP-complete. However we need to use a different proof for coNP-hardness: in the unconstrained case, we could reduce the complement of the NP-complete membership problem to disjointness. But this approach does not work here, since treewidth-bounded membership is in P. Therefore we use a similar proof as for $(\mathrm{tw} \leq k)$-Containment in Theorem 3.11.4.

Problem 3.11.4 ($(\mathsf{tw} \leq k)$-Disjointness)**.**
Given VTRATGs G_1, G_2 such that $\mathsf{tw}(D(G_1)) \leq k$ and $\mathsf{tw}(D(G_2)) \leq k$, is $L(G_1) \cap L(G_2) = \emptyset$?

Theorem 3.11.5. *$(\mathsf{tw} \leq k)$-Disjointness is* coNP*-complete for $k \geq 1$.*

Proof. The problem is in coNP since for every derivation of a term $t \in L(G_1)$ we can check whether $t \notin L(G_2)$ in polynomial time by Theorem 3.11.2. For coNP-hardness, we show a reduction from the complement of 3SAT as in the proof of Theorem 3.11.4. So let φ be a formula in 3CNF with the variables $\overline{x}$. We set $G_1 = \mathrm{Inst}_{\varphi,\overline{x}}$. For G_2 we pick a TRATG that generates only true formulas, including all true substitution instances of φ:

$$
\begin{aligned}
A &\to \mathsf{and}(B_1, \ldots, B_n) \\
B_i &\to \mathtt{or}(C_i, D_{i,1}, D_{i,2}) \mid \mathtt{or}(D_{i,1}, C_i, D_{i,2}) \mid \mathtt{or}(D_{i,1}, D_{i,2}, C_i) \\
C_i &\to \mathtt{true} \mid \mathtt{neg}(\mathtt{false}) \\
D_{i,j} &\to \mathtt{true} \mid \mathtt{false} \mid \mathtt{neg}(\mathtt{false}) \mid \mathtt{neg}(\mathtt{false})
\end{aligned}
$$

The nonterminal B_i expands to an instance of a clause where at least one literal evaluates to true, C_i expands to a true instance of a literal, and $D_{i,j}$ expands to an arbitrary instance of a literal. As usual, we have $\mathsf{tw}(D(G_2)) = 1$ and G_2 is also polynomial-time computable. As in Theorem 3.11.4, $L(G_1) \cap L(G_2) = \emptyset$ iff φ is unsatisfiable, i.e., there is a substitution instance of φ that evaluates to true. □

3.11.5 Equivalence

The complexity of the equivalence problem follows directly from containment using the same reduction as for the unconstrained case in Theorem 3.6.1 using the union operation on VTRATGs.

Problem 3.11.5 ($(\mathsf{tw} \leq k)$-Equivalence)**.**
Given VTRATGs G_1, G_2 such that $\mathsf{tw}(D(G_1)) \leq k$ and $\mathsf{tw}(D(G_2)) \leq k$, is $L(G_1) = L(G_2)$?

Theorem 3.11.6. *$(\mathsf{tw} \leq k)$-Equivalence is* coNP*-complete for $k \geq 1$.*

Proof. We first show that the problem is in coNP via a reduction to $(\mathrm{tw} \leq k)$-Containment: $L(G_1) = L(G_2) \leftrightarrow L(G_1) \subseteq L(G_2) \wedge L(G_2) \subseteq L(G_1)$—and coNP is closed under intersection. Hardness follows by a reduction from $(\mathrm{tw} \leq k)$-Containment: $L(G_1) \subseteq L(G_2)$ iff $L(G_1) \cup L(G_2) = L(G_2)$. This is equivalent to $L(G_1 \cup G_2) = L(G_2)$, an instance of $(\mathrm{tw} \leq k)$-Equivalence. □

3.11.6 Minimization

The minimization problem has the same complexity as in the unconstrained case: it is NP-complete.

Problem 3.11.6 ($(\mathrm{tw} \leq k)$-Minimization)**.**
Given a VTRATG $G = (N, \Sigma, P, \mathrm{A})$ with $\mathrm{tw}(D) \leq k$, a set of terms L such that $L(G) \supseteq L$, and $n \geq 0$, is there a subset $P' \subseteq P$ of the productions such that $|P'| \leq n$ and $L(G') \supseteq L$ where $G' = (N, \Sigma, P', \mathrm{A})$?

Note that such a grammar G' also has the property that $\mathrm{tw}(D(G')) \leq k$.

Theorem 3.11.7. $(\mathrm{tw} \leq k)$*-Minimization is* NP*-complete for* $k \geq 1$.

Proof. The problem is in NP because it is a special case of the VTRATG-Minimization problem. Hardness follows by reduction from Set Cover as in Theorem 3.8.1: the grammar used in the proof of Theorem 3.8.1 has the property that $\mathrm{tw}(D(G)) = 1$. □

3.11.7 Cover

The complexity of the Cover and n-Cover problems is also unchanged from the unconstrained case.

Problem 3.11.7 ($(\mathrm{tw} \leq k)$-Cover)**.**
Given a finite set of terms L and $n \geq 0$, is there a VTRATG G such that $|G| \leq n$, $\mathrm{tw}(D(G)) \leq k$, and $L(G) \supseteq L$?

In contrast to Corollary 3.7.1 for VTRATG-Cover, there is no obvious trivial solution to $(\mathrm{tw} \leq k)$-Cover. The optimally compressing grammar given by Theorem 3.7.1 has a dependency graph of unbounded treewidth.

Theorem 3.11.8. $(\mathsf{tw} \leq k)$-*Cover* $\in$ NP.

Proof. By Lemma 3.11.1, the maximum length of a nonterminal vector is bounded by $k+1$. A covering VTRATG of size $\leq n$ then has at most n productions of length $k+1$, and we can guess such a VTRATG in polynomial time and check whether $\mathsf{tw}(D(G)) \leq k$. Checking $L(G) \supseteq L$ is also in NP by Theorem 3.2.1. □

Once again, it is open whether the cover problem is NP-complete:

Open Problem 3.11.1. Is $(\mathsf{tw} \leq k)$-Cover NP-complete?

And the variant where we bound the number of nonterminals is NP-complete by reduction from Regular-n-Cover and verifying that the TRATGs have a dependency graph of bounded treewidth:

Problem 3.11.8 ($(\mathsf{tw} \leq k)$-n-Cover)**.**
Given a finite set of terms L and $l \geq 0$, is there a VTRATG $G = (N, \Sigma, P, A)$ such that $|G| \leq l$, $\mathsf{tw}(D(G)) \leq k$, $|N| \leq n$, and $L(G) \supseteq L$?

Theorem 3.11.9. $(\mathsf{tw} \leq k)$-n-*Cover is* NP*-complete for* $n \geq 2$ *and* $k \geq 1$.

Proof. The problem is in NP analogously to Theorem 3.11.8: checking the extra condition $|N| \leq n$ can be done in polynomial time. For hardness we need to analyze the TRATGs used in the proofs of Lemmas 3.7.3 to 3.7.5 and verify that the treewidth of their dependency graph is at most one. In Lemmas 3.7.3 and 3.7.4 we use TRATGs G with at most 2 nonterminals, hence $\mathsf{tw}(D(G)) = 1$. In Lemma 3.7.5, we transform a TRATG by adding a new start symbol A at the beginning; we add two kinds of productions: productions which do not contain a nonterminal on the right-hand side, these do not contribute to the dependency graph. And productions from the new start symbol A, which only contain a single nonterminal on the right-hand side, namely B. Hence the dependency graph is only extended by a single new vertex A and an edge $A - B$, preserving the treewidth (which is one). □

3.12 Decision problems on induction grammars

3.12.1 Membership

We have motivated the class of treewidth-bounded VTRATGs in Section 3.11 by instance grammars, which are a natural class of VTRATGs whose dependency graph has bounded treewidth. There is however a small technical issue, which we will now explain. Let us first recall Example 3.11.1:

Example 3.12.1. Consider the following induction grammar G:

$$\begin{aligned} \tau &\to r(\gamma) \\ \gamma &\to c \mid f(\gamma) \end{aligned}$$

Then the instance grammar $I(G, s^n(0))$ has the following form:

$$\begin{aligned} \tau &\to r(\gamma_0) \mid \cdots \mid r(\gamma_n) \\ \gamma_0 &\to c \mid f(\gamma_1) \\ \gamma_1 &\to c \mid f(\gamma_2) \\ &\vdots \\ \gamma_{n-1} &\to c \mid f(\gamma_n) \\ \gamma_n &\to c \end{aligned}$$

The treewidth of the dependency graph is $\mathrm{tw}(D(I(G, s^n(0)))) \leq 2$, as witnessed by the following tree decomposition. That is, the treewidth is uniformly bounded for all instance terms.

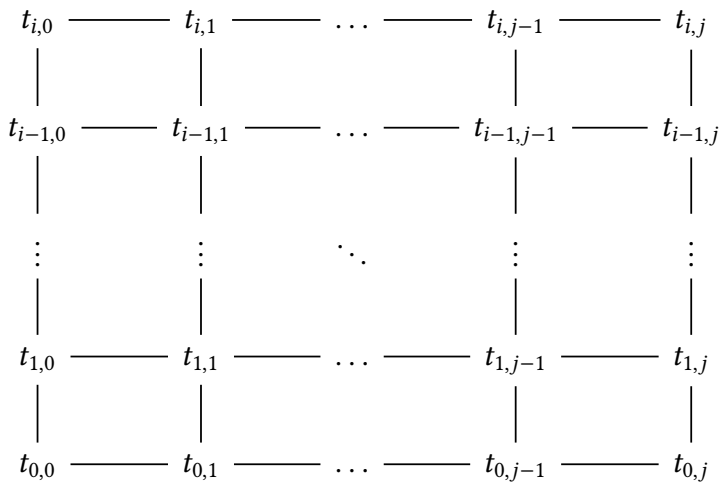
$$\{\tau, \gamma_0, \gamma_1\} - \{\tau, \gamma_1, \gamma_2\} - \cdots - \{\tau, \gamma_{n-1}, \gamma_n\}$$

However, a small issue arises if the inductive datatype has a constructor with two recursive occurrences since the instance grammars have nonterminal vectors $\overline{\gamma_s}$ for every subterm s, even if it occurs multiple times:

Example 3.12.2. Consider the inductive type of binary trees with the constructors l^τ and $n^{\tau\to\tau\to\tau}$ and the following induction grammar:

$$\begin{aligned} \tau &\to r(\gamma) \\ \gamma &\to c \mid f(\gamma) \end{aligned}$$

Define the (free constructor) terms $t_{i+1,j+1} = n(t_{i,j+1}, t_{i+1,j})$, $t_{i+1,0} = t_{0,i+1} = n(t_{i,0}, t_{i,0})$, $t_{0,0} = l$. Then the subterm structure of $t_{i,j}$ contains a grid, where two subterms are connected by an edge if they are direct subterms (there are more edges than shown here, since $t_{i,j} = t_{j,i}$ is symmetric):



The treewidth of such a $(i \times j)$-grid is exactly $\min(i, j)$, and this grid is a subgraph of $D(I(G, t_{i,j}))$. Thus $\mathrm{tw}(D(I(G, t_{i,i}))) \geq i$ is not uniformly bounded.

This minor technical issue is easily solved by using a slightly different definition for the instance grammar, with nonterminals $\overline{\gamma_p}$ for every position p (instead of $\overline{\gamma_s}$ for every subterm):

Definition 3.12.1. Let $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ be an induction grammar, and r a constructor term of the same type as α. The *modified instance grammar* $I'(G, r) = (\tau, N, P')$ is a VTRATG with nonterminal vectors $N = \{\tau\} \cup \{\overline{\gamma_p} \mid p \in \mathrm{Pos}(r)\}$ and productions $P' = \{p' \mid \exists p \in P\ (p \rightsquigarrow p')\}$. The instantiation relation $p \rightsquigarrow' p'$ is defined as follows:

- $\tau \to t[\alpha, \overline{\nu_i}, \overline{\gamma}] \ \rightsquigarrow' \ \tau \to t[r, \overline{s}, \overline{\gamma_p}]$ for $p \in \mathrm{Pos}(r)$ with $r|_p = c_i(\overline{s})$
- $\overline{\gamma} \to \overline{t}[\alpha] \ \rightsquigarrow \ \overline{\gamma_p} \to \overline{t}[r]$ for $p \in \mathrm{Pos}(r)$
- $\overline{\gamma} \to \overline{t}[\alpha, \overline{\nu_i}, \overline{\gamma}] \ \rightsquigarrow \ \overline{\gamma_{pj}} \to \overline{t}[r, \overline{s}, \overline{\gamma_p}]$ for $p \in \mathrm{Pos}(r)$ such that $r|_p = c_i(\overline{s})$, where j is a recursive occurrence in c_i

How do the VTRATGs $I(G, r)$ and $I(G, r)$ differ? They only differ (ignoring the names of the nonterminals) if r contains a subterm that occurs at two different positions. In general, r regarded as a tree can be exponentially larger than r regarded as a DAG. The VTRATG $I'(G, r)$ treats r as a tree since it is defined in terms of the positions of r, while $I(G, r)$ treats r as a DAG since it is defined in terms of the subterms of r. For example, the term $t_{i,0} = n(t_{i-1,0}, t_{i-1,0}) = \dots$ in Example 3.12.2 contains $i+1$ subterms, but $2^{i+1} - 1$ positions.

The VTRATG $I'(G, r)$ is polynomial-time computable from a tree representation of r, and $I(G, r)$ is polynomial-time computable from a DAG representation of r. There is a size difference between $I(G, r)$ and $I'(G, r)$: the tree representation of r can be exponentially larger than the DAG representation. In this case, $I'(G, r)$ will also be exponentially larger than $I(G, r)$.

Lemma 3.12.1. $L(G, i) = L(I(G, i)) = L(I'(G, i))$.

Proof. We can transform derivations between the two definitions of instance grammars. If we have a derivation in $I(G, i)$ then we can pick the corresponding productions in $I'(G, i)$ to derive the same term (every nonterminal vector in $I(G, i)$ corresponds to at most one nonterminal vector in $I'(G, i)$, so it is clear which production to pick.)

On the other hand if we have a $\overset{1}{\Rightarrow}$-derivation in $I'(G, i)$, then it only contains nonterminal vectors $\overline{\gamma_p}$ for positions p along a single branch of the term i. (We can see this by looking at the productions.) That is, there is a position p_0 such that $p \subseteq p_0$ for every nonterminal vector $\overline{\gamma_p}$ occurring in the derivation. Every nonterminal vector in $I(G, i)$ corresponds to at most one such nonterminal vector $\overline{\gamma_p}$ (since we do not have parallel occurrences). And we get a derivation of the same term in $I(G, i)$ by picking the corresponding productions. □

Lemma 3.12.2. *Let* $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ *be an induction grammar. Then* $\mathrm{tw}(D(I'(G, t))) \leq 2|\overline{\gamma}|$.

Proof. We construct a tree decomposition with the vertices $V_T = \{\epsilon\} \cup \{p \in \mathrm{Pos}(t) \mid p \text{ is a recursive occurrence}\}$. Two vertices are adjacent if they are positions of direct subterms, i.e. $E_T = \{\{p, pi\} \mid pi \in V_T\}$. For every $pi \in V_T$

we define $X_{pi} = \{\tau\} \cup \overline{\gamma_p} \cup \overline{\gamma_{pi}}$, and $X_\epsilon = \{\tau\} \cup \overline{\gamma_\epsilon}$. This tree decomposition has the desired width. □

Lemma 3.12.2 allows us to reduce the membership problem for induction grammars the one for VTRATGs with dependency graph of bounded treewidth.

Problem 3.12.1 ($(|\overline{\gamma}| \leq k)$-Ind-Membership)**.**
Given an induction grammar $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ such that $|\overline{\gamma}| \leq k$, a free constructor term i, and a term t represented as a tree, is $t \in L(G, i)$?

Theorem 3.12.1. *$(|\overline{\gamma}| \leq k)$-Ind-Membership $\in$ P for all k.*

Proof. By Lemma 3.12.1, $t \in L(G, i)$ iff $t \in L(I'(G, i))$. This is an instance of $(\mathrm{tw} \leq 2|\overline{\gamma}|)$-Membership by Lemma 3.12.2, which is in P by Theorem 3.11.2. □

The restriction of $|\overline{\gamma}| \leq k$ is necessary in Theorem 3.12.1 (assuming P $\neq$ NP). If the size of $|\overline{\gamma}|$ is not bounded, then the membership problem becomes NP-complete:

Problem 3.12.2 (Ind-Membership)**.**
Given an induction grammar $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$, a free constructor term i represented as a tree, and a term t, is $t \in L(G, i)$?

This is because we can encode any given VTRATG G as an induction grammar by making $\overline{\gamma}$ as wide as necessary. The general idea is that we use the instance grammar with n nonterminal vectors to simulate a VTRATG with n nonterminal vectors. This induction grammar will generate $L(G)$ in the sense that $L(\mathrm{Embed}_G, n) \cap \mathcal{T}(\Sigma) = L(G)$ as we will see in Lemma 3.12.3. We need to restrict the language of the induction grammar to the original signature since the induction grammar also generates junk terms, which correspond to "derivations" (in G) that do not substitute all the nonterminals. The reason for these junk terms is that we want every nonterminal vector in the instance grammars to have at least one production, yielding a stronger result.

All of the induction grammars that we construct for the hardness and undecidability proofs will be for the natural numbers. This is no significant restriction, we could perform the same construction for any other datatype that has

free constructor terms of unbounded size (or equivalently, if it has infinitely many up to renaming). In the case of the natural numbers, $L(\text{Embed}_P, n)$ produces the language of the VTRATG. In the general setting, we would get that $L(\text{Embed}_P, t)$ produces that language for any free constructor term t of term depth $n - 1$.

Definition 3.12.2. Let $G = (N, \Sigma, P, A)$ be a VTRATG with and $N = \{A \prec \overline{B_1} \prec \cdots \prec \overline{B_n}\}$. Let r_β be a new constant for every type β, and for every $\overline{B_i}$, define $\overline{r_i} = (r_{\beta_1}, \ldots, r_{\beta_{k_i}})$ with the same types as $\overline{B_i}$. We define the induction grammar $\text{Embed}_G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P')$ with $\overline{\gamma} = (\overline{B_1}, \ldots, \overline{B_n})$ and the following productions:

$$\tau \longrightarrow s \quad \text{for every production } A \longrightarrow s \in P$$
$$\overline{\gamma} \longrightarrow (\overline{r_1}, \ldots, \overline{r_{i-1}}, \overline{s}, \overline{B_{i+1}}, \ldots, \overline{B_n}) \quad \text{for every production } \overline{B_i} \longrightarrow \overline{s} \in P$$

Lemma 3.12.3. *Let $G = (N, \Sigma, P, A)$ be a VTRATG. Then $L(\text{Embed}_G, n) \cap \mathcal{T}(\Sigma) = L(G)$ for all $n \geq |N| - 1$.*

Proof. If δ is a derivation in G, then the derivation δ' with the corresponding productions in $I(G, t)$ derives the same term. On the other hand if δ is a derivation of a term $q \in \mathcal{T}(\Sigma)$, then q does not contain r_β for any β and the derivation δ' using the corresponding productions in G derives the same term q. □

Theorem 3.12.2. *Ind-Membership is* NP*-complete.*

Proof. By a reduction from VTRATG-Membership, which is NP-complete by Theorem 3.2.1, using Lemma 3.12.3. □

3.12.2 Emptiness

The complexity of the emptiness problem on induction grammars is surprising: in this section, we will show that it is PSPACE-complete. In a sense it stands between the (relatively tractable) NP-complete membership problem and the undecidable containment, equivalence, and disjointness problems that we will consider in the following sections.

Problem 3.12.3 (Ind-Emptiness)**.**
Given an induction grammar G, is $L(G, i) = \emptyset$ for all free constructor terms i?

Problem 3.12.4 ($(|\overline{\gamma}| \leq k)$-Ind-Emptiness)**.**
Given an induction grammar $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ such that $|\overline{\gamma}| \leq k$, is $L(G, i) = \emptyset$ for all free constructor terms i?

These problems are deeply connected to another model in formal language theory, namely non-deterministic finite automata. Intuitively, the hardness of emptiness in induction grammars is due to a large size of $\overline{\gamma} = (\gamma_1, \ldots, \gamma_n)$. If we only care about the set of occurring nonterminals, applying a production $\overline{\gamma} \to \overline{t}$ (to be precise, an instance of the production in the instance grammar) then has the effect of replacing one set of nonterminals $\{\gamma_3, \gamma_5, \gamma_8\}$ occurring by an other set $\{\gamma_7, \gamma_9\}$. The language of the induction grammar is empty iff we can never reach the empty set. On the level of the automaton, $\gamma_1, \ldots, \gamma_n$ will be the states, the sequence of productions in a derivation will be a word, and reaching the empty set means that a word is not accepted by the automaton (since the empty set obviously contains no final states).

Definition 3.12.3. A non-deterministic finite automaton (NFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states.

We extend the transition relation δ recursively to a relation $\delta \subseteq Q \times \Sigma^* \times Q$ by setting $(q, \epsilon, q') \in \delta$ iff $q = q'$, and $(q, ab, q') \in \delta$ iff $\exists q''\ (q, a, q'') \in \delta \wedge (q'', b, q') \in \delta$. The language $L(A)$ accepted by A is defined as $L(A) = \{w \in \Sigma^* \mid \exists q \in F\ (q_0, w, q) \in \delta\}$.

In our construction, the language of an induction grammar will be empty iff the automation accepts all words. Interestingly, this universality problem for NFAs is PSPACE-complete:

Problem 3.12.5 (NFA-Universality)**.**
Given an NFA $A = (Q, \Sigma, \delta, q_0, F)$, is $L(A) = \Sigma^*$?

Meyer and Stockmeyer first considered the complexity of NFA-Universality in [70], where they also introduce the polynomial-time hierarchy. While they

do not explicitly state that the problem is PSPACE-complete, they show that it is Π_k^P-hard for every k, and exhibit a polynomial-time reduction from the membership problem for context-sensitive grammars, which is PSPACE-hard.

Theorem 3.12.3 ([70], Lemma 2.3)**.** *NFA-Universality is* PSPACE*-complete.*

Our strategy is now as follows: we will assign to every induction grammar an NFA, such that the NFA is (almost) universal iff the language of the induction grammar is empty. Given an induction grammar $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ we define the sets of productions $P_\tau = \{p \in P \mid \exists t\ p = \tau \to t\}$ and $P_{\overline{\gamma}} = \{p \in P \mid \exists \overline{t}\ p = \overline{\gamma} \to \overline{t}\}$. To simplify the construction of the automaton, we do not aim for universality but for $L(A) = P_\tau P_{\overline{\gamma}}^*$. This is not a significant problem, since the complement of $P_\tau P_{\overline{\gamma}}^*$ is recognized by a small DFA so it will be easy to reduce $L(A) = P_\tau P_{\overline{\gamma}}^*$ to $L(A') = \Sigma^*$ for some only polynomially larger automaton A'.

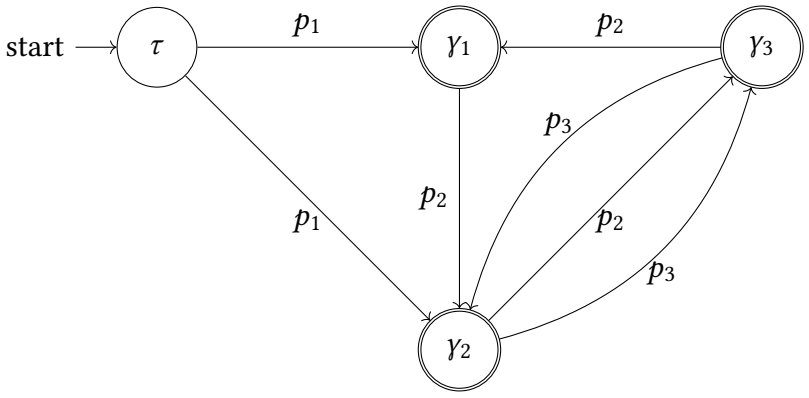
Definition 3.12.4. Let $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ be an induction grammar such that $\overline{\gamma} = (\gamma_1, \ldots, \gamma_n)$. We define the NFA $A_{\text{empty}}(G) = (Q, \Sigma, \delta, q_0, F)$ as follows:

- $Q = \{\tau, \gamma_1, \ldots, \gamma_n\}$
- $\Sigma = P$
- $q_0 = \tau$
- $F = Q \setminus \{\tau\}$
- $(q, p, q') \in \delta$ iff one of the following cases applies:
  - $q = \tau$, $p = \tau \to t$, and q' occurs in t
  - $q = \gamma_i$, $p = \overline{\gamma} \to \overline{t}$, and q' occurs in t_i.

Example 3.12.3. Consider the induction grammar G with the following productions:

$$\tau \to r(\gamma_1, \gamma_2) \tag{p_1}$$
$$(\gamma_1, \gamma_2, \gamma_3) \to (\gamma_2, \gamma_3, \gamma_1) \tag{p_2}$$
$$(\gamma_1, \gamma_2, \gamma_3) \to (c, \gamma_3, \gamma_2) \tag{p_3}$$

Then the corresponding NFA $A_{\text{empty}}(G)$ has the following states and transitions. As we can clearly see, the word $p_1p_3p_2p_3$ is not accepted and the language of the induction grammar is hence nonempty.



Lemma 3.12.4. *Let G be an induction grammar. Then $L(A_{\text{empty}}(G)) = P_\tau P_{\overline{\gamma}}^*$ iff $\forall i\ L(G, i) = \emptyset$.*

Proof. It is clear that $L(A_{\text{empty}}(G)) \subseteq P_\tau P_{\overline{\gamma}}^*$ by Definition 3.12.4. A word accepted by $A_{\text{empty}}(G)$ is a sequence of productions such that these productions cannot be a derivation in any instance grammar: there always remains a nonterminal at the end in the derivation in the instance grammar (namely the final state).

On the other hand, if $p_0 p_1 \dots p_n \in P_\tau P_{\overline{\gamma}}^*$ is a sequence of productions not accepted by $A_{\text{empty}}(G)$, then we can build a derivation in an instance grammar using these productions for a free constructor term t with depth larger than n. The instances of the productions $p_0, \dots, p_n$ then form a derivation, since the resulting term does not contain any nonterminals—if it contained a nonterminal, then $A_{\text{empty}}(G)$ would have accepted. □

Lemma 3.12.5. *Ind-Emptiness $\leq_P$ NFA-Universality*

Proof. We already know that $L(A_{\text{empty}}(G)) \subseteq P_\tau P_{\overline{\gamma}}^*$, hence $L(A_{\text{empty}}(G)) = P_\tau P_{\overline{\gamma}}^*$ iff $L(A_{\text{empty}}(G)) \cup (\Sigma^* \setminus P_\tau P_{\overline{\gamma}}^*) = \Sigma^*$. By Lemma 3.12.4, it only remains to show that $L(A_{\text{empty}}(G)) \cup (\Sigma^* \setminus P_\tau P_{\overline{\gamma}}^*)$ can be recognized by a (polynomial-time) computable NFA. Such an NFA can be easily constructed as a product automaton of $A_{\text{empty}}(G)$ and an obvious DFA recognizing $\Sigma^* \setminus P_\tau P_{\overline{\gamma}}^*$. □

Lemma 3.12.6. *NFA-Universality $\leq_P$ Ind-Emptiness*

Proof. We need to do a bit of work in this reduction as well. Note that the map $G \mapsto A_{\text{empty}}(G)$ is not surjective: first, there are no transitions back to

the start symbol, and second (and more importantly), all states in $A_{\text{empty}}(G)$ are final. At least implicitly, we will have to take care of these two issues.

So we are given an NFA $A = (Q, \Sigma, \delta, q_0, F)$, and we will construct an induction grammar G. If there is a non-accepting word $w \notin L(A)$, then we want a suitable instance language $L(G, i) \neq \emptyset$ to be nonempty. Let $Q = (q_0, \ldots, q_n)$, we set $\overline{\gamma} = (\gamma_0, \ldots, \gamma_n, \gamma_{n+1})$. Define the transition function $\delta(q, a) = \{q' \mid (q, a, q') \in \delta\}$. Assign to every set of states a term $F(\{q_{i_1}, \ldots, q_{i_n}\}) = f(\gamma_{i_1}, f(\gamma_{i_2}, \ldots f(\gamma_{i_n}, c) \ldots))$ containing exactly the γ-nonterminals corresponding to the states. Then G contains the following productions:

$$\tau \to r(\gamma_0)$$
$$\overline{\gamma} \to (t_{a,0}, \ldots, t_{a,n}, \gamma_{n+1}) \quad \text{for each } a \in \Sigma, \text{ where } t_{a,i} = F(\delta(q_i, a))$$
$$\overline{\gamma} \to (s_0, \ldots, s_i, \gamma_{n+1}) \quad \text{where } s_i = c \text{ if } q_i \notin F, \text{ and } s_i = \gamma_{n+1} \text{ otherwise}$$

Call the productions p_0, p_a, and p_F, respectively. Let $w = a_1 \ldots a_n \notin L(A)$ be a non-accepting word, that is, $\delta(q_0, w) \notin F$. Then $p_0 p_{a_1} \ldots p_{a_n} p_F$ is a derivation in $I(G, t)$ for some large enough t. On the other hand let $L(G, i) \neq \emptyset$ witnessed by a derivation δ with the productions $p_0 p_1 \ldots p_n$. If $p_i = p_F$ for $i < n$, then we can drop the remaining productions. So δ uses the productions $p_0 p_{a_1} \ldots p_{a_n}$ or $p_0 p_{a_1} \ldots p_{a_n} p_F$. Now $a_1 \ldots a_n$ is a non-accepting word. □

Theorem 3.12.4. *Ind-Emptiness is* PSPACE*-complete.*

Proof. By Lemmas 3.12.5 and 3.12.6, there is a polynomial reduction to and from NFA-Universality, which is PSPACE-complete by Theorem 3.12.3. □

If $|\overline{\gamma}|$ is bounded by a fixed constant, then $A_{\text{empty}}(G)$ has only a bounded number of states, thus making the problem tractable:

Theorem 3.12.5. $(|\overline{\gamma}| \leq k)$*-Ind-Emptiness* $\in$ P

Proof. By Lemma 3.12.4, emptiness is equivalent to $L(A_{\text{empty}}(G)) = P_\tau P_{\overline{\sigma}}^*$. Since the number of states in $A_{\text{empty}}(G)$ is bounded by $k + 1$, we can compute an equivalent DFA in constant time, and then verify $L(A_{\text{empty}}(G)) = P_\tau P_{\overline{\sigma}}^*$ in constant time. □

3.12.3 Containment

We will show undecidability of containment, disjointness, and equivalence of induction grammars by exhibiting a reduction from the undecidable Post correspondence problem [81]:

Problem 3.12.6 (Post-Correspondence)**.**
Given two finite lists of words $w_1, \ldots, w_n$ and $v_1, \ldots, v_n$, is there a sequence of indices $i_1, \ldots, i_k$ for some $k \geq 1$ such that $w_{i_1} \ldots w_{i_k} = v_{i_1} \ldots v_{i_k}$?

Such a sequence $i_1, \ldots, i_k$ is called a solution.

Problem 3.12.7 (Ind-Containment)**.**
Given induction grammars G_1, G_2 for the same datatype, is $L(G_1, t) \subseteq L(G_2, t)$ for all free constructor terms t?

To show that containment is undecidable, we construct two induction grammars: the first one generates all pairs $(w_{i_1} \ldots w_{i_k}, v_{i_1} \ldots v_{i_k})$. The second one generates all pairs (u, v) where $u \neq v$ are different words. As usual, we encode a word abc as the term $a(b(c(\epsilon)))$. Given a word $a_1 a_2 \cdots a_n$ (with $a_j \in \Sigma$ for all j) and a term t, we define $(a_1 a_2 \cdots a_n) \cdot t = a_1(a_2(\cdots a_n(t) \cdots))$.

Definition 3.12.5. Let $P = ((w_1, \ldots, w_n), (v_1, \ldots, v_n))$ be an instance of Post-Correspondence. The induction grammar Image_P has the following productions:

$$\begin{aligned} \tau &\to r(\gamma_1, \gamma_2) \\ (\gamma_1, \gamma_2) &\to (w_1 \cdot \gamma_1, v_1 \cdot \gamma_2) \mid \cdots \mid (w_n \cdot \gamma_1, v_n \cdot \gamma_2) \\ (\gamma_1, \gamma_2) &\to (w_1, v_1) \mid \cdots \mid (w_n, v_n) \end{aligned}$$

Lemma 3.12.7. $L(\mathrm{Image}_P, k) = \{r(w_{i_1} \cdots w_{i_l}, v_{i_1} \cdots v_{i_l}) \mid 1 \leq l \leq k+1\}$

Proof. Obvious from the construction. □

The second induction grammar that we use to show the undecidability of containment produces all different words. The idea is that (γ_3, γ_4) expands to (possibly equal) words v, w, while (γ_1, γ_2) expands to *different* words.

Definition 3.12.6. Let Σ be a set of letters and $l \geq 1$. Then the induction grammar $\mathrm{Diff}_{\Sigma,l}$ has the following productions (where t, u, v, w range over all words in Σ^*, and $\overline{\gamma} = (\gamma_1, \gamma_2, \gamma_3, \gamma_4)$):

$$\tau \to r(\gamma_1, \gamma_2) \tag{3.1}$$

$$\overline{\gamma} \to (t \cdot \gamma_1, u \cdot \gamma_2, v \cdot \gamma_3, w \cdot \gamma_4) \quad (|t| = |u| = l, \text{ and } \max(|v|, |w|) \leq l) \tag{3.2}$$

$$\overline{\gamma} \to (t \cdot \gamma_3, u \cdot \gamma_4, v \cdot \gamma_3, w \cdot \gamma_4) \quad (|t| = |u| \leq l,\ t \neq u, \text{ and } \max(|v|, |w|) \leq l) \tag{3.3}$$

$$\overline{\gamma} \to (t, u, v, w) \quad (\max(|t|, |u|, |v|, |w|) \leq l \text{ and } t \neq u) \tag{3.4}$$

Lemma 3.12.8. $L(\mathrm{Diff}_{\Sigma,l}, k) = \{r(v, w) \mid v, w \in \Sigma^* \wedge v \neq w \wedge \max(|v|, |w|) \leq l(k+1)\}$

Proof. The instance grammar $I(\mathrm{Diff}_{\Sigma,l}, k)$ has the following $k+1$ nonterminal vectors: $\tau, (\gamma_{1,1}, \gamma_{1,2}, \gamma_{1,3}, \gamma_{1,4}), \ldots, (\gamma_{k,1}, \gamma_{k,2}, \gamma_{k,3}, \gamma_{k,3})$. The pair $(\gamma_{i,3}, \gamma_{i,4})$ expands to exactly all pairs of words (v, w) such that $\max(|v|, |w|) \leq (k-i+1)l$. The production (3.4) starts with $\leq l$ letters at the end, and the other two productions (3.2) and (3.3) prepend $\leq l$ many letters as well. Now we can see that the pair $(\gamma_{i,1}, \gamma_{i,2})$ expands to exactly all pairs of words (v, w) such that $v \neq w$ and $\max(|v|, |w|) \leq (k-i+1)l$. The production (3.3) clearly generates all different words of the desired size. For larger words, there are two possibilities: either the words differ within the first l letters, then they are generated by production (3.3). Or their first l letters are the same, then they are generated by production (3.2). □

The induction grammar $\mathrm{Diff}_{\Sigma,l}$ is not polynomial-time computable. Its size is exponential in l (if $|\Sigma| \geq 2$). However this poses no problem since we only use it in a reduction to show undecidability, where there are no constraints on the runtime.

Lemma 3.12.9. *Let $P = ((w_1, \ldots, w_n), (v_1, \ldots, v_n))$ be an instance of Post-Correspondence, let Σ be a finite alphabet such that $w_i, v_i \in \Sigma^*$ for all i, and let $l \geq 1$ be such that $|w_i|, |v_i| \leq l$ for all i. Then P has a solution iff there exists a k such that $L(\mathrm{Image}_P, k) \not\subseteq L(\mathrm{Diff}_{\Sigma,l}, k)$.*

Proof. The words $w_{i_1} \cdots w_{i_m}$ and $v_{i_1} \cdots v_{i_m}$ necessarily have both have length $|w_{i_1} \cdots w_{i_m}|, |v_{i_1} \cdots v_{i_m}| \leq lm \leq l(k+1)$ due to the choice of l. So by Lemmas 3.12.7 and 3.12.8, $L(\text{Image}_P, k) \not\subseteq L(\text{Diff}_{\Sigma,l}, k)$ if and only if there is a tuple $(i_1, \ldots, i_m)$ such that $m \leq k+1$ and $w_{i_1} \ldots w_{i_m} = v_{i_1} \ldots v_{i_m}$. Hence $L(\text{Image}_P, k) \not\subseteq L(\text{Diff}_{\Sigma,l}, k)$ iff P has a solution of length $\leq k+1$. □

Theorem 3.12.6. *Ind-Containment is undecidable.*

Proof. By Lemma 3.12.9, the undecidable Post-Correspondence problem can be reduced to Ind-Containment. □

3.12.4 Disjointness

To show that the disjointness problem is undecidable on induction grammars, we use a similar reduction from Post-Correspondence as in Lemma 3.12.9.

Problem 3.12.8 (Ind-Disjointness)**.**
Given induction grammars G_1, G_2 for the same datatype, is $L(G_1, t) \cap L(G_2, t) \neq \emptyset$ for all free constructor terms t?

However as the second grammar we now use $\text{Equal}_{\Sigma,l}$ instead of $\text{Diff}_{\Sigma,l}$, which generates all equal words:

Definition 3.12.7. Let Σ be a set of letters and $l \geq 1$. Then the induction grammar $\text{Equal}_{\Sigma,l}$ has the following productions (where t, u, v, w range over all words in Σ^*):

$$\begin{aligned} &\tau \to r(\gamma, \gamma) \\ &\gamma \to w \cdot \gamma \mid w \quad (w \in \Sigma^* \text{ and } |w| \leq l) \end{aligned}$$

Lemma 3.12.10. $L(\text{Equal}_{\Sigma,l}, k) = \{r(w, w) \mid w \in \Sigma^*, |w| \leq l(k+1)\}$

Proof. Obvious by construction. □

Lemma 3.12.11. *Let $P = ((w_1, \ldots, w_n), (v_1, \ldots, v_n))$ be an instance of Post-Correspondence, let Σ be a finite alphabet such that $w_i, v_i \in \Sigma^*$ for all i, and let $l \geq 1$ be such that $|w_i|, |v_i| \leq l$ for all i. Then P has a solution iff there exists a k such that $L(\text{Image}_P, k) \cap L(\text{Equal}_{\Sigma,l}, k) \neq \emptyset$.*

Proof. Similar to Lemma 3.12.9. Again, we know that $|w_{i_1} \cdots w_{i_m}|, |v_{i_1} \cdots v_{i_m}| \leq lm \leq l(k+1)$ due to the choice of l. So by Lemmas 3.12.7 and 3.12.10, $L(\text{Image}_P, k) \cap L(\text{Equal}_{\Sigma,l}, k) \neq \emptyset$ if and only if there is a tuple $(i_1, \ldots, i_m)$ such that $m \leq k+1$ and $w_{i_1} \ldots w_{i_m} = v_{i_1} \ldots v_{i_m}$, which by definition is the case if and only if P has a solution of length $\leq k+1$. □

Theorem 3.12.7. *Ind-Disjointness is undecidable.*

Proof. We can reduce Post-Correspondence to Ind-Disjointness using Lemma 3.12.11. □

3.12.5 Equivalence

Just as we did for VTRATGs, we can reduce equivalence to containment using a union operation on grammars.

Problem 3.12.9 (Ind-Equivalence)**.**
Given induction grammars G_1, G_2 for the same datatype, is $L(G_1, t) = L(G_2, t)$ for all free constructor terms t?

Definition 3.12.8. Let $G_1 = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma_1}, P_1)$ and $G_2 = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma_2}, P_2)$ be induction grammars such that $\overline{\gamma_1}$ and $\overline{\gamma_2}$ are disjoint and P_i contains a production $\overline{\gamma}_i \to \overline{t}$ for $i \in \{1, 2\}$ such that $\overline{t}$ does not contain any nonterminals. Then we define the induction grammar $G_1 \cup G_2 = (\tau, \alpha, (\overline{\nu_c})_c, (\overline{\gamma_1}, \overline{\gamma_2}), P')$ with the following productions P':

$$\begin{aligned} \tau &\to t && \text{if } \tau \to t \in P_1 \cup P_2 \\ (\overline{\gamma_1}, \overline{\gamma_2}) &\to (\overline{t_1}, \overline{t_2}) && \text{if } \overline{\gamma_1} \to \overline{t_1} \in P_1, \overline{\gamma_2} \to \overline{t_2} \in P_2, \text{ and} \\ & && (\overline{\gamma_1}, \overline{\gamma_2}) \to (\overline{t_1}, \overline{t_2}) \text{ is a possible production} \end{aligned}$$

Lemma 3.12.12. $L(G_1 \cup G_2, t) = L(G_1, t) \cup L(G_2, t)$.

Proof. Let δ be derivation in $I(G_1 \cup G_2, t)$. Then δ contains a production $\tau \to t \in P_1 \cup P_2$. Without loss of generality, assume that $\tau \to t \in P_1$. We can now construct a derivation of the same term in $I(G_1, t)$ by replacing every production $(\overline{\gamma_1}, \overline{\gamma_2}) \to (\overline{t_1}, \overline{t_2})$ in δ by $\overline{\gamma_1} \to \overline{t_1}$.

If on the other hand δ is a derivation in $I(G_1, t)$ (the case for G_2 is symmetric), then we can construct a derivation in $I(G_1 \cup G_2, t)$ as follows: first, pick some production $\overline{\gamma_2} \to \overline{t'}$ in G_2 such that $\overline{t'}$ does not contain any nonterminals. Now replace every production $\overline{\gamma_1} \to \bar{t}$ by $(\overline{\gamma_1}, \overline{\gamma_2}) \to (\bar{t}, \overline{t'})$ □

Theorem 3.12.8. *Ind-Equivalence is undecidable.*

Proof. We can reduce Ind-Containment to Ind-Equivalence by noting that $L(G_1, t) \subseteq L(G_2, t) \leftrightarrow L(G_1 \cup G_2, t) = L(G_2, t)$. □

3.12.6 Minimization

The minimization problem for induction grammars is also NP-complete. For the hardness proof we can use essentially the same reduction from Set Cover as in Theorem 3.8.1.

Problem 3.12.10 (Ind-Minimization)**.**
Given an induction grammar $G = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$, a finite family of languages $(L_i)_{i \in I}$ such that $L(G, i) \supseteq L_i$ for all i, and $n \geq 0$, is there a subset $P' \subseteq P$ of the productions such that $|P'| \leq n$ and $L(G', i) \supseteq L_i$ for all i where $G' = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P))$?

Theorem 3.12.9. *Ind-Minimization is* NP*-complete.*

Proof. The problem is in NP since we can guess the subset of productions in polynomial time and checking $L(G, i) \supseteq L_i$ is in NP. Hardness follows by a reduction from Problem 3.8.1, which is NP-complete by Theorem 3.8.1. Let $G = (N, \Sigma, P, A)$ be a VTRATG, $n \geq 0$, L a set such that $L(G) \supseteq L$. Then $G' \subseteq G$ iff $\text{Embed}_{G'} \subseteq \text{Embed}_G$, $L(G') \supseteq L$ iff $L(\text{Embed}_{G'}, |N|) \supseteq L$ by Lemma 3.12.3, and $|\text{Embed}_{G'}| \leq n$ iff $|G'| \leq n$. □

We can also show NP-completeness for the minimization problem for induction grammars with bounded $|\overline{\gamma}|$. However this requires a different reduction from Regular-2-Cover:

Problem 3.12.11 ($(|\overline{\gamma}| \leq k)$-Ind-Minimization)**.**
Given an induction grammar $G = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$ such that $|\overline{\gamma}| \leq k$, a finite

family of languages $(L_i)_{i\in I}$ such that $L(G,i) \supseteq L_i$ for all i, and $n \geq 0$, is there a subset $P' \subseteq P$ of the productions such that $|P'| \leq n$ and $L(G',i) \supseteq L_i$ for all i where $G' = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P))$?

Theorem 3.12.10. *($|\overline{\gamma}| \leq k$)-Ind-Minimization is* NP*-complete for $k \geq 1$.*

Proof. By reduction from Regular-2-Cover, which is NP-complete by Theorem 3.7.3. We can reduce Regular-2-Cover to a grammar minimization problem on acyclic regular grammars (for words): Regular-2-Cover requires us to find (the size of) a minimal grammar $G' = (N, \Sigma, P', A)$ with $N = \{A, B\}$ and $L(G') \supseteq L$ for some given L. We can assume that the right-hand sides of productions in G' are all substrings of words in L (otherwise they cannot in a derivation of a word in L). Since L is a set of words, there are only quadratically many substrings of L. Hence we can define an acyclic regular grammar $G = (N, \Sigma, P, A)$ with $P = \{C \to w \mid C \in N, w \text{ substring of word in } L\}$ such that $G' \subseteq G$. Instead of finding a minimal grammar covering L, we can now search for a minimal subgrammar $G' \subseteq G$ such that $L(G') \supseteq L$ (an instance of VTRATG-Minimization if we treat words as terms). As in Theorem 3.12.9, we can reduce it to ($|\overline{\gamma}| \leq k$)-Ind-Minimization using Lemma 3.12.3: we now search for a minimal $H' \subseteq \mathrm{Embed}_G$ with $L(H', 1) \supseteq L$. Inspecting Definition 3.12.2, we see that $|\overline{\gamma}| = 1$. □

3.12.7 Cover

The complexity of the cover problem is hard to determine for induction grammars as well. Even showing that the problem is in NP requires some effort. And just as for the other types of tree grammars considered in this chapter, it is open whether Ind-Cover is NP-complete:

Problem 3.12.12 (Ind-Cover)**.**
Given a finite family of languages $(L_i)_{i\in I}$ and $n \geq 0$, is there an induction grammar G such that $|G| \leq n$ and $L(G,i) \supseteq L_i$ for all i?

Open Problem 3.12.1. Is Ind-Cover NP-complete?

To show that Ind-Cover is in NP, we will need to bound the size of the nonterminal vector $\overline{\gamma}$. So as a first step, let us show that the cover problem is in NP if we bound the size of $\overline{\gamma}$:

Problem 3.12.13 ($(|\overline{\gamma}| \leq k)$-Ind-Cover)**.**
Given a finite family of languages $(L_i)_{i \in I}$ and $n \geq 0$, is there an induction grammar G such that $|G| \leq n$, $|\gamma| \leq k$, and $L(G, i) \supseteq L_i$ for all i?

In the following proof of Lemma 3.12.13, we do not just get an algorithm for every k but a single polynomial-time algorithm that works for all k. In short, the assumption $|\overline{\gamma}| \leq k$ is only necessary because we do not know yet that $|\overline{\gamma}|$ is polynomially bounded.

Lemma 3.12.13. $(|\overline{\gamma}| \leq k)$-*Ind-Cover* $\in$ NP.

Proof. Let us first compute an upper bound to the productions we might need in a covering induction grammar: each term $t \in L_i$ can use at most $|\mathrm{st}(i)| + 1$ productions in $I(G, i)$ since $I(G, i)$ has only $|\mathrm{st}(i)|+1$ many nonterminal vectors. Hence $|P| \leq \sum_i |L_i|(|\mathrm{st}(i)| + 1)$. We can also assume that each term in the right-hand side of a production in P subsumes a subterm of some $t \in L_i$ for some i (otherwise we can replace the term in the right-hand side by a constant). Therefore the symbolic size of each production is bounded by k times the symbolic size of $(L_i)_{i \in I}$. The symbolic size of a covering grammar is hence polynomial-sized since both the symbolic size of each production as well as the number of productions is bounded polynomially in the symbolic size of the input and k. Hence we can guess the induction grammar in polynomial time, and checking $L(G, i) \supseteq L_i$ is also in NP by Theorem 3.2.1. □

Open Problem 3.12.2. Is $(|\overline{\gamma}| \leq k)$-Ind-Cover NP-complete?

For VTRATGs we could use Theorem 3.7.1 to bound the size of the nonterminal vectors by exhibiting a trivial covering VTRATG of optimal size. However the same trick does not work for induction grammars. Here we will take a given covering grammar and iteratively reduce the size of the nonterminal vectors until they are bounded by a suitable polynomial. Let us first show this approach on VTRATGs:

Lemma 3.12.14. *Let L be a set of terms and $G = (N, \Sigma, P, A)$ a VTRATG such that $L(G) \supseteq L$. Then there exists a VTRATG $G' = (N', \Sigma, P', A)$ such that $|G'| \leq |G|$, $|N'| = |N|$, $L(G') \supseteq L$ and $|\overline{B}| \leq |N| \sum_{t \in L} |\mathrm{Pos}(t)|$ for all $\overline{B} \in N$.*

Proof. Let δ_t be a $\overset{1}{\Rightarrow}$-derivation of t in G for every $t \in L$. That is, $A = \delta_{t,1} \overset{1}{\Rightarrow} \delta_{t,2} \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} \delta_{t,n_t} = t$ for every t. Let $B_{t,i}$ be the nonterminal that is being replaced in $\delta_{t,i}$, and define the set of nonterminals used in the derivation of t as $N_t = \{B_{t,1}, \ldots, B_{t,n_t-1}\} \subseteq \bigcup N$. We can now count the number of these nonterminals and obtain $|N_t| \leq n_t \leq |\mathrm{Pos}(t)||N|$ by Lemma 2.6.1.

Since the derivation δ_t only contains nonterminals from N_t, we can change the right-hand side of, or even drop, any other nonterminal in $\bigcup N \setminus N_t$, and t can still be derived in the changed grammar. In total, we used $|\bigcup_{t\in L} N_t| \leq \sum_{t\in L} |N_t| \leq |N| \sum_{t\in L} |\mathrm{Pos}(t)|$ nonterminals. We then obtain $G' = (N', \Sigma, P', A)$ by deleting all nonterminals in $\bigcup N \setminus N_t$ from G (and also deleting the corresponding lines in the productions). The number of nonterminal vectors in G' is the same as in G since we did not remove any of them, and the length of every nonterminal vector $\overline{B} \in N'$ is bounded by $|\overline{B}| \leq |\bigcup N| \leq |N| \sum_{t\in L} |\mathrm{Pos}(t)|$. □

Using Lemma 3.12.14 we can bound the size of a nonterminal vector in a covering VTRATG of minimal size (in a different way than Theorem 3.7.1): there are at most $|P| \leq |L|$ productions, hence at most $|N| \leq |P|$ nonterminal vectors, and thus $|\overline{B}| \leq |L| \sum_{t\in L} |\mathrm{Pos}(t)|$, a polynomial bound in the symbolic size of L.

Lemma 3.12.15. *Let $(L_i)_{i\in I}$ be a finite family of languages, and the induction grammar $G = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$ be such that $L(G, i) \supseteq L_i$ for all i. Then there exists an induction grammar $G' = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma'}, P')$ such that $|G'| \leq |G|$, $L(G', i) \supseteq L_i$ for all i, and $|\overline{\gamma'}| \leq \sum_{i\in I}(1 + |\mathrm{st}(i)|) \sum_{t\in L_i} |\mathrm{Pos}(t)|$.*

Proof. As in Lemma 3.12.14, we count the number of nonterminals used in the derivations of $t \in L_i$. The instance grammar $I(G, i)$ has $1 + |\mathrm{st}(i)|$ nonterminal vectors. So the derivations use at most $\sum_{i\in I}(1 + |\mathrm{st}(i)|) \sum_{t\in L_i} |\mathrm{Pos}(t)|$ nonterminals in the instance grammars. Hence we can only need to keep this number of nonterminals in $\overline{\gamma}$. □

Theorem 3.12.11. *Ind-Cover* ∈ NP.

Proof. By reduction to ($|\overline{\gamma}| \leq k$)-Ind-Cover: if there is a covering induction grammar $G = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$ with $|G| \leq n$, then there is one with $|\overline{\gamma}| \leq \sum_{i\in I}(1 + |\mathrm{st}(i)|) \sum_{t\in L_i} |\mathrm{Pos}(t)|$ by Lemma 3.12.15. As in Lemma 3.12.13,

this gives a polynomial bound on the symbolic size of a covering induction grammar. We can then guess an induction grammar of that size and check that it covers $(L_i)_{i \in I}$. □

4 Practical algorithms to find small covering grammars

We saw in Chapter 2 that cut- and induction-elimination correspond to the computation of the language of grammars. To reverse cut-elimination and induction-elimination we need to find a grammar that covers a given language. We are hence looking for algorithms that achieve the following:

1. Given a finite set of terms L, return a VTRATG G such that $L(G) \supseteq L$.
2. Given a finite family of terms $(L_i)_{i \in I}$ (where I is a finite set of free constructor terms), return an induction grammar G such that $L(G, i) \supseteq L_i$ for all $i \in I$.

There are trivial solutions for these problems as formulated above. For example, given a finite set of terms L we might return a VTRATG with one nonterminal A and productions $A \to t$ for every $t \in L$. However such trivial grammars correspond to trivial cuts. If we want to introduce interesting structure into proofs, we need to impose extra conditions on the grammar. One condition that we have already seen in TRATG-Cover was that the size of the grammar should be minimal. We also want to avoid another kind of triviality: there are trivial VTRATGs of minimal size if we do not bound the nonterminal vectors (by Corollary 3.7.1 and Theorem 3.7.1). From a practical point of view however, we also present algorithms which do not always produce grammars of minimal size. For example, the Reforest algorithm we will describe in Section 4.5 will be fast but without any guarantees on the size of the resulting grammar. Only one of the algorithms in this chapter will have such a correctness guarantee to always return a grammar of minimal size, solving the following problems:

Problem 4.0.1 (Parameterized Language Cover)**.**
Given a finite set of terms L and a sequence of nonterminal vectors N, find a VTRATG $G = (N, \Sigma, P, A)$ of minimal size such that $L(G) \supseteq L$.

Problem 4.0.2 (Parameterized Indexed Termset Cover)**.**
Given a finite family of sets of terms $(L_i)_{i \in I}$, a structurally inductive datatype ρ with constructors $(\overline{\nu_c})_c$, and a nonterminal vector $\overline{\gamma}$, find an induction grammar $G = (\tau, \alpha, (\overline{\nu_c})_c, \overline{\gamma}, P)$ of minimal size such that $L(G, i) \supseteq L_i$ for all $i \in I$.

Describing a large set of terms using a small grammar can be viewed as a kind of compression. For strings, grammars are used in several popular compressions algorithms [91, 75, 64, 63]. Tree grammars are a standard technique for the compression of XML documents [86]. In the context of data compression, grammars have the practical advantage that many operations can be directly performed on the compressed representation [65]. These compression algorithms differ from our application in several aspects:

1. Such grammar-based algorithms for text or document compression typically compress a single string or tree. We compress a set of terms or even a family of sets of terms.

2. For compression algorithms, we can choose any class of grammar. The main objective is to produce a small description, with the potential goal of supporting operations on the compression representation. By contrast, we need to use classes of grammars that correspond to the cut/induction-elimination operation, that is, VTRATGs and induction grammars.

3. We are considering a form of lossy compression: when compressing a finite set of terms L, we are looking for a minimal grammar G such that $L(G) \supseteq L$ instead of reproducing L perfectly.

In this chapter we will describe three algorithms that find covering VTRATGs. Section 4.2 describing the delta-table is based on [36, Section 3], and the algorithm in Section 4.3 based on MaxSAT was previously published as the journal article [29]. Both articles [36, 29] only treat the single-sorted case, in this chapter we consider the general case for many-sorted terms and grammars.

4.1 Least general generalization and matching

Least general generalization (alternatively called anti-unification) is a useful operation on terms that is central to several algorithms in this chapter. As the name implies, it finds a common generalization of two (or more) terms. This operation was first introduced by Plotkin [80, 79] and Reynolds [83].

Definition 4.1.1. Let $t, s \in \mathcal{T}(\Sigma \cup X)$ be terms. The term t subsumes s (written $t \preceq s$) iff there exists a substitution σ such that $t\sigma = s$.

If $t \preceq s$ then t and s have the same type since $s = t\sigma$ for some σ and substitution preserves types. The subsumption relation $\preceq$ is a preorder on $\mathcal{T}(\Sigma \cup X)$. Every preorder induces an equivalence relation where $t \approx s$ iff $t \preceq s$ and $s \preceq t$. In the subsumption order, two terms are equivalent if they differ only by variable renaming. If t, s are terms without common variables, and σ a most general unifier of t and s, then $t\sigma = s\sigma$ is their least upper bound. The least general generalization is then the greatest lower bound in this order, and can be recursively computed as follows:

Definition 4.1.2. The least general generalization $\mathrm{lgg}(t, s)$ of two terms $t, s \in \mathcal{T}(\Sigma \cup X)$ of the same type can be computed recursively, where $x_{t,s}$ is a different variable for each pair (t, s):

$$\mathrm{lgg}(f(t_1, \ldots, t_n), f(s_1, \ldots, s_n)) = f(\mathrm{lgg}(t_1, s_1), \ldots, \mathrm{lgg}(t_n, s_n))$$
$$\mathrm{lgg}(t, s) = x_{t,s} \quad \text{otherwise}$$

Example 4.1.1. $\mathrm{lgg}(f(c, c, e), f(d, d, e)) = f(x, x, e)$.

Note that the least general generalization is only unique up to variable renaming, and we will often choose the variables to be fresh. As a greatest lower bound of terms in the subsumption preorder, the least general generalization satisfies $\mathrm{lgg}(t, s) \approx \mathrm{lgg}(s, t)$, $\mathrm{lgg}(\mathrm{lgg}(t, s), r) \approx \mathrm{lgg}(t, \mathrm{lgg}(s, r))$, and also $\mathrm{lgg}(t, s) \preceq t$ for all terms t, s, r. Since the least general generalization is associative-commutative, we will also write $\mathrm{lgg}(L)$ for the least general generalization of a non-empty set of terms L of the same type; this is even well-defined if L is infinite: given a term t there are only finitely many terms s such that $s \preceq t$ (up to variable renaming):

Lemma 4.1.1. *Let $t \in \mathcal{T}(\Sigma \cup X)$. Then the set $\{s \in \mathcal{T}(\Sigma \cup X) \mid s \preceq t\}$ of lower bounds of t is finite (identifying terms up to renaming).*

Proof. Each term $s \preceq t$ can be uniquely described by its set of positions $\mathrm{Pos}(s) \subseteq \mathrm{Pos}(t)$ and the equivalence relation $\equiv_s$ on these subterms, where $p \equiv_s q \leftrightarrow s|_p = s|_q$. The set of positions determines at which positions in s there are variables, and $\equiv_s$ determines which of the variables are equal. There are only finitely many such subsets and equivalence relations. □

Definition 4.1.3. Let $t, s \in \mathcal{T}(\Sigma \cup X)$ be terms such that $t \preceq s$. We define s/t as the unique substitution such that $t(s/t) = s$, and $x(s/t) = x$ for all variables $x \notin \mathrm{FV}(t)$.

The substitution s/t is called a matching from t to s. The set of matching substitutions to the lgg are in a sense orthogonal: they can distinguish any two terms.

Lemma 4.1.2. *Let $T \subseteq \mathcal{T}(\Sigma)$ be a nonempty set of ground terms of the same type, and $r, s \in \mathcal{T}(\Sigma \cup X)$. Let $u = \mathrm{lgg}(T)$, and for every $t \in T$, let $\iota_t := t/u$. If $r\iota_t = s\iota_t$ for all $t \in T$, then $r = s$.*

Proof. By induction on r. The critical case is when r is a variable and $r \notin \mathrm{FV}(s)$.

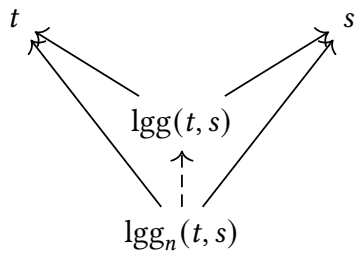
If $r \notin \mathrm{FV}(u)$, then $r\iota_t = r$ for all t and hence s is necessarily a variable as well. When additionally $s \in \mathrm{FV}(u)$, we can choose two terms t and t' such that $s\iota_t \neq s\iota_{t'}$ (otherwise u would not be the greatest lower bound), a contradiction to $s\iota_t = s\iota_{t'} = r$. Otherwise $s \notin \mathrm{FV}(u)$ and $s = s\iota_t = r\iota_t = r$ with any t.

Hence $r \in \mathrm{FV}(u)$. We have already handled the case that s is a variable and $s \notin \mathrm{FV}(u)$ by symmetry. If $s \in \mathrm{FV}(u)$, then we have $r \neq s$ because of $r \notin \mathrm{FV}(s)$ and there exists a t such that $r\iota_t \neq s\iota_t$.

So we have that $s = f(s_1, \ldots, s_n)$ is a function application. Choose terms t and t' such that $r\iota_t$ and $r\iota_{t'}$ have a different root function symbol. One of the root symbols is different from f, a contradiction. □

We can obtain an interesting variant on the least general generalization by considering a restriction to the terms with at most n variables $\mathcal{T}_n(\Sigma \cup X) \subset \mathcal{T}(\Sigma \cup X)$. Let us call the greatest lower bound on this subset $\mathrm{lgg}_n(t, s)$, if it exists. Observe that if $\mathrm{lgg}_n(t, s)$ exists, then clearly $\mathrm{lgg}_n(t, s) \preceq t$ and

$\mathrm{lgg}_n(t,s) \preceq s$ so necessarily $\mathrm{lgg}_n(t,s) \preceq \mathrm{lgg}(t,s)$. This generalization of the lgg will be used as an alternative way to compute generalizations in the delta-table algorithm in Section 4.2.



Another way to view this situation is to consider the map $\mathrm{lgg}(t,s) \mapsto \mathrm{lgg}_n(t,s)$: this is the partial function π_n that assigns to every term $t \in \mathcal{T}(\Sigma \cup X)$ the greatest lower bound in $\mathcal{T}_n(\Sigma \cup X)$ if it exists. In that case we have $\mathrm{lgg}_n(t,s) = \pi_n(\mathrm{lgg}(t,s))$.

Definition 4.1.4. Let n be a natural number, then $\pi_n : \mathcal{T}(\Sigma \cup X) \to \mathcal{T}_n(\Sigma \cup X)$ is the partial function such that $\pi_n(t)$ is the greatest lower bound of t in $\mathcal{T}_n(\Sigma \cup X)$ if it exists, and undefined otherwise.

Lemma 4.1.3. $\mathrm{lgg}_n(t,s) = \pi_n(\mathrm{lgg}(t,s))$ *if either side exists.*

Proof. Both sides of the equation are defined as greatest lower bounds: the left-hand side of the set $\{t,s\}$, and the right-hand side of $\{\mathrm{lgg}(t,s)\}$. However these two sets have the same lower bounds: $r \preceq \{t,s\} \Leftrightarrow (r \preceq t \wedge r \preceq s) \Leftrightarrow r \preceq \mathrm{lgg}(t,s)$. □

Example 4.1.2. $\pi_1(f(x,y)) = x$, $\pi_2(f(x,y)) = f(x,y)$, $\pi_0(f(x,y))$ is not defined.

We can now characterize the numbers n for which lgg_n is a total function. The results depends a bit on the signature: if the signature only contains unary function symbols and constants, then every lgg could contain at most one variable and lgg_n would be total for all $n \geq 1$. Therefore we require that there exists at least one constant and one binary function symbol (ternary, quaternary, quinary, etc. function symbols would also suffice).

Theorem 4.1.1. *Let n be a natural number, and Σ a signature containing a constant c^α and a binary function symbol $f^{\alpha\to\alpha\to\alpha}$ for some type α. Then the following are equivalent:*

1. $\mathrm{lgg}_n(t,s)$ *exists for all* $t,s \in \mathcal{T}_n(\Sigma \cup X)$ *of the same type.*
2. $\pi_n(t)$ *exists for all* $t \in \mathcal{T}(\Sigma \cup X)$.
3. $n \in \{1,2\}$.

Proof. $2 \Rightarrow 1$. By Lemma 4.1.3.

$1 \Rightarrow 2$. We show that for every term $t \in \mathcal{T}(\Sigma \cup X)$ there exist ground terms $t_1, t_2 \in \mathcal{T}(\Sigma) \subseteq \mathcal{T}_n(\Sigma \cup X)$ such that $t = \mathrm{lgg}(t,s)$ (up to renaming). Once we have showed that, $\pi_n(t)$ exists by Lemma 4.1.3. Define the sequence of terms $(d_i)_{i\in\mathbb{N}}$ by $d_0 = c$ and $d_{i+1} = f(c,d_i)$. These terms will be used as a kind of additional constants: we have for example $\mathrm{lgg}(d_0,d_1) = x$ and $\mathrm{lgg}(f(d_0,d_0),f(d_i,d_j)) = f(x,y)$ for $i \neq j$. Let $\mathrm{FV}(t) = \{x_1,\dots,x_m\}$. We can now set $t_1 = t[x_1\backslash d_0, x_2\backslash d_0, \dots, x_m\backslash d_0]$ and $t_2 = t[x_1\backslash d_1, x_2\backslash d_2, \dots, x_m\backslash d_m]$ and have $t = \mathrm{lgg}(t_1,t_2)$ (up to renaming).

$3 \Rightarrow 2$. We can construct $\pi_n(t)$ explicitly as the supremum of all lower bounds of t, i.e., $\pi_n(t) = \bigvee\{s \in \mathcal{T}_n(\Sigma \cup X) \mid s \preceq t\}$. There are only finitely many such lower bounds by Lemma 4.1.1. We only need to show that $\{s \in \mathcal{T}_n(\Sigma\cup X) \mid s \preceq t\}$ is nonempty, and that $\mathcal{T}_n(\Sigma\cup X)$ is closed under most general unification (i.e., binary least upper bounds) of unifiable terms (in $\mathcal{T}(\Sigma \cup X)$). The set is nonempty because it contains all variables of the same type as t. For closure under most general unification, consider the standard algorithm to compute the most general unifier of two terms s_1 and s_2 (in our case, s_1 and s_2 have disjoint variables): we start with the equation $s_1 \doteq s_2$ and iteratively decompose it to obtain a unifying substitution. In general, the number of variables in the unified term is bounded by the number of variables in s_1 and s_2; every variable that is assigned during unification reduces this number by one. If s_1 and s_2 have one or two variables in total, then the unified term will have at most $2-1$ variables since one variable will be assigned. Hence $\mathcal{T}_1(\Sigma \cup X)$ is closed under most general unification (of unifiable terms). If s_1 has one variable and s_2 has two variables, the resulting unified term has at most two variables by the same argument. If both s_1 and s_2 have two variables, then

pick one variable $s_i|_p$ for some i with $|p|$ minimal, and call it x. This variable will be assigned during unification. Since s_i contains two variables, there is another variable $y = s_i|_q$. Unless there exists a variable $z = s_{3-i}|_{q'}$ for a prefix q' of q, the variable y will be assigned during unification. Otherwise $z \neq x$ will be assigned. In either case, the unified term will have at most $4 - 2$ variables and $\mathcal{T}_2(\Sigma \cup X)$ is closed under most general unification (of unifiable terms).

$\neg 3 \Rightarrow \neg 2$. If $n = 0$, then $\pi_0(x)$ would need to be a variable and is hence not defined. For $n > 2$, we simulate i-ary function symbols g_i by defining $g_0 = c$ and $g_{i+1}(t_1, t_2, \ldots, t_{i+1}) = f(t_1, g_i(t_2, \ldots, t_{i+1}))$. Consider the term $t = g_{n-1}(x_1, x_2, x_3, \ldots, x_{n-3}, f(x_{n-2}, x_{n-1}), f(x_n, x_{n+1})) \in \mathcal{T}(\Sigma \cup X) \setminus \mathcal{T}_n(\Sigma \cup X)$. It suffices to exhibit two terms $a, b \in \mathcal{T}_n(\Sigma \cup X)$ such that $a, b \preceq t$ and that their least upper bound (in $\mathcal{T}(\Sigma \cup X)$) is given by $a \vee b = \mathrm{lgg}(t, s) \notin \mathcal{T}_n(\Sigma \cup X)$:

$$a = g_{n-1}(x_1, x_2, x_3, \ldots, x_{n-2}, f(x_n, x_{n+1}))$$
$$b = g_{n-1}(x_1, x_2, x_3, \ldots, f(x_{n-2}, x_{n-1}), x_n)$$ □

Example 4.1.3. $\mathrm{lgg}_1(f(g(c,d), g(c,d)), f(g(a,b), g(a,b))) = f(x,x)$, $\mathrm{lgg}_2(f(g(c,d), g(c,d)), f(g(a,b), g(a,b))) = f(g(x,y), g(x,y))$.

4.2 Delta-table

The delta-table algorithm was first described in [49, 48] and produces VTRATGs with two non-terminal vectors $A, \overline{B}$ where the length of the nonterminal vector $\overline{B}$ is determined by the algorithm. The algorithm is based on the observation that we can to some degree reverse language generation and recover the original productions of a VTRATG by computing the lgg of subsets of the language:

Example 4.2.1. Let $G = (A, \{A, B\}, \Sigma, P)$ be a VTRATG with the set of productions $P = \{A \to f(B, h(B)) \mid g(B,B), B \to c \mid d\}$. This grammar generates the language $L(G) = \{f(c, h(c)), g(c,c), f(d, h(d)), g(d,d)\}$. We can recover the productions of the nonterminal A by computing $\mathrm{lgg}\{f(c, h(c)), f(d, h(d))\} = f(x, h(x))$ as well as $\mathrm{lgg}\{g(c,c), g(d,d)\} = g(x,x)$. The productions of the nonterminal B can be obtained from the corresponding matching substitutions: $g(c,c)/g(x,x) = [x\backslash c]$ and $g(d,d)/g(x,x) = [x\backslash d]$.

The algorithm is based on an operation called delta-vector, which computes substitutions that correspond to VTRATGs with only one production from the start symbol A covering subsets of the input language L. They are stored in a data structure called delta-table, which is later processed and combined into the covering VTRATG. The delta-table algorithm is incomplete in the sense that it does not always return a VTRATG of minimal size, as we will see in Section 4.2.3.

4.2.1 The delta-vector

Given a non-empty set of ground terms T, the delta-vector for T is a pair of the lgg and the matching substitutions. To have a canonical result term, we use the names $x_1, \ldots, x_n$ for the variables in $\mathrm{lgg}(t, s)$, read left-to-right.

Definition 4.2.1. Let T be a finite non-empty set of ground terms of the same type. Then we define its delta-vector as $\Delta(T) = (\mathrm{lgg}(T), \{t/\mathrm{lgg}(T) \mid t \in T\})$.

We can also define the variant $\Delta_n(T) = (\mathrm{lgg}_n(T), \{t/\mathrm{lgg}_n(T) \mid t\ in T\})$. By Theorem 4.1.1, $\Delta_n(T)$ is defined for $n \in \{1, 2\}$. The name delta-vector was introduced in [49] for the operation Δ_1; there it is defined on sequences of terms. The order of the terms in the set T is not relevant; so for simplicity we define the delta-vector on sets here. Δ was then called generalized delta-vector in [48].

Example 4.2.2. Consider the set of terms $T = \{f(a, b), f(c, d)\}$, then:

$$\Delta(T) = (f(x_1, x_2), \{[x_1 \backslash a, x_2 \backslash b], [x_1 \backslash c, x_2 \backslash d]\})$$
$$\Delta_1(T) = (f(x), \{[x \backslash f(a, b)], [x \backslash f(c, d)]\})$$

The delta-vectors correspond to VTRATGs covering a *subset* of the input language L, namely to VTRATGs where there is only a single production of the start symbol A:

Example 4.2.3. Consider the set of terms $T \subseteq L$:

$$T = \{f(c, h(c)), f(d, h(d))\} \subseteq L = \{f(c, h(c)), g(c, c), f(d, h(d)), g(d, d)\}$$

Then the delta-vector is $\Delta(T) = (f(x_1, h(x_1)), \{[x_1 \backslash c], [x_1 \backslash d]\})$ and corresponds to the VTRATG with the following productions:

$$A \to f(x_1, h(x_1))$$
$$B \to c \mid d$$

Observe that the right-hand side of the single production of the start symbol A is the lgg in the delta-vector. In grammars that describe proofs, productions of the start symbol A have the special property that the root symbol of the right-hand side indicates the instantiated formula as in Definition 2.7.2. That is, the right-hand side is not a nonterminal; we cannot have a production of the form $A \to B$. With this observation, we define non-trivial delta-vectors, and will subsequently ignore trivial delta-vectors:

Definition 4.2.2. A delta-vector $\Delta(T) = (u, S)$ is called *non-trivial* if u is not a variable.

4.2.2 The delta-table

The delta-table is a data-structure that stores all *non-trivial* delta-vectors of subsets of L, indexed by their sets of substitutions. Some of these are later combined into a grammar covering L.

Definition 4.2.3. A delta-row is a pair $S \to U$ where S is a set of substitutions, and U is a set of pairs (u, T) such that $uS = \{u\sigma \mid \sigma \in S\} = T$. A delta-table is a map where every key-value pair is a delta-row.

Algorithm 1 computes a delta-table containing the delta-vectors for all subsets of T. As an optimization, we do not iterate over all subsets. Instead we incrementally add terms to the subset, stopping as soon as the delta-vector is trivial. This optimization is justified by the following lemma:

Theorem 4.2.1. *Let T be a set of terms. If $\Delta(T)$ is trivial, then so is $\Delta(T')$ for every $T' \supseteq T$.*

Proof. Let $\Delta(T) = (u, S)$ and $\Delta(T') = (u', S')$. Since we have that $u = \mathrm{lgg}(T) \preceq \mathrm{lgg}(\mathrm{lgg}(T), \mathrm{lgg}(T')) = \mathrm{lgg}(T' \cup T) = \mathrm{lgg}(T')$, there is a substitution σ such that $u'\sigma = u$. So if u is a variable, then u' is necessarily a variable as well. □

Algorithm 1 Delta-table algorithm

```
function Populate(M: delta-table, L: list of terms, T: set of terms)
    if L is non-empty then
        T' ← T ∪ {Head(L)}
        (u, S) ← Δ-vector(T')
        if u is not a variable then
            M[S] ← M[S] + (u, T')
            Populate(M, Tail(L), T')
        end if
        Populate(M, Tail(L), T)
    end if
end function
function DeltaTableAlgorithm(L: set of terms, row-merging: boolean)
    M ← new delta-table
    Populate(M, L, ∅)
    if row-merging then
        MergeSubsumedRows(M)
    end if
    CompleteRows(M, L)
    return FindMinimalGrammar(M)
end function
```

After having computed the delta-table, we use it to compute a covering VTRATG. Each row in the delta-table corresponds to a VTRATG in the following way: given a row $S \to U$, let $x_1, \ldots, x_n$ be the variables in S. The resulting VTRATG $G(S \to U)$ has the nonterminals $A, (B_1, \ldots, B_n)$ and the productions $A \to u$ for $(u, T) \in U$ and $\overline{B} \to \overline{x}\sigma$ for $\sigma \in S$. However this VTRATG may not generate the whole input set L, we have $L(G(S \to U)) = \bigcup_{(u,T)\in U} T$. Therefore, we first "complete" each row of the delta-table by adding the pairs $(t, \{t\})$ to U for every $t \in L$ as a post-processing step.

Then we minimize each of the completed rows by removing pairs from U under the side condition that $\bigcup_{(u,T)\in U} T = L$. The side condition ensures that the resulting VTRATG covers L. Each of the minimized rows then corresponds

to a covering VTRATG, we pick the VTRATG with the smallest size.

4.2.3 Incompleteness

The delta-table algorithm does not always find VTRATGs of minimal size. Consider for example the following set of terms:

$$L = \{r(c), r(f(c)), \ldots, r(f^8(c)), s(d), s(g(d)), \ldots, s(g^8(d))\}$$

There is a covering VTRATG of size 12:

$$\begin{aligned} A &\to r(B) \mid r(f^3(B)) \mid r(f^6(B)) \mid s(B) \mid s(g^3(B)) \mid s(g^6(B)) \\ B &\to c \mid f(c) \mid f(f(c)) \mid d \mid g(d) \mid g(g(d)) \end{aligned}$$

However the set of substitutions corresponding to this small VTRATG does not occur as the second component of the delta-vector for any subset $T \subseteq L$:

$$\{[x_1\backslash c], [x_1\backslash f(c)], [x_1\backslash f(f(c))], [x_1\backslash d], [x_1\backslash g(d)], [x_1\backslash g(g(d))]\}$$

Hence the delta-table algorithm only finds the following VTRATG of size 14 (or the symmetric version where r, f, c and s, g, d are swapped):

$$\begin{aligned} A &\to r(B) \mid r(f^3(B)) \mid r(f^6(B)) \mid s(d) \mid s(g(d)) \mid \cdots \mid s(g^8(d)) \\ B &\to c \mid f(c) \mid f(f(c)) \end{aligned}$$

4.2.4 Row-merging

One approach to ameliorate the issue of incompleteness is to merge rows in the table. Consider as a different example the following set L:

$$\begin{aligned} L &= L_1 \cup L_2 \cup L_3 \\ L_1 &= \{q(a,b,c), q(b,c,a), q(c,a,b)\} \\ L_2 &= \{r(a,b,c), r(b,c,a), r(c,a,b)\} \\ L_3 &= \{s(b,a), s(c,b), s(a,c)\} \end{aligned}$$

The delta-table algorithm will not find the following VTRATG as the substitutions in the delta-vector do not match—for $s(a,b), s(b,c), s(c,a)$ we have a

substitution of two variables, and in the other cases three variables.

$$A \to q(B_1, B_2, B_3) \mid r(B_1, B_2, B_3) \mid s(B_1, B_2)$$
$$\overline{B} \to (a, b, c) \mid (b, c, a) \mid (c, a, b)$$

In particular the delta-table will contain the following two rows (for space reasons, we abbreviate $[x_1\backslash a, x_2\backslash b, x_3\backslash c]$ as $[a, b, c]$ and $[x_1\backslash a, x_2\backslash b]$ as $[a, b]$):

$$\{[a, b, c], [b, c, a], [c, a, b]\} \to \{(q(x_1, x_2, x_3), L_1), (r(x_1, x_2, x_3), L_2)\}$$
$$\{[b, a], [c, b]\} \to \{(s(x_1, x_2), L_3)\}$$

If we could just put the contents of the second row into the first one, then we would find the desired VTRATG immediately. Intuitively, the reason we can merge the rows without violating the invariant of the delta-table algorithm is because the substitutions of the second row are in a sense contained in the substitutions of the first row. The following definition makes this intuition precise:

Definition 4.2.4 (substitution-set subsumption)**.** Let S_1, S_2 be sets of substitutions. Then S_1 subsumes S_2, written $S_1 \preceq S_2$, if and only if there exists a renaming substitution σ with the following property:

$$\forall \tau_1 \in S_1 \quad \exists \tau_2 \in S_2 \quad \forall x \in \mathrm{dom}(\tau_1) \quad x\tau_1 = x\sigma\tau_2$$

The reason why we only consider renaming substitutions here is to simplify the implementation: we do not need to compute matching during subsumption. Substitution-set subsumption is reflexive and transitive; it is also monotone: if $S_1' \subseteq S_1$ and $S_2' \supseteq S_2$, then $S_1 \preceq S_2$ implies $S_1' \preceq S_2'$.

Example 4.2.4. $\{[b, a], [c, b]\} \preceq \{[a, b, c], [b, c, a]\}$ with $\sigma = [x_1\backslash x_2, x_2\backslash x_1]$.

Lemma 4.2.1 (row-merging)**.** *Let $S_1 \to R_1$ and $S_2 \to R_2$ be delta-rows, and $S_1 \preceq S_2$ with the substitution σ witnessing this subsumption. Then $S_2 \to (R_2 \cup R_1\sigma)$ is a delta-row as well.*

Proof. Let $(u, T') \in R_1$. We need to show that $u\sigma S_2 = T'$. But this follows from $uS_1 = T'$ since $S_1 \preceq S_2$ via σ. □

After the initial computation of the delta-table, we use this lemma to merge all pairs of rows where one set of substitutions subsumes the other. Whenever we have rows $S_1 \longrightarrow R_1$ and $S_2 \longrightarrow R_2$ such that $S_1 \preceq S_2$, we replace $S_2 \longrightarrow R_2$ by $S_2 \longrightarrow R_2 \cup R_1\sigma$ and keep the S_1 row as it is. This increases the set of possible VTRATGs that we can find, since we did not remove any elements of the rows. This allows to find the desired VTRATGs in the example. We have $\{[b, a], [c, b]\} \preceq \{[a, b, c], [b, c, a], [c, a, b]\}$ via the substitution $[x_1 \backslash x_2, x_2 \backslash x_1]$, and generate the following new row:

$$\begin{aligned}\{[a, b, c], [b, c, a], [c, a, b]\} \longrightarrow \{&(q(x_1, x_2, x_3), L_1),\\ &(r(x_1, x_2, x_3), L_2),\\ &(s(x_2, x_1), L_3)\}\end{aligned}$$

While row-merging significantly improves the number of grammars the delta-table algorithm can find as we will see in Section 4.6, there are still cases where it will not find a minimal covering grammar. For example, the example from Section 4.2.3 will still produce a non-minimal grammar of size 14.

4.3 Using MaxSAT

In this section we present a second algorithm to generate covering VTRATGs. Compared to the delta-table algorithm this algorithm has several advantages: it always generates grammars of minimal size, and with any desired (but fixed) number of nonterminals. MaxSAT is an optimization variant of the Boolean satisfaction problem (SAT); the algorithm uses a MaxSAT solver as a backend, and proceeds in the following three steps:

1. Compute a large grammar that covers the term set and contains a covering subgrammar of minimal size, in polynomial time.

2. Produce a MaxSAT problem that encodes the minimization of this large grammar.

3. Use an off-the-shelf solver to obtain a solution to the MaxSAT problem, and return the minimal VTRATG corresponding to this solution.

The crucial and critical challenge here is to define this large grammar, show that it contains a covering subgrammar of minimal size, and that it is efficiently computable. To this end we will define a rewriting operation $\sqsubset_L$ on terms describing the regularities of terms in the input term set L in Section 4.3.2. Such a rewriting operation defines a set of normal forms—and the large grammar $S_{N,L}$ will contain exactly those productions whose right-hand side is in normal form with regards to $\sqsubset_L$.

In order to show that $S_{N,L}$ contains a subgrammar of minimal size, we will lift the rewriting operation to grammars and languages in Section 4.3.1 and show that these two lifted rewriting operations commute in Theorem 4.3.1. The main tool to show the polynomial-time computability of $S_{N,L}$ will be Theorem 4.3.5, which gives a strong characterization of the normal forms of $\sqsubset_L$ in terms of the lgg. The encoding of the minimization problem in Section 4.3.5 is comparatively straightforward.

4.3.1 Rewriting grammars

The goal of this section is to show that term rewriting and language generation commutes, i.e., instead of applying a rewriting relation to a grammar and then generate the language, we can also generate the language from the original grammar and apply the rewriting to the language. (Note that the rewrite relation we will be using is non-confluent, and that the results of rewriting are therefore not deterministic.)

$$\begin{array}{ccc} G & \xrightarrow{R} & G' \\ \downarrow & & \downarrow \\ L(G) & \dashrightarrow^{R} & L(G') \end{array}$$

Following [27], we will define a (single-step) rewrite relation as binary relation that is closed under substitution (called fully invariant) and congruence (called monotonic). Since we work with many-sorted terms, we also require that both sides of the relation have the same type:

Definition 4.3.1. Let $\rightarrow_R$ be a binary relation on $\mathcal{T}(\Sigma \cup X)$. Then $\rightarrow_R$ is called *type-preserving* iff t and s such that $t \rightarrow_R s$ the terms t and s have the same type.

Definition 4.3.2. Let $\to_R$ be a type-preserving binary relation on $\mathcal{T}(\Sigma \cup X)$. Then $\to_R$ is called *monotonic* if $s \to_R t$ implies $u[s]_p \to_R u[t]_p$ for all $s, t, u \in \mathcal{T}(\Sigma \cup X)$ and $p \in \mathrm{Pos}(u)$. It is called *fully invariant* if $s \to_R t$ implies $s\sigma \to_R t\sigma$ for all $s, t \in \mathcal{T}(\Sigma \cup X)$ and substitutions σ. It is called a *rewrite relation* if it is both monotonic and fully invariant.

We will now show how to lift the rewriting from terms to productions, to derivations, to grammars, and then describe the effect the rewriting has on the generated language. For the rest of the section, let $\to_R$ be a fixed rewrite relation, and $\to_R^*$ its reflexive and transitive closure. (In the following sections, the relation will always be $\to_R = \to_R^* = \sqsubseteq_L$, which we will define in Definition 4.3.4.)

Definition 4.3.3. Let $N = \{A_0, \overline{A_1}, \ldots, \overline{A_n}\}$, $L \subseteq \mathcal{T}(\Sigma \cup X \cup N)$ be a set of terms, $G = (N, \Sigma, P, A_0)$ and $G' = (N, \Sigma, P', A_0)$ VTRATGs, $p = (\overline{B} \to \overline{s}) \in P$, $p' = (\overline{B} \to \overline{s'}) \in P'$ be productions, and $\delta = [\overline{A_0} \backslash \overline{s_0}][\overline{A_1} \backslash \overline{s_1}] \cdots [\overline{A_n} \backslash \overline{s_n}]$, $\delta' = [\overline{A_0} \backslash \overline{s'_0}][\overline{A_1} \backslash \overline{s'_1}] \cdots [\overline{A_n} \backslash \overline{s'_n}]$ be derivations. We extend rewriting on terms to sets, derivations, productions, and grammars in the natural way as follows:

- $L \to_R^* L'$ iff for all $t' \in L'$ there exists a $t \in L$ such that $t \to_R^* t'$ and for all $t \in L$ there exists a $t' \in L'$ such that $t \to_R^* t'$.
- $\delta \to_R^* \delta'$ iff $s_{i,j} \to_R^* s'_{i,j}$ for all i and j.
- $(\overline{B} \to \overline{s}) \to_R^* (\overline{B} \to \overline{s'})$ iff $s_j \to_R^* s'_j$ for all j.
- $G \to_R^* G'$ iff for all $p' \in P'$ there exists a $p \in P$ such that $p \to_R^* p'$ and for all $p \in P$ there exists a $p' \in P'$ such that $p \to_R^* p'$.

The extension of $\to_R^*$ from terms to sets of terms is a very natural definition that coincides with the image $\mathcal{P}(\to_R^*)$ under the usual powerset functor $\mathcal{P}\colon \mathbf{Rel} \to \mathbf{Rel}$ on the category of sets with relations as morphisms [85, 11]. There, the powerset functor maps a relation $R \subseteq A \times B$ to a relation $\mathcal{P}(R) \subseteq \mathcal{P}(A) \times \mathcal{P}(B)$ on the powersets such that $a\, \mathcal{P}(R)\, b \leftrightarrow (\forall x \in a \exists y \in b\, xRy) \wedge (\forall y \in b \exists x \in a\, xRy)$.

The following two easy lemmas allow us to move rewriting out of the right-hand side of a production, and to the end of a derivation, respectively:

Lemma 4.3.1. *Let t be a term, and $[\overline{x}\backslash\overline{s}]$ a substitution. If $s_i \to_R^* s_i'$ for all i, then $t[\overline{x}\backslash\overline{s}] \to_R^* t[\overline{x}\backslash\overline{s'}]$.*

Proof. This follows from the fact that $\to_R^*$ is monotonic and reflexive-transitive. □

Lemma 4.3.2. *Let σ be a substitution, and t, t' terms. If $t \to_R^* t'$, then $t\sigma \to_R^* t'\sigma$.*

Proof. The result is clear for single step rewritings, and then extends to the transitive closure. □

Using these two lemmas, we can now lift rewriting to derivations:

Lemma 4.3.3. *Let δ, δ' be derivations such that $\delta \to_R^* \delta'$. Then $t\delta \to_R^* t\delta'$ for any term t.*

Proof. Let $\delta = [\overline{A_0}\backslash\overline{s_0}][\overline{A_1}\backslash\overline{s_1}]\cdots[\overline{A_n}\backslash\overline{s_n}]$, and $\delta' = [\overline{A_0}\backslash\overline{s_0'}][\overline{A_1}\backslash\overline{s_1'}]\cdots[\overline{A_n}\backslash\overline{s_n'}]$, We will now iteratively change the derivation δ to the rewritten derivation $\delta_i = [\overline{A_0}\backslash\overline{s_0'}]\cdots[\overline{A_{i-1}}\backslash\overline{s_{i-1}'}][\overline{A_i}\backslash\overline{s_i}]\cdots[\overline{A_n}\backslash\overline{s_n}]$ while maintaining the invariant that $\delta \to_R^* \delta_i$. At step i we rewrite the substitution $[\overline{A_i}\backslash\overline{s_i}]$. First we apply Lemma 4.3.1:

$$\begin{aligned} & t[\overline{A_0}\backslash\overline{s_0'}]\cdots[\overline{A_{i-1}}\backslash\overline{s_{i-1}'}][\overline{A_i}\backslash\overline{s_i}] \\ \to_R^* \quad & t[\overline{A_0}\backslash\overline{s_0'}]\cdots[\overline{A_{i-1}}\backslash\overline{s_{i-1}'}][\overline{A_i}\backslash\overline{s_i'}] \end{aligned}$$

And then we apply Lemma 4.3.2:

$$\begin{aligned} & t[\overline{A_0}\backslash\overline{s_0'}]\cdots[\overline{A_{i-1}}\backslash\overline{s_{i-1}'}][\overline{A_i}\backslash\overline{s_i}][\overline{A_{i+1}}\backslash\overline{s_{i+1}}]\cdots[\overline{A_n}\backslash\overline{s_n}] \\ \to_R^* \quad & t[\overline{A_0}\backslash\overline{s_0'}]\cdots[\overline{A_{i-1}}\backslash\overline{s_{i-1}'}][\overline{A_i}\backslash\overline{s_i'}][\overline{A_{i+1}}\backslash\overline{s_{i+1}}]\cdots[\overline{A_n}\backslash\overline{s_n}] \end{aligned}$$

At the end $\delta_n = \delta'$ and we have $\delta \to_R^* \delta'$. □

We can now prove that rewriting a grammar changes the generated language by rewriting as well:

Theorem 4.3.1. *Let $G = (N, \Sigma, P, A)$ and $G' = (N, \Sigma, P', A)$ be VTRATGs. If $G \to_R^* G'$, then $L(G) \to_R^* L(G')$.*

Proof. Let $A\delta \in L(G)$. Since $G \to_R^* G'$, there exists a derivation δ' in G' such that $\delta \to_R^* \delta'$. By Lemma 4.3.3, hence $A\delta \to_R^* A\delta' \in L(G)$. On the other hand, let $A\delta' \in L(G')$. By a symmetric argument, there exists a δ such that $A\delta \in L(G)$ and $A\delta \to_R^* A\delta'$. Thus $L(G) \to_R^* L(G')$. □

In contrast to this result on rewriting VTRATGs, there is no corresponding result for regular tree grammars. Consider for example the regular tree grammar G with the productions $A \to f(A, A) \mid c \mid d$ and the rewrite relation $\to_R$ given by $R = \{f(x, x) \to d\}$. Then $G \to_R^* G'$ where G' has the productions $A \to c \mid d$, and $L(G') = \{c, d\}$. However, $f(c, d) \in L(G)$ but $f(c, d) \not\mapsto_R^* c$ and $f(c, d) \not\mapsto_R^* d$.

There are important differences: VTRATGs only produce finite languages and we are concerned with the preservation of size, while for regular tree grammars the question is whether the resulting infinite language can be generated at all. However, the results of [40] suggest a correspondence: they show that the language obtained by rewriting a regular tree language can be recognized by a tree automaton with equality and disequality constraints. A VTRATG where every nonterminal vector has length 1 can also be recognized by a tree automaton with equality constraints of similar size.

4.3.2 Stable terms

The set of terms L that we would like to cover often has some regularity. We will make use of this regularity to simplify grammars using a rewrite relation $\sqsubseteq_L$ derived from L. This rewrite relation consists of all the transformations that keep every subterm of L intact. We then obtain a characterization of the fully simplified grammars—those are called stable grammars.

Example 4.3.1. The following set of terms L has some obvious regularities: the first and second arguments of f are always the same, and in addition the third argument is always e:

$$L = \{f(c, c, e), f(d, d, e)\}$$

The VTRATG G given by the following productions covers L without making

use of these regularities:

$$A \to f(B, C, D)$$
$$B \to c \mid d$$
$$C \to c \mid d$$
$$D \to e$$

Clearly the right hand side of the production $A \to f(B, C, D)$ is unnecessarily general, we could have simplified it to $f(B, B, e)$ (thus saving 2 nonterminals and 3 productions).

Definition 4.3.4. Let L be a set of terms, and t, t' be any terms of the same type. Then $t \sqsubseteq_L t'$ iff $t\sigma \in \mathrm{st}(L)$ implies $t\sigma = t'\sigma$ for all substitutions σ. We also define the strict relation $\sqsubset_L$ by: $t \sqsubset_L t'$ iff $t \sqsubseteq_L t'$ and $t' \not\sqsubseteq_L t$.

Example 4.3.2. For $L = \{f(c, c, e), f(d, d, e)\}$, we have $f(x, y, e) \sqsubseteq_L f(x, x, e)$—this expresses the fact that the first two arguments to f are always equal in L. But also less meaningful statements such as $c \sqsubseteq_L c$ are true, or even $f(f(x, y, e), z, e) \sqsubseteq_L c$. It is *not* the case that $f(x, y, e) \sqsubseteq_L f(c, c, e)$, as $\sigma = [x\backslash d, y\backslash d]$ is a counterexample: we have $f(x, y, e)\sigma = f(d, d, e) \in \mathrm{st}(L)$, but $f(x, y, e)\sigma = f(d, d, e) \neq f(c, c, e) = f(c, c, e)\sigma$.

It follows via routine arguments that $\sqsubseteq_L$ is a preorder and $\sqsubset_L$ a strict partial order.

Lemma 4.3.4. *Let L be a set of terms, then the relation $\sqsubseteq_L$ is a rewrite relation.*

Proof. Let s and t be such that $s \sqsubseteq_L t$. For monotonicity, we need to show that $u[s]_p \sqsubseteq_L u[t]_p$ for all u and p. So let σ be such that $u[s]_p\sigma \in \mathrm{st}(L)$. Then clearly $s\sigma \in \mathrm{st}(L)$ as well, we have $s\sigma = t\sigma$, and thus $u[s]_p\sigma = u[t]_p\sigma$.

To show that $\sqsubseteq_L$ is fully invariant, we need to show that $s\sigma \sqsubseteq_L t\sigma$ for all σ. So let τ be such that $(s\sigma)\tau \in \mathrm{st}(L)$, then clearly $s(\sigma\tau) \in \mathrm{st}(L)$ and $s\sigma\tau = t\sigma\tau$ by assumption. □

Let $\to \in A \times A$ be a binary relation on a set A. An element t is called a normal form with respect to the relation $\to$ if there is no s such that $t \to s$, i.e., if it cannot be reduced using $\to$. The relation $\to$ is called weakly normalizing

if for every $a \in A$ there is a normal form $a' \in A$ such that $a \to^* a'$, where $\to^*$ denotes the reflexive-transitive closure of $\to$. Simplifying a production corresponds to taking normal forms under $\sqsubset_L$.

Definition 4.3.5. A term t is said to be *stable* if t is in normal form with respect to $\sqsubset_L$. The set $S(L)$ consists of all stable terms (with respect to L).

For a stable term t there exists no term s such that $t \sqsubset_L s$.

Example 4.3.3. Let $L = \{f(c,c,e), f(d,d,e)\}$, then the terms $f(z,z,e)$ and c are stable, but $f(x,y,e)$ is not.

Example 4.3.4. We can now simplify the grammar from Example 4.3.1: there we had the production $A \to f(B,C,D)$. If we apply $f(x,y,e) \sqsubseteq_L f(x,x,e)$ to the right hand side of this production, we obtain the production $A \to f(B,B,e)$, as promised.

Lemma 4.3.5. *Let L be a set of terms and $t \in S(L)$. Then t subsumes a subterm of L.*

Proof. Let τ be the type of t. If L contains no subterm of type τ, then $s \sqsubseteq_L r$ for any terms s and r of type τ (since $s\sigma \notin \mathrm{st}(L)$ for any substitution σ). Hence $t \notin s(L)$, a contradiction. So let $t_0 \in \mathrm{st}(L)$ be of type τ. Assume towards a contradiction that t subsumes no subterm of L. In particular $t \not\trianglelefteq t_0$ and hence $t \neq t_0$. Since $t\sigma \notin \mathrm{st}(L)$ for any substitution σ, we have $t \sqsubseteq_L z$, where z is a fresh variable of type τ. We also have $z \not\sqsubseteq_L t$, since $t_0 = z[z\backslash t_0] \neq t[z\backslash t_0] = t$ even though $z[z\backslash t_0] = t_0 \in \mathrm{st}(L)$. Hence $t \sqsubset_L z$ and $t \notin S(L)$. □

Example 4.3.5. Let $L = \{h(f(c,c,e)), h(f(d,d,e))\}$. Then $f(z,z,e) \in S(L)$ subsumes $f(c,c,e) \trianglelefteq h(f(c,c,e)) \in L$, and $g(c) \notin S(L)$ does not subsume any subterm of L.

We will use the relation $\sqsubseteq_L$ to simplify VTRATGs that cover L. Recall that when rewriting a grammar, the generated language is rewritten using the same relation by Theorem 4.3.1. But if we rewrite using $\sqsubseteq_L$, then any term in L remains unchanged:

Lemma 4.3.6. *Let L be a set of terms, then $L \subseteq S(L)$.*

Proof. Let $t \in L$, we need to show that $t \in S(L)$ as well. Assume towards a contradiction that $t \sqsubset_L s$ for some s, and let σ be the identity substitution. Then $t\sigma = t \in L \subseteq \mathrm{st}(L)$ and hence $t = t\sigma = s\sigma = s$, a contradiction. □

We will now show that $\sqsubset_L$ is weakly normalizing, from which we can then conclude that we can rewrite every term into a stable term.

Lemma 4.3.7. *Let L be a set of terms and $t \sqsubseteq_L s$. If t subsumes a subterm of L, then s subsumes that subterm as well, and* $\mathrm{FV}(t) \supseteq \mathrm{FV}(s)$.

Proof. Let σ be a substitution such that $t\sigma = r \in \mathrm{st}(L)$. By definition of $t \sqsubseteq_L s$, we then have $t\sigma = s\sigma = r$. For the second property, assume towards a contradiction that $x \in \mathrm{FV}(s) \setminus \mathrm{FV}(t)$. Let $z_1 \neq z_2$ be two distinct variables of the same type as x. Consider the substitutions $\sigma_i = [x \backslash z_i]\sigma$ for $i \in \{1, 2\}$. We have $t\sigma_1 = t\sigma_2 = r \in \mathrm{st}(L)$ but $s\sigma_1 \neq s\sigma_2$, and hence $t\sigma_1 \neq s\sigma_1$ or $t\sigma_2 \neq s\sigma_2$, in contradiction to $t \sqsubseteq_L s$. □

Theorem 4.3.2. *Let $L \neq \emptyset$ be a set of terms, then $\sqsubset_L$ is weakly normalizing.*

Proof. Let t be a term. We need to show that there exists a term s such that $t \sqsubseteq_L s$ and $s \not\sqsubset_L r$ for any term r. If t does not subsume a subterm of L, then we have $t \sqsubseteq_L t'$ for any $t' \in L \subseteq S(L) \neq \emptyset$. Hence assume without loss of generality that t subsumes a subterm $r \in \mathrm{st}(L)$. If t is a normal form of $\sqsubset_L$, then we are done. Otherwise by Lemma 4.3.7, for any term s such that $t \sqsubset_L s$, it is the case that s subsumes r as well, and $\mathrm{FV}(s) \subseteq \mathrm{FV}(t)$. Since there are only finitely many such terms by Lemma 4.1.1, and $\sqsubset_L$ is transitive and irreflexive, at least one such term s is a normal form of t. □

The following crucial result guarantees that we can rewrite grammars into a form that we can effectively search for—namely those grammars where all productions are stable. Hence we need to show that we can rewrite every term into a stable term:

Corollary 4.3.1. *Let $L \neq \emptyset$ be a set of terms, and t a term. Then there exists a term $t' \in S(L)$ such that $t \sqsubseteq_L t'$.*

Proof. Take any $\sqsubset_L$-normal form of t. □

4.3.3 Stable grammars

Being able to rewrite grammars allows us to transform any grammar until all right-hand sides of productions are stable or, seen differently, to transform any grammar into a subgrammar of the grammar consisting of all stable productions—without increasing its size. The resulting subgrammar still covers the input term set: by Theorem 4.3.1, the language is rewritten with $\sqsubseteq_L$, but by Lemma 4.3.6, $\sqsubseteq_L$ keeps every $t \in L$ unchanged!

Definition 4.3.6. Let L be a set of ground terms, and $N = (A_0, \ldots, \overline{A_n})$ a sequence of nonterminal vectors. Then the stable grammar $S_{N,L} = (N, \Sigma, P, A_0)$ contains all productions with stable right hand sides:

$$P = \{\overline{A_i} \to \overline{s} \mid s_j \in S(L) \text{ for all } j \wedge \mathrm{FV}(\overline{s}) \subseteq \{\overline{A}_{i+1}, \ldots, \overline{A}_n\}\}$$

If L is finite, then $S_{N,L}$ has only finitely many productions: by Lemma 4.3.5, all right-hand sides of productions in $S_{N,L}$ subsume subterms of L, and there are only finitely many such terms in $\mathcal{T}(\Sigma \cup N)$ by Lemma 4.1.1. Hence $S_{N,L}$ is a well-defined and finite grammar. In Corollary 4.3.3, we will even see that $S_{N,L}$ is only polynomially larger than L for fixed N.

Definition 4.3.7. Let $G_1 = (N, \Sigma, P, A)$ and $G_2 = (N', \Sigma', P', A')$ be VTRATGs. G_1 is a subgrammar of G_2, written $G_1 \subseteq G_2$, if $P \subseteq P'$, $A = A'$, $N = N'$, and $\Sigma = \Sigma'$.

We can now prove the main result about the stable grammar $S_{N,L}$:

Theorem 4.3.3. *Let $G = (N, \Sigma, P, A)$ be a VTRATG, and L a set of terms such that $L \subseteq L(G)$. Then there exists a VTRATG $G' = (N, \Sigma, P', A)$ such that:*

1. $G \sqsubseteq_L G'$
2. $G' \subseteq S_{N,L}$
3. $|G'| \leq |G|$
4. $L \subseteq L(G')$

Proof. Let P' be the set of productions that is obtained by $\sqsubseteq_L$-rewriting the right hand side of each production in P to a stable term, this is possible by Corollary 4.3.1. We have $G \sqsubseteq_L G'$ and $G' \subseteq S_{N,L}$. Since this rewriting does not increase the number of productions, we also have $|G'| \leq |G|$. Let $t \in L \subseteq L(G)$. By Theorem 4.3.1, there is a $t' \in L(G')$ such that $t \sqsubseteq_L t'$. But t is in $L \subseteq S(L)$, so $t' = t$. Hence $t \in L(G')$, and therefore $L \subseteq L(G')$. □

Corollary 4.3.2. *Let L be a finite set of terms, and N a sequence of nonterminal vectors. Then the VTRATG $S_{N,L}$ contains a subgrammar $H \subseteq S_{N,L}$ of minimal size covering L.*

4.3.4 Computing all stable terms

In Section 4.3.5, we will minimize the grammar $S_{N,L}$ in order to produce a solution for Parameterized Language Cover. Hence we need to compute $S_{N,L}$. Let $N = (A_0, \overline{A_1}, \ldots, \overline{A_n})$ be a sequence of nonterminal vectors, and $k_0, \ldots, k_n$ be the lengths of these vectors; then the right-hand side of a production in $S_{N,L}$ may contain up to $k_1 + \cdots + k_n$ different nonterminals.

The right hand sides of the productions in $S_{N,L}$ are therefore included in the subset of $S(L)$ of terms with at most $k_1 + \cdots + k_n$ variables (treating nonterminals as variables). In this section, we will show how to compute this subset from L. To this end, we will characterize stable terms as generalizations of least general generalizations with injective matching. Substitutions are injective iff we cannot express one variable in terms of the others:

Lemma 4.3.8. *Let σ be a substitution. Then σ is injective on $\mathcal{T}(\Sigma \cup X)$ iff $u\sigma \neq x\sigma$ for all variables $x \in X$ and terms u such that $\mathrm{FV}(u) \subseteq X \setminus \{x\}$.*

Proof. If there are u and x such that $u\sigma = x\sigma$, then clearly σ is not injective. For the converse, we prove that $t\sigma = s\sigma$ implies $t = s$ by induction on t: If t is a variable, we distinguish three cases: the first case is $t \notin \mathrm{FV}(s)$, here we have a contradiction to the assumption. The second (trivial) case is $t = s$. In the third case both $t \neq s$ and $t \in \mathrm{FV}(s)$, then t is a strict subterm of s, a contradiction to $t\sigma = s\sigma$. If s is a variable, a symmetric argument applies.

If now t and s are both functions, we have two cases: first, if t and s share the same root symbol, then $t|_i\sigma = s|_i\sigma$ for all arguments, hence $t|_i = s|_i$ by

the inductive hypothesis, and $t = s$ by congruence. Second, if t and s have different root symbols, then already $t\sigma = s\sigma$ is a contradiction. □

With this definitions, we can now proceed to characterize the stable terms: in Theorem 4.3.4, we will show that $t \in S(L)$ if and only if the matching to $\mathrm{lgg}(L_t)$ is injective on $\mathcal{T}(\Sigma \cup \mathrm{FV}(t))$, where L_t is the set of subterms of L subsumed by t. We will then prove an even stronger result: it suffices to only consider bounded subsets of L_t, where the bound only depends on the number of variables in t. Since there are only polynomially many such bounded subsets, we will be able to effectively use this characterization to compute the stable terms with a bounded number of variables in Theorem 4.3.6.

Theorem 4.3.4. *Let L be a set of ground terms, t a term that subsumes a subterm of L. Define $L_t := \{s \in L \mid s \preceq t\}$. If $\mathrm{FV}(t) \cap \mathrm{FV}(\mathrm{lgg}(L_t)) = \emptyset$, then $t \in S(L)$ if and only if $\sigma = \mathrm{lgg}(L_t)/t$ is injective on $\mathcal{T}(\Sigma \cup \mathrm{FV}(t))$.*

Proof. First, we show that $t \in S(L)$, assuming that σ is injective. Let s be any term such that $t \sqsubseteq_L s$. For every $r \in L_t$ let $\iota_r := r/\mathrm{lgg}(L)$. We have $t\sigma\iota_r = r \in \mathrm{st}(L)$, and $t\sigma\iota_r = s\sigma\iota_r$ by $t \sqsubseteq_L s$. By Lemma 4.1.2 we obtain $t\sigma = s\sigma$, hence $t = s$ by injectivity of σ, and thus $t \not\sqsubset_L s$ and $t \in S(L)$.

In order to show that σ is injective, we apply Lemma 4.3.8, and have x, u such that $x\sigma = u\sigma$ and $x \in \mathrm{FV}(t) \setminus \mathrm{FV}(u)$. First, we have $t[x\backslash u] \not\sqsubseteq_L t$ because of Lemma 4.3.7. We will now show that $t \sqsubseteq_L t[x\backslash u]$, which contradicts $t \in S(L)$ since $t[x\backslash u] \not\sqsubseteq_L t$. Let τ be a substitution such that $t\tau \in \mathrm{st}(L)$, then $t\tau \in L_t$ as well and there exists a substitution ρ such that $\tau = \sigma\rho$. We can now compute $t[x\backslash u]\tau = t[x\backslash u]\sigma\rho = t\sigma\rho = t\tau$, where $[x\backslash u]\sigma = \sigma$ because the variables in u and the domain of σ are disjoint. □

Theorem 4.3.5. *Let L be a set of ground terms, and $t \in S(L)$. Then there exists a subset $L' \subseteq L_t$ such that $\sigma' = \mathrm{lgg}(L')/t$ is injective on $\mathcal{T}(\Sigma \cup \mathrm{FV}(t))$ and $|L'| \leq |\mathrm{FV}(t)| + 1$.*

Proof. We construct L' in stages: we will define a sequence of $L_0 \subseteq L_1 \subseteq \cdots \subseteq L_n = L'$ such that $|L_i| \leq i{+}1$. In each step we have a substitution $\sigma_k = \mathrm{lgg}(L_k)/t$, in the end $\sigma' = \sigma_n$. First, pick a term $s_0 \in L_t$ and set $L_0 = \{s_0\}$, then we have $t\sigma_0 = s_0$. We can now order the variables $x_1, \ldots, x_n$ in t in such a way that $x_i\sigma_0 \lhd x_j\sigma_0$ implies $i < j$.

By Lemma 4.3.8, it will suffice to show that $x_i\sigma' \neq r\sigma'$ for any x_i and r such that $x_i \notin \mathrm{FV}(r)$. Note that due to symmetry, we can assume without loss of generality that if $r = x_j$ is a variable as well, then $j < i$. Furthermore, if the disequality $x_i\sigma_k \neq r\sigma_k$ already holds for some σ_k, then it also holds for σ' since $\sigma_k = \sigma'(\mathrm{lgg}(L_k)/\mathrm{lgg}(L'))$.

In step i we now ensure that that there are is no r such that $x_i\sigma_i = r\sigma_i$ and $x_i \notin \mathrm{FV}(r)$, and $j < i$ in the case that $r = x_j$ is a variable. Assume that there is such an r with $x_i\sigma_{i-1} = r\sigma_{i-1}$ (otherwise we can set $L_i = L_{i-1}$). With these restrictions, we can show that this r is unique: let $r' \neq r$ be another such term, then we have $r\sigma_{i-1} = r'\sigma_{i-1}$. Similar to Lemma 4.3.8, we can assume that at least one of r or r' is a variable. If $r = x_j$ and $r' = x_k$ are both variables, then without loss of generality $j < k < i$ and hence $x_k\sigma_{i-1} = x_j\sigma_{i-1}$ is a contradiction to the inductive hypothesis. If r is a function and $r' = x_j$ is a variable (or vice versa), then $x_j\sigma_{i-1} = r\sigma_{i-1}$ is also a contradiction.

Now there is a unique r such that $x_i\sigma_{i-1} = r\sigma_i$ with the restrictions above. We have $t \not\sqsubseteq_L t[x_i\backslash r]$ because of $t \in S(L)$ and Lemma 4.3.7. Hence there exists a substitution τ such that $t\tau \in \mathrm{st}(L)$ and $t\tau \neq t[x_i\backslash r]\tau$. Furthermore $t\tau \in L_t$ and $x_i\tau \neq r\tau$. Set $L_i = L_{i-1} \cup \{t\tau\}$. Now $\tau = \sigma_i(t\tau/\mathrm{lgg}(L_i))$ and hence $x_i\sigma_i \neq r\sigma_i$. □

Example 4.3.6. Consider $L = \{f(g(a), g(a), e), f(g(b), g(b), e), f(g(c), g(c), e)\}$. The term $t = f(x, x, e)$ is in $S(L)$, and hence the substitution $\sigma = [x\backslash g(y)]$ is injective, where $t\sigma = \mathrm{lgg}(L_t) = f(g(y), g(y), e)$. But since t has only 1 free variable, there is a subset $L' \subseteq L_t$ with the same property and at most 1+1 elements: for example $L' = \{f(g(c), g(c), e), f(g(d), g(d), e)\}$. The ground term $t_2 = f(g(c), g(c), e)$ has 0 variables, hence there is a subset $L'_2 \subseteq L$ with at most 0 + 1 elements such $t_2\sigma'_2 = \mathrm{lgg}(L'_2)$. This set L'_2 is necessarily the singleton $L'_2 = \{t_2\}$.

Our strategy for computing stable terms will be to compute all terms t such that σ is injective, where $t\sigma = L'$ for some subset $L' \subseteq \mathrm{st}(L)$. We enumerate all subsets of $\mathrm{st}(L')$ of the bounded size given by Theorem 4.3.5, and then infer the stable term t from L' by generalization. However, in general there exists more than one term t that has an injective substitution σ satisfying $t\sigma = \mathrm{lgg}(L')$: for example, with $L' = \{f(f(c)), f(f(d))\}$, all of the terms x, $f(x)$, and $f(f(x))$

have an injective substitution to $\mathrm{lgg}(L')$.

Let $m\colon \mathcal{T}(X \cup \Sigma) \to X$ be a type-preserving partial function from terms to variables. Then $R_m\colon \mathcal{T}(X \cup \Sigma) \to \mathcal{T}(X \cup \Sigma)$ denotes the replacement function that replaces all occurrences of the terms in the domain of m by the corresponding variables.

Example 4.3.7. Recall that the terms x, $f(x)$, and $f(f(x))$ are all in $S(L)$ where $L = \{f(f(c)), f(f(d))\}$. Let $m_1 = \{f(f(y))) \mapsto x\}$, $m_2 = \{f(y) \mapsto x\}$, and $m_3 = \{y \mapsto x\}$. If we consider the least general generalization $\mathrm{lgg}(L) = \mathrm{lgg}\{f(f(c)), f(f(d))\} = f(f(y))$, then we can obtain the three stable terms from it using R_{m_i}: $R_{m_1}(\mathrm{lgg}(L)) = x$, $R_{m_2}(\mathrm{lgg}(L)) = f(x)$, and $R_{m_3}(\mathrm{lgg}(L)) = f(f(x))$.

If a partial function m is injective, its inverse $m^{-1} = \{(s \mapsto t) \mid (t \mapsto s) \in m\}$ is a partial function as well. If $m : X \to Y$ is a partial function, and $Z \subseteq X$ is a subset of its domain, then $m \restriction Z = \{(s \mapsto t) \mid (s \mapsto t) \in m \wedge s \in Z\}$ is the restriction of m to Z.

Since terms are equivalent modulo variable renaming in the subsumption order, in the following lemmas we will assume without loss of generality that the variables in the least general generalization $\mathrm{lgg}(L)$ are distinct from the variables in X. The following lemma now shows how least general generalizations and stable terms relate:

Lemma 4.3.9. *Let $k \in \mathcal{T}(\Sigma \cup X)$, and $k\sigma = \mathrm{lgg}(L)$, where σ is injective on $\mathcal{T}(\Sigma \cup X)$. Then $k = R_{(\sigma \restriction X)^{-1}}(\mathrm{lgg}(L))$.*

Proof. Follows homomorphically from $R_{(\sigma \restriction \mathrm{st}(k))^{-1}}(r\sigma) = r$ for all $r \in \mathrm{st}(k) \cap X$. □

Example 4.3.8. Let $L = \{f(g(c)), f(g(d))\}$, $\mathrm{lgg}(L) = f(g(y))$, and $k = f(x)$. Then $k\sigma = \mathrm{lgg}(L)$ for $\sigma = [x \backslash g(y)]$ and indeed $R_{\{g(y) \mapsto x\}}(f(g(y))) = f(x)$.

We now have enough constraints on stable terms to enumerate all of them (with a given bound on the number of variables) in polynomial time by simply applying all possible replacements to all least general generalizations. However, this would also produce many terms that are not stable if we use replacements that correspond to non-injective substitutions. For example,

take $L = \{f(c,e), f(d,e)\}$, $k = f(x,e) = \mathrm{lgg}(L) \preceq L$, and $X = \{y,z\}$. Using the replacement $m = (\sigma \restriction X)^{-1} = \{x \mapsto y, e \mapsto z\}$ we can obtain the term $R_m(f(x,e)) = f(y,z)$, which is not in $S(L)$. Hence it is important to check that the substitutions are injective:

Lemma 4.3.10. *Let σ be a substitution and X a set of variables. Then we can decide in polynomial time whether σ is injective on $\mathcal{T}(\Sigma \cup X)$.*

Proof. We define a binary predicate s on variables and terms:

$$s(x,t) :\leftrightarrow \exists u \in \mathcal{T}(\Sigma \cup X \setminus \{x\}),\ u\sigma = t$$

This predicate can be computed recursively:

$$s(x,t) \leftrightarrow \begin{cases} \top & \text{if } y\sigma = t \text{ for some } y \neq x \\ s(x,t_1) \wedge \cdots \wedge s(x,t_n) & \text{if } t = f(t_1,\ldots,t_n) \\ \bot & \text{otherwise} \end{cases}$$

By Lemma 4.3.8, σ is injective iff $\forall x \in X\ \neg s(x, x\sigma)$. The runtime of s is quadratic in the size of t and X, and we iterate it for every $x \in X$, hence it has polynomial runtime in X and σ. □

We can now give the algorithm that computes the stable terms, and prove its correctness. The runtime of the algorithm depends on the symbolic complexity $|L|_s$ of the set of terms L: the symbolic complexity is the sum of the number of positions of each term in L.

Theorem 4.3.6. *Let L be a set of ground terms and X a set of variables, then the set of all terms $k \in S(L)$ such that $\mathrm{FV}(k) \subseteq X$ can be computed as follows:*

1. *For each subset $L' \subseteq \mathrm{st}(L)$ such that $|L'| \leq |X| + 1$:*
2. *Compute $\mathrm{lgg}(L')$.*
3. *For each injective partial function $m\colon \mathrm{st}(\mathrm{lgg}(L')) \to X$:*
4. *Check that m^{-1} is injective on $\mathcal{T}(\Sigma \cup X)$.*
5. *Then output $k = R_m(\mathrm{lgg}(L'))$.*

The runtime of this procedure is bounded by a polynomial in $|L|_s$ for fixed $|X|$.

Proof. Let us first show the correctness of the algorithm, i.e., that it does indeed generate exactly the terms $k \in S(L)$ such that $\mathrm{FV}(k) \subseteq X$. Assume that k is such a term. Then by Theorem 4.3.4, there exists an injective substitution σ such that $k\sigma = \mathrm{lgg}(L_k)$. By Theorem 4.3.5, we then have an $L' \subseteq L_k$ such that $|L'| \leq |k|{+}1 \leq |X|{+}1$ and the substitution σ' such that $k\sigma' = \mathrm{lgg}(L')$ is injective as well. By Lemma 4.3.9, the injective partial function $m := (\sigma \restriction \mathrm{st}(k))^{-1}$ satisfies $k = R_m(\mathrm{lgg}(L'))$. Hence the algorithm outputs k. On the other hand, every term in the output is in $S(L)$ by Theorem 4.3.4.

In step (1), there are at most $\binom{\mathrm{st}(L)}{|X|+1} = O(|L|_s^{|X|+1})$ possible subsets; in step (3), there are at most $|L|_s$ positions in $\mathrm{lgg}(L')$ and hence at most $O(|L|_s^{|X|+1})$ partial functions σ. Computing least general generalizations and replacements is linear in the input, checking injectivity is also polynomial due to Lemma 4.3.10, therefore the total runtime is polynomial in $|L|_s$. □

Corollary 4.3.3. *Let a sequence of nonterminals N be fixed. Then the grammar $S_{N,L}$ is polynomial-time computable from a finite set of ground terms L.*

Proof. Apply Theorem 4.3.6 to $X = N$. □

4.3.5 Minimization

In Corollary 4.3.3, we have produced a polynomial-time computable VTRATG $S_{N,L}$ that is guaranteed to contain a subgrammar H covering L of minimal size. In particular, this subgrammar H solves the Parameterized Language Cover for L and N. Since we can efficiently compute $S_{N,L}$, we have reduced Parameterized Language Cover to VTRATG-Minimization.

We will reduce this problem to MaxSAT by giving a propositional formula expressing the property that the subgrammar H covers L. MaxSAT is an optimization variant of the Boolean satisfaction problem (SAT), for which a number of efficient off-the-shelf solvers exist, see the yearly MaxSAT competition [5] for a list of solvers. We only consider the partial and unweighted variant of MaxSAT, and simply call it MaxSAT:

Problem 4.3.1 (MaxSAT)**.**
Given two sets of propositional clauses H and S (so-called "hard" and "soft" clauses), find an interpretation I such that $I \models H$ and I maximizes the number of satisfied clauses in S.

We will encode "H covers L" by stating for each $t \in L$ that "there exists a derivation δ_t of t in H". The concrete encoding of "δ_t is a derivation of t in H" is based on a so-called sparse encoding of the function $B \mapsto B\delta_t$ (for nonterminals B), i.e., encoding the function as a binary relation. One important observation about this function is that it usually returns only subterms of t. For example consider the following derivation:

$$\delta = [A \backslash f(B, B)]\,[B \backslash d]$$

In this case, $t = A\delta = f(d, d)$, and $B\delta = d$ is indeed a subterm of t. However this subterm property can fail in the presence of "unused" nonterminals, for example:

$$\delta_2 = [A \backslash f(B, B)]\,[(B, C) \backslash (d, c)]$$

Again $t = A\delta_2 = f(d, d)$ and $B\delta_2 = d$ are subterms, but now $C\delta_2 = c$ is not a subterm of $f(d, d)$. If $B\delta$ is not a subterm of t, it will turn out to be irrelevant to the derivation, hence we will ignore it and assign the dummy term $\bot$ to all such terms which are not subterms of t. This allows us to consider the smaller range $\mathrm{st}(t) \cup \{\bot\}$ for the function δ.

Definition 4.3.8 (Propositional encoding for $t \in L(H)$)**.** Let $G = (N, \Sigma, P, A)$ be a VTRATG, and t a ground term. We use the following atoms:

- $B\delta_t = r$ where B is a nonterminal and $r \in \mathrm{st}(t) \cup \{\bot\}$ a ground term.
- $p \in P'$ where $p \in P$ is a production.

We define the following abbreviations for formulas:

$$\overline{B}\delta_t = \overline{r} \equiv \bigwedge_j B_j\delta_t = r_j$$

$$\mathbf{Match}_{G,t}(u, s) \equiv \begin{cases} \top & \text{if } u = \bot \\ \bigwedge_{(C,r) \in u/s} C\delta_t = r & \text{if } s \preceq u \\ \bot & \text{otherwise} \end{cases}$$

$$\mathbf{Case}_{G,t}(\overline{B},\overline{u}) \equiv \overline{B}\delta_t = \overline{u} \rightarrow \bigvee_{\overline{B}\rightarrow\overline{s}\in P}\left(\overline{B}\rightarrow\overline{s}\in P' \wedge \bigwedge_j \mathbf{Match}_{G,t}(u_j,s_j)\right)$$

$$\mathbf{Func}_{G,t} \equiv \left(\bigwedge_{B \text{ nonterminal}} \bigvee_{s\in \mathrm{st}(t)\cup\{\bot\}} B\delta_t = s \right.$$
$$\left. \wedge \bigwedge_{B \text{ nonterminal}} \bigwedge_{s_1\neq s_2\in \mathrm{st}(t)\cup\{\bot\}} \neg(B\delta_t = s_1 \wedge B\delta_t = s_2)\right)$$

$$\mathbf{GenTerm}_{G,t} \equiv A\delta_t = t \wedge \mathbf{Func}_{G,t} \wedge \bigwedge_{\overline{B}\in N} \bigwedge_{\overline{s}\in(\mathrm{st}(L)\cup\bot)^{|\overline{B}|}} \mathbf{Case}_{G,t}(i,\overline{s})$$

The formula $\mathbf{Match}_{G,t}(u,s)$ encodes $s\delta_t = u$, extending the $B\delta_t = u$ atom to arbitrary terms s. In order to ensure the correctness of the whole encoding, we have to add implied constraints for each possible function value of δ_t and for each nonterminal vector $\overline{B}$: the $\mathbf{Case}_{G,t}(\overline{B},\overline{u})$ formula encodes these constraints. Then $\mathbf{Func}_{G,t}$ states that $\delta\colon N \rightarrow \mathrm{st}(t)\cup\{\bot\}$ is a function, and $\mathbf{GenTerm}_{G,t}$ combines the other formulas to encode that δ is a derivation of t using only the productions from the subset $P'\subseteq P$, i.e., those which are present in H.

Example 4.3.9. Consider $t = f(c,c,e)$, $N = \{A,B\}$, and $G = (N,\Sigma,P,A)$ with the following productions $P = \{p_1,p_2,p_3,p_4\}$:

$$p_1 = A \rightarrow f(B,B,B) \qquad p_3 = B \rightarrow c$$
$$p_2 = A \rightarrow f(B,B,e) \qquad p_4 = B \rightarrow d$$

Then we encode $t\in L(H)$ using the following formula $\mathbf{GenTerm}_{G,t}$ (slightly simplified propositionally):

$$\begin{aligned}
\mathbf{Case}_{G,t}(A,f(c,c,e)) &\equiv A\delta_t = f(c,c,e) \rightarrow (p_2\in P' \wedge B\delta_t = c)\\
\mathbf{Case}_{G,t}(A,c) &\equiv A\delta_t = c \rightarrow \bot\\
\mathbf{Case}_{G,t}(A,e) &\equiv A\delta_t = e \rightarrow \bot\\
\mathbf{Case}_{G,t}(A,\bot) &\equiv A\delta_t = \bot \rightarrow \top\\
\mathbf{Case}_{G,t}(B,f(c,c,e)) &\equiv B\delta_t = f(c,c,e) \rightarrow \bot\\
\mathbf{Case}_{G,t}(B,c) &\equiv B\delta_t = c \rightarrow (p_3\in P')\\
\mathbf{Case}_{G,t}(B,e) &\equiv B\delta_t = e \rightarrow \bot
\end{aligned}$$

$$\begin{aligned}\mathbf{Case}_{G,t}(B,\bot) &\equiv B\delta_t = \bot \to \top \\ \mathbf{GenTerm}_{G,t} &\equiv A\delta_t = f(c,c,e) \wedge \mathbf{Func}_{G,t} \wedge \bigwedge_{C,s} \mathbf{Case}_{G,t}(C,s)\end{aligned}$$

Many of the formulas $\mathbf{Case}_{G,t}(C,s)$ are of the form $(\cdots \to \bot)$; these correspond to derivations that are impossible: for example we have $A\delta_t = c \to \bot$ because there is no production of A that has c as the root symbol. In this example, $\mathbf{GenTerm}_{G,t}$ entails $p_2 \in P' \wedge p_3 \in P'$, hence any covering subgrammar $H \subseteq G$ necessarily includes these two productions; the minimal covering subgrammar consists precisely of these two productions.

Lemma 4.3.11. *Let $G = (N, \Sigma, P, A)$ be a VTRATG, $H = (N, \Sigma, P', A) \subseteq G$ a subgrammar of G, and t a term. Then for every derivation δ of t in H, there exists a satisfying interpretation $I \models \mathbf{GenTerm}_{G,t}$ such that for all $p \in P$, $I \models p \in P'$ iff $p \in P'$.*

Conversely, if $I \models \mathbf{GenTerm}_{G,t}$ is a satisfying interpretation such that for all $p \in P$, $I \models p \in P'$ iff $p \in P'$, then there exists a derivation of t in H.

Proof. We need to construct a satisfying interpretation I for every derivation δ in H. So we would like to set $I \models B\delta_t = r$ if and only if $B\delta = r$; but this could fail if r is not a subterm of t. But the following assignment only results in $\bot$ or subterms of t, as required:

$$I \models B\delta_t = r \quad \Leftrightarrow \quad \begin{cases} r = B\delta & \text{if } B \text{ is a subterm of } t \\ r = \bot & \text{otherwise} \end{cases}$$

It remains to verify that $I \models \mathbf{Case}_{G,t}(\overline{B}, \overline{s})$ for all i and $\overline{s}$, that is, $I \models \mathbf{Match}_{G,t}(B_j\delta_t, s_j)$ for all j, where $\overline{B} \to \overline{s}$ is the chosen production in δ. This is trivial if $B_j\delta$ is not a subterm of t—then $\mathbf{Match}_{G,t}(B_j\delta_t, s_j)$ expands to $\top$—in the other case, it is clear from the definition of a derivation.

Conversely, let I be a satisfying interpretation for $\mathbf{GenTerm}_{G,t}$ such that $I \models p \in P'$ iff $p \in P'$. We will now construct an actual derivation δ in H such that $A\delta = t$. By $\mathbf{Func}_{G,t}$ there is a unique value for each $B\delta_t$ in I, let $f(B) \in \mathrm{st}(t) \cup \{\bot\}$ be the term such that $I \models B\delta_t = f(B)$. Hence $I \models \mathbf{Case}_{G,t}(\overline{B}, (f(B_1), \ldots, f(B_{|\overline{B}|})))$, and we can choose the production $\overline{B} \to$

$(s_{B_1}, \ldots, s_{B_{|\overline{B}|}})$ for δ such that $I \models \overline{B} \rightarrow (s_{B_1}, \ldots, s_{B_{|\overline{B}|}}) \wedge \bigwedge_j \mathbf{Match}_{G,t}(f(B_j), s_{B_j})$. This immediately ensures that δ really is a derivation in the subgrammar.

We will now verify that $f(B) \neq \bot$ implies $B\delta = f(B)$ for all B, by backwards induction on the index of B in the nonterminal vector N: if $f(B) \neq \bot$, then $\mathbf{Match}_{G,t}(f(B), s_B) \equiv \bigwedge_l C_l\delta = r_l$. Because the nonterminals in s_B can only be of the form C where C occurs after B in N, computing $B\delta$ first substitutes B with s_B, and then each C_l with r_l, since $C_l\delta = r_l$ per induction hypothesis. Now since $f(A) = t \neq \bot$, we conclude that δ is a derivation of $A\delta = t$. □

So far we have only considered the encoding for the derivability of a single term t, we will now turn to derive multiple terms.

Theorem 4.3.7. *VTRATG-Minimization $\leq_P$ MaxSAT.*

Proof. Let $G = (N, \Sigma, P, A)$ be the VTRATG, and $L \subseteq L(G)$ the set of terms. We need to find a subset $P' \subseteq P$ of minimal size such that the grammar $H = (N, \Sigma, P', A)$ still satisfies $L \subseteq L(H)$. By Lemma 4.3.11, any interpretation I satisfying $\bigwedge_{t \in L} \mathbf{GenTerm}_{G,t}$ corresponds to a covering VTRATG. Set the hard clauses of the MaxSAT problem to a CNF of this formula. (Such a CNF can be obtained in polynomial time by a Tseitin transformation [97].)

For each production $p \in P$, we add the soft clause $\neg(p \in P')$. The number of satisfied soft clauses is then exactly $|P| - |P'|$, the number of productions not included in the minimized VTRATG. Maximizing the number of soft clauses then minimizes the number of productions in the VTRATG given by the interpretation. From a solution I to this MaxSAT problem, we obtain the minimal covering VTRATG by setting $P' = \{p \mid I \models p \in P'\}$, which is the corresponding subgrammar according to Lemma 4.3.11. □

4.4 Induction grammars

In Section 4.3, we developed the theory for stable grammars underlying the MaxSAT algorithm in a very general way. We can now apply this theory to induction grammars as well, and thereby extend the MaxSAT algorithm to solve Parameterized Indexed Termset Cover. First, we will show that language generation and rewriting commutes:

Algorithm 2 Solution of the Parameterized Language Cover Problem via MaxSAT

Input: set of ground terms L, and a sequence of nonterminals N.
Output: minimal VTRATG H with nonterminals N such that $L \subseteq L(H)$.
G = stableGrammar(N, L)
φ = minimizationFormula(G, L)
I = maxSatSolver(φ, softClauses(G))
H = grammarFromAssignment(I)

Definition 4.4.1. Let $G = (\tau, \alpha, (\overline{v}_c)_c, \overline{\gamma}, P)$ and $G' = (\tau, \alpha, (\overline{v}_c)_c, \overline{\gamma}, P')$ be induction grammars, and $\rightarrow_R$ a rewrite relation. Then $G \rightarrow_R^* G'$ iff for all $p' \in P'$ there exists a $p \in P$ such that $p \rightarrow_R^* p'$ and for all $p \in P$ there exists a $p' \in P'$ such that $p \rightarrow_R^* p'$.

Lemma 4.4.1. *Let $\rightarrow_R$ be a rewrite relation, and G, G' be induction grammars such that $G \rightarrow_R^* G'$. Then $I(G,t) \rightarrow_R^* I(G',t)$ and $L(G,t) \rightarrow_R^* L(G',t)$ for all t.*

Proof. We have $I(G,t) \rightarrow_R^* I(G',t)$ since all productions in the instance grammar are substitution instances of productions in G and G', and $\rightarrow_R$ is a rewrite relation. Commutation of language generation and rewriting $L(G,t) \rightarrow_R^* L(G',t)$ directly lifts from the result for VTRATGs in Theorem 4.3.1. □

Similar to Theorem 4.3.3, we can now show that we can transform grammars into subgrammars of the stable grammar without increase in size.

Definition 4.4.2. Let $(L_i)_{i \in I}$ be a family of languages, and $\alpha, (\overline{v_c})_c, \overline{\gamma}$ as in Definition 2.8.5. Then the *stable grammar* $S((L_i)_{i \in I}) = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$ contains all possible productions where the right-hand side is a stable term of the appropriate type, that is:

$$P = \{\overline{\kappa} \rightarrow \overline{t}[\alpha, \overline{v_i}, \overline{\gamma}] \mid \overline{\kappa} \in \{\tau, \overline{\gamma}\} \wedge \overline{t}[\alpha, \overline{v_i}, \overline{\gamma}] \in S(\bigcup_i L_i)\}$$

Definition 4.4.3. Let $G = (\tau, \alpha, (\overline{v}_c)_c, \overline{\gamma}, P)$ and $G' = (\tau, \alpha, (\overline{v}_c)_c, \overline{\gamma}, P')$ be induction grammars. We say that G' is a subgrammar of G, written $G' \subseteq G$, iff $P' \subseteq P$.

Theorem 4.4.1. *Let $(L_i)_{i\in I}$ be a finite family of languages, and $G = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$ an induction grammar that covers $(L_i)_{i\in I}$. Then there exists an induction grammar $G' = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P')$ such that:*

- *G' covers $(L_i)_{i\in I}$*
- *$|G'| \leq |G|$*
- *$G' \subseteq S((L_i)_{i\in I})$*

Proof. We obtain the induction grammar G' by replacing the right-hand side of every production in G by one of its $\sqsubset_L$-normal forms, which are in $S((L_i)_{i\in I})$. The resulting induction grammar G' still covers $(L_i)_{i\in I}$ by applying Lemma 4.4.1 to $\sqsubseteq_L$ since we have $G \sqsubseteq_L^* G'$. □

We can now turn to the problem of induction grammar minimization, analogous to VTRATG-Minimization. For this part we can reuse the MaxSAT encoding of Definition 4.3.8, defining the minimization of induction grammar in terms of the instance grammars:

Definition 4.4.4. Let $(L_i)_{i\in I}$ be a finite family of languages, and $G = (\tau, \alpha, (\overline{v_c})_c, \overline{\gamma}, P)$ an induction grammar. We define the propositional formula $\mathbf{Gen}(G, (L_i)_{i\in I})$ as the following formula:

$$\bigwedge_{i\in I}\bigwedge_{t\in L_i} \mathbf{GenTerm}'(I(G,i),t) \quad \wedge \quad \bigwedge_{i\in I}\bigwedge_{p'}\left(p' \in P'_{I(G,i)} \rightarrow \bigvee_{p \rightsquigarrow_i p'} p \in P\right)$$

The formula $\mathbf{GenTerm}'$ is a syntactical variant of **GenTerm**, which differs only in the names of the propositional variables:

- $p' \in P'_{I(G,i)}$ instead of $p' \in P'$
- $\delta_{t,I(G,i)}$ instead of δ_t

Theorem 4.4.2. *Let $(L_i)_{i\in I}$ be a finite family of languages, G an induction grammar, and $G' \subseteq G$ a subgrammar. Then the formula $\mathbf{Gen}(G, (L_i)_{i\in I}) \wedge \bigwedge\{\neg p \in P' \mid p \in G \setminus G'\}$ is satisfiable if and only if G' covers $(L_i)_{i\in I}$.*

Proof. By Lemma 4.3.11, the formula is satisfiable iff for all $i \in I$, the instance grammar $I(G', i)$ covers L_i. The second conjunct of Definition 4.4.4 then ensures that whenever an instantiated production is used to cover a term in the instance grammar, it is already included in the induction grammar. □

Algorithm 3 Cover a family of languages by an induction grammar

```
procedure FindGrammar(σ, τ̄, (L_i)_i)
    S ← S((L_i)_i)
    hard ← Gen(S, (L_i)_i)
    soft ← {¬p ∈ P' | p ∈ S}
    if Sat(I) ← MaxSAT(hard, soft) then
        return {p | I ⊨ p ∈ P'}
    else
        fail
    end if
end procedure
```

Lemma 4.4.2. *Let $(L_i)_{i\in I}$ be a finite family of languages, then Algorithm 3 produces an induction grammar covering $(L_i)_{i\in I}$ of minimal size, or fails if there is no covering grammar.*

Proof. First, assume that there exists a covering induction grammar; by Theorem 4.4.1 there is a grammar G covering $(L_i)_{i\in I}$ of minimal size such that $G \subseteq S((L_i)_{i\in I})$. By Theorem 4.4.2, the formula $\mathtt{hard} \wedge \bigwedge_{p\in \mathtt{S}\setminus \mathtt{G}} \neg(p \in P')$ is satisfiable. Hence the interpretation I returned by the MaxSAT solver sets at most $|G|$ many $p \in P'$ atoms to true, corresponding to a grammar with at most $|G|$ productions. If there is no covering grammar, then MaxSAT will return unsatisfiable as return status, and Algorithm 3 will fail. □

4.5 Reforest

4.5.1 TreeRePair

Given the close relation of the problem of finding small covering VTRATGs and grammar-based text- and document compression, we can also try to adapt

algorithms from grammar-based compression. In this section we will describe such an adaptation of the TreeRePair [65] algorithm to produce covering VTRATGs. TreeRePair compresses XML documents (formally, ranked and unranked trees, i.e. single-sorted first-order terms with function symbols of variable arity) using context-free tree grammars.

Definition 4.5.1. A *context-free tree grammar* $G = (N, \Sigma, P, A)$ is a tuple consisting of a start symbol $A \in N$ of arity 0, a set of function symbols N as nonterminals (whose arity is not necessarily 0), and a set of productions P. A production is a pair $B(\overline{x}) \to t$ of two terms of the same type such that $\overline{x}$ is a vector of pairwise distinct variables and $\mathrm{FV}(t) \subseteq \overline{x}$. The grammar G is *acyclic* iff the relation $\prec$ on nonterminals, defined by $B \prec C$ iff there exists a production $B(\overline{x}) \to t \in P$ such that t contains C, is acyclic. The grammar G is *straight-line* iff for every nonterminal $B \in N$, there is exactly one production of the form $B(\dots) \to \dots$ in P.

Definition 4.5.2. Let $G = (N, \Sigma, P, A)$ be a context-free tree grammar. The *single-step derivation relation* $\Rightarrow_G$ is a binary relation on terms defined by $t \Rightarrow_G t_p[r[\overline{x}\backslash\overline{s}]]$ iff $t_p = B(\overline{s})$ and $B(\overline{x}) \to r \in P$ for some position p. A term $t \in \mathcal{T}(\Sigma)$ is derivable in G iff $A \Rightarrow_G^* t$. The *language* $L(G) = \{t \in \mathcal{T}(\Sigma) \mid A \Rightarrow_G^* t\}$ consists of all derivable terms.

The context-free tree grammars produced by TreeRePair are acyclic and straight-line. Such grammars produce exactly a single term (the original XML document). The central concept behind the algorithm are digrams: a digram is a triple (f, i, g) consisting of two function symbols f, g and a natural number i. This digram describes the pattern $f(x_1, \dots, x_{i-1}, g(\overline{y}), z_{i+1}, \dots, z_n)$, i.e., an occurrence of f where the i-th argument has g as root symbol. A digram can be compressed by first introducing a new nonterminal B with the production $B(\overline{x}, \overline{y}, \overline{z}) \to f(\overline{x}, g(\overline{y}), \overline{z})$, and then replacing all occurrences $f(\overline{t}, g(\overline{s}), \overline{r})$ by $B(\overline{t}, \overline{s}, \overline{r})$. Given a term t to compress, TreeRePair starts with the trivial grammar with the single production $A \to t$ and then repeatedly compresses the most frequent digram by adding a new production as described before. (There is also an additional post-processing step called "pruning" that we do not show here, and which does not apply in the example below.)

Example 4.5.1. TreeRePair compresses the term $f^8(c)$ in the following steps:

$P_0 = \{A \to f^8(c)\}$ (the most frequent digram is $(f, 0, f)$)
$P_1 = \{A \to B^4(c), B(x) \to f(f(c))\}$ (the most frequent digram is $(B, 0, B)$)
$P_2 = \{A \to C^2(c), B(x) \to f(f(c)), C(x) \to B(B(x))\}$

The resulting context-free tree grammar G then has the productions P_2 and satisfies $L(G) = \{f^8(c)\}$.

Looking at the digram compression step again, the generated production has a noteworthy property: it is linear, that is, each variable occurs at most once on the right hand side. In our adaption, we will also generate productions that are non-linear to better capture the rigid behavior of VTRATGs.

4.5.2 Adaptation to tree languages

There are two general big differences from our setting to that of TreeRePair: first, we do not want to compress just a single term but a set of terms. And second, we expect a *lossy* compression that does not reproduce the input term set exactly, but can produce a superset. Similar to the delta-table algorithm, we consider only the introduction of a single nonterminal in the resulting covering VTRATG. That is, given a finite set of terms L we produce a covering VTRATG with the nonterminals $A, \overline{B}$ where the length of $\overline{B}$ is chosen by the algorithm. This adaptation, called Reforest, proceeds in three steps:

1. Produce a trivial context-free tree grammar G_0.
2. Iteratively compress G_0 by introducing abbreviating productions.
3. Use the compressed context-free tree grammar to directly read off the productions for the covering VTRATG.

We then iterate this process to produce a VTRATG with more nonterminals. The first two steps are already present in TreeRePair, albeit in a simpler way. The third step is specific to our adaptation. In the first step we produce the context-free tree grammar $G_0 = \{A \to t \mid t \in L\}$. This grammar is no longer straight-line, since it produces multiple terms: namely $L(G_0) = L$. However the

$$
\begin{aligned}
P_0 &= \{A \to r(c,c) \mid \cdots \mid r(f^7(c), f^7(c))\} \\
P_1 &= \{A \to r(c,c) \mid r(f(c), f(c)) \mid r(B(c), B(c)), \\
&\quad A \to r(f(B(c)), f(B(c))) \mid r(B^2(c), B^2(c)) \mid r(f(B^2(c)), f(B^2(c))), \\
&\quad A \to r(B^3(c), B^3(c)) \mid r(f(B^3(c)), f(B^3(c))), \\
&\quad B(x) \to f(f(x))\} \\
P_2 &= \{A \to r(c,c), \mid r(f(c), f(c)) \mid r(B(c), B(c)), \\
&\quad A \to r(f(B(c)), f(B(c))) \mid r(C(c), C(c)) \mid r(f(C(c)), f(C(c))), \\
&\quad A \to r(B(C(c)), B(C(c))) \mid r(f(B(C(c))), f(B(C(c)))), \\
&\quad B(x) \to f(f(x)), C(x) \to B(B(x))\} \\
P_3 &= \{A \to D(c) \mid D(f(c)) \mid D(B(c)) \mid D(f(B(c))) \\
&\quad A \to D(C(c)) \mid D(f(C(c))) \mid D(B(C(c))) \mid D(f(B(C(c)))), \\
&\quad B(x) \to f(f(x)), C(x) \to B(B(x)), D(x) \to r(x,x)\} \\
P_4 &= \{A \to D(c) \mid E(c) \mid D(B(c)) \mid E(B(c)), \\
&\quad A \to D(C(c)) \mid E(C(c)) \mid D(B(C(c))) \mid E(B(C(c))) \\
&\quad B(x) \to f(f(x)), C(x) \to B(B(x)), D(x) \to r(x,x), E(x) \to D(f(x))\} \\
P_5 &= \{A \to D(c) \mid E(c) \mid D(B(c)) \mid E(B(c)), \\
&\quad A \to D(F) \mid E(F) \mid D(B(F)) \mid E(B(F)), \\
&\quad B(x) \to f(f(x)), C(x) \to B(B(x)), D(x) \to r(x,x), E(x) \to D(f(x)), \\
&\quad F \to C(c)\} \\
P_6 &= \{A \to D(c) \mid E(c) \mid G(c) \mid E(B(c)), \\
&\quad A \to D(F) \mid E(F) \mid G(F) \mid E(B(F)), \\
&\quad B(x) \to f(f(x)), C(x) \to B(B(x)), D(x) \to r(x,x), E(x) \to D(f(x)), \\
&\quad F \to C(c), G(x) \to D(B(x))\} \\
P_7 &= \{A \to D(c) \mid E(c) \mid G(c) \mid H(c), \\
&\quad A \to D(F) \mid E(F) \mid G(F) \mid H(F), \\
&\quad B(x) \to f(f(x)), C(x) \to B(B(x)), D(x) \to r(x,x), E(x) \to D(f(x)), \\
&\quad F \to C(c), G(x) \to D(B(x)), H(x) \to E(B(x))\}
\end{aligned}
$$

Figure 4.1: Run of the first compression step in the Reforest algorithm on $L = \{r(c,c), r(f(c), f(c)), \ldots, r(f^7(c), f^7(c))\}$.

context-free tree grammars that we will produce will be almost straight-line: for any nonterminal $B \neq A$, there will be at most one production $B(\dots) \to \dots$.

In the second step, we introduce two kinds of abbreviating productions: digrams (as in TreeRePair), and rigid trigrams. A rigid trigram is a triple (f, i, j) of a function symbol f and two natural numbers i and j. It describes the pattern $f(x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{j-1}, x_i, x_{j+1}, \dots, x_n)$, i.e., an occurrence of f where the i-th and j-th argument are the same term. This rigid trigram can be abbreviated by introducing a production $B(\overline{x}, y, \overline{z}, \overline{w}) \to f(\overline{x}, y, \overline{z}, y, \overline{w})$. We then iteratively compress the most frequent feature, which is either a digram or a rigid trigram.

Example 4.5.2. Let $L = \{r(c, c), r(f(c), f(c)), \dots, r(f^7(c), f^7(c))\}$. Reforest first produces the context-free tree grammar P_7 by abbreviating digrams and rigid trigrams as shown in Section 4.5.2. This compression has introduced new structure into the term set: instead of the "homogeneous" terms $r(f^n(c), f^n(c))$, the productions of the start symbol A are now all combinations of the 6 different symbols $D(\cdot), E(\cdot), G(\cdot), H(\cdot)$ and c, F.

In the third step, we now inspect the compressed context-free tree grammar G_n and use it to construct a covering VTRATG. Since all nonterminals in G_n except A have at most one production, we can define $t \Downarrow_{G_n}$ to be the unique term derivable from t in G_n provided that t does not contain A. We now look at all the productions $A \to f(\overline{t})$ in G_n. If all the function symbols f such that there exists a production $A \to f(\overline{t})$ have the same arity, then we produce a VTRATG with the productions $\{A \to (f(\overline{x}) \Downarrow_{G_n})[\overline{x} \backslash \overline{B}] \mid A \to f(\overline{t})\} \cup \{\overline{B} \to \overline{t} \Downarrow_{G_n} \mid A \to f(\overline{t})\}$. If the function symbols do not have the same arity, we pad them with dummy constants.

Example 4.5.3 (continuing Example 4.5.2). From the A-productions in P_7 we get the function symbols D, E, G, H and the arguments c, F and we produce the VTRATG with the following 6 productions:

$$A \to D(B) \Downarrow \mid E(B) \Downarrow \mid G(B) \Downarrow \mid H(B) \Downarrow$$
$$B \to c \Downarrow \mid F \Downarrow$$

After expanding $\Downarrow$:

$$A \to r(B,B) \mid r(f(B),f(B)) \mid r(f^2(B),f^2(B)) \mid r(f^3(B),f^3(B))$$
$$B \to c \mid f^4(c)$$

4.6 Experimental evaluation

In this chapter we have presented three different algorithms to find small covering grammar. All of these algorithms are implemented in the GAPT framework. We benchmarked their implementation in GAPT 2.2 on two data sets:

1. Synthetic examples bundled with GAPT[1]. These are sequences of proofs parameterized by a natural number and produce sets of terms similar to Example 4.5.2. For example one of these synthetic examples, `LinearExampleProof`(n), constructs a proof of the following sequent:

$$P(0), \forall x\ (P(x) \to P(s(x))) \vdash P(s^n(x))$$

2. Proofs from the TSTP collection produced by automated theorem provers (Thousands of Solutions from Theorem Provers [93]; all proofs from the FOF and CNF categories as of November 2015). From the total 137989 proofs in the TSTP, we could import term sets from 68185 proofs (49.41%).

Both data sets consist of single-sorted languages. We have extracted term sets from the TSTP collection of proofs produced by automated theorem provers (Thousands of Solutions from Theorem Provers [93]; all proofs from the FOF and CNF categories as of November 2015). Unfortunately many of these proofs are in custom formats that we have been unable to import. However from the total 137989 proofs in the TSTP, we could import term sets from 68185 proofs (49.41%). Of these 68185 term sets, 35467 (52.02%) can not be compressed using VTRATGs (with regard to our size measure) as each term has a different root symbol.

[1] These can be found in the `gapt.examples` package.

For each of the data sets, we show two plots: one comparing the runtime of the algorithms when they produce a VTRATG (for Reforest we only count grammars with fewer productions than the size of the term set). The second one compares the runtime in the cases where the cut-introduction algorithm in GAPT could use the grammar to generate a non-trivial lemma [36]. We also show the results for a "virtual best" method which has the minimum runtime of the others, i.e., it behaves as if we ran all the algorithms in parallel in a portfolio mode and terminated as soon as one of them finishes.

We compared the implementation of these three algorithms in a prerelease version[2] of GAPT 2.3 [37] and used GNU parallel [96] for scheduling. The MaxSAT solver we used is OpenWBO version 2.0 [67]. The comparison was conducted on a Debian Linux system with an Intel i5-4570 CPU and 8 GiB RAM, with a time limit of 60 seconds. In the Figures 4.2 and 4.3 we use the following names for the grammar generation algorithms: m`_maxsat` is the MaxSAT algorithm with the parameter $N = \{A, (B_1, \ldots, B_m)\}$, m`_`n`_maxsat` is the MaxSAT algorithm with parameter $N = \{A, (B_1, \ldots, B_m), (C_1, \ldots, C_n)\}$, `many_dtable` is the delta-table algorithm without row-merging, `many_dtable_ss` is the delta-table algorithm with row-merging, `1_dtable` is the delta-table algorithm using lgg_1 instead of lgg.

In an earlier evaluation we compared the delta-table algorithm and the MaxSAT algorithm [29] in GAPT 2.0 on the same proofs from the TSTP. There are surprising differences in the results: the most striking difference concerns the proof import. The GAPT 2.3 release introduced an algorithm that directly converts resolution proofs (as imported from the TSTP) to expansion proofs, without constructing LK proofs as an intermediate step. (We will discuss this algorithm in more detail in Chapter 6.) Using this new and improved algorithm we can import almost twice as many proofs as term sets, increasing the number from 36494 term sets in GAPT 2.0 to 68185 term sets in GAPT 2.3.

Another noteworthy change in GAPT 2.3 is the rewrite of the delta-table implementation that was necessary to fix several bugs. As a side effect, the performance of the delta-table algorithm was significantly improved. While the MaxSAT algorithm was significantly more successful in the previous eval-

[2] git revision `9b9909820744046ff57af9b399965824bf3a4919`

uation [29], in the present evaluation the delta-table algorithm is more competitive now.

Figure 4.2 compares the runtime for the synthetic examples. Here Reforest finds the largest number of grammars by a significant margin. It finds more than 600 compressing VTRATGs, three times as many as the next algorithm, `1_maxsat`, which finds about 200. The line for Reforest also almost coincides with the virtual best: only very few term sets can be compressed by another algorithm but not by Reforest. The next most successful algorithm is the MaxSAT algorithm with the parameter $N = \{(A), (B)\}$. Many of the synthetic problems have short natural proofs with cuts that only have a single universal quantifier, this is most likely why that parameter is the most successful choice.

If we only consider the grammars that could actually be successfully used by cut-introduction to produce a nontrivial lemma, then the gap shortens a bit: Reforest is still ahead by a significant margin, but the margin is smaller. Reforest finds more than 200 useful grammars, and `1_maxsat` finds about 150. There are two explanations for this result: one is that for our proof-theoretic applications in the form of cut-introduction it is not just important to find a compressing VTRATG, but that there are additional qualitative features necessary for a VTRATG to correspond to an interesting proof with cut. The other explanation is that the algorithm used to find the actual formula for the lemma is exponential in the number of nonterminal vectors; and Reforest produces a larger number of nonterminal vectors than the other algorithms.

Within a timeout of 60 seconds per term set, the delta-table algorithm finds grammars for 6873 term sets. Algorithm 2 improves on this and finds grammars for 7503 term sets. This is a significant improvement, since the additional 630 term sets are the most difficult instances solved, even though they only account for a 9% increase in solutions.

The results for the TSTP data set in Figure 4.3 are completely different: here, the delta-table algorithm is the most successful one, tied with the MaxSAT algorithm with parameter $N = \{A, (B_1, B_2)\}$. Reforest is the (second) least successful algorithm both by the number of compressing grammars, and the number of grammars that were useful for cut-introduction. This is probably caused by the different nature of the proofs in the TSTP: only very few of these proofs contain purely repetitive structures such as $\{c, \ldots, f^n(c)\}$, variants of

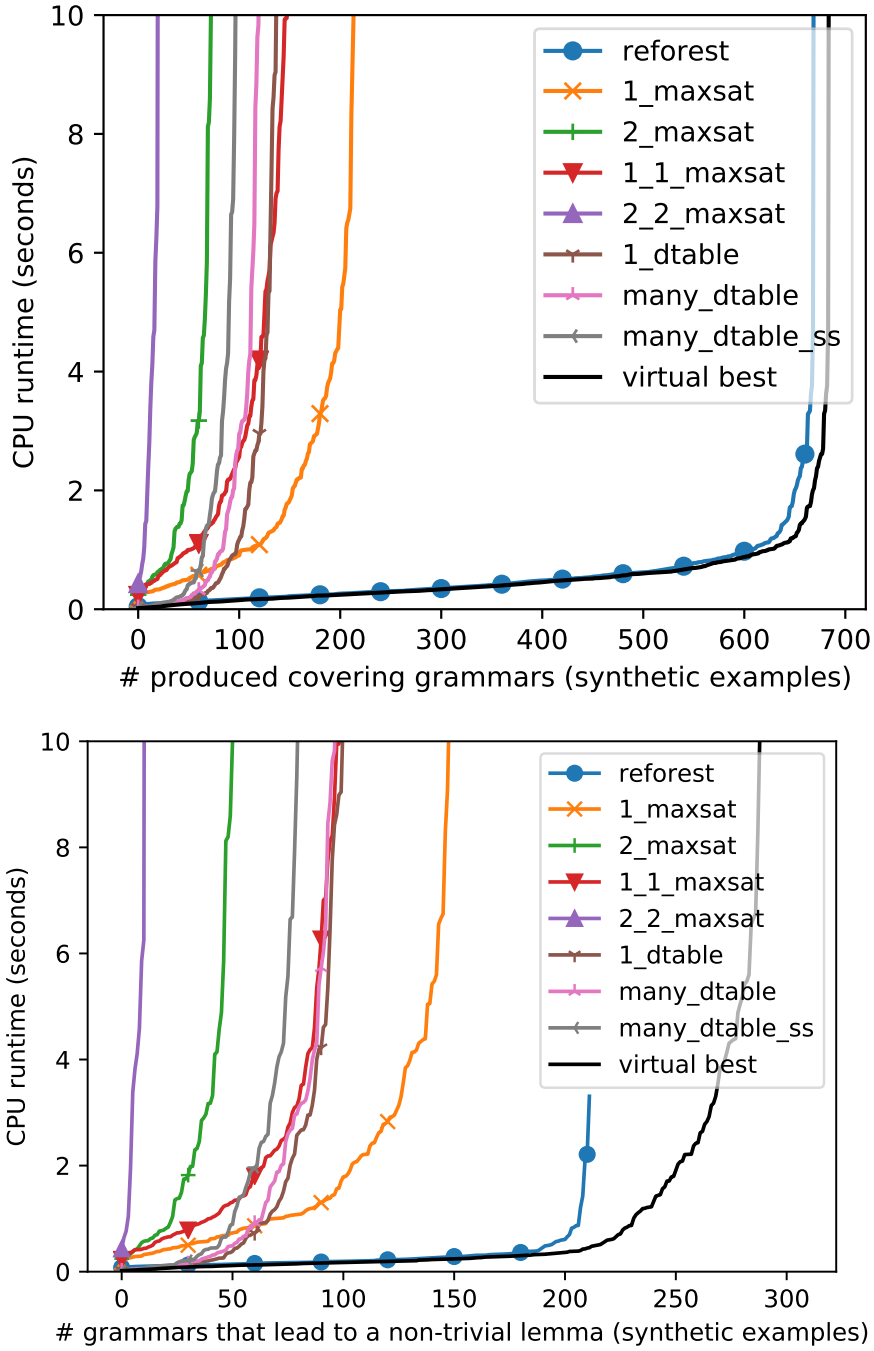


Figure 4.2: Cactus plot of grammar generation runtime on synthetic examples.

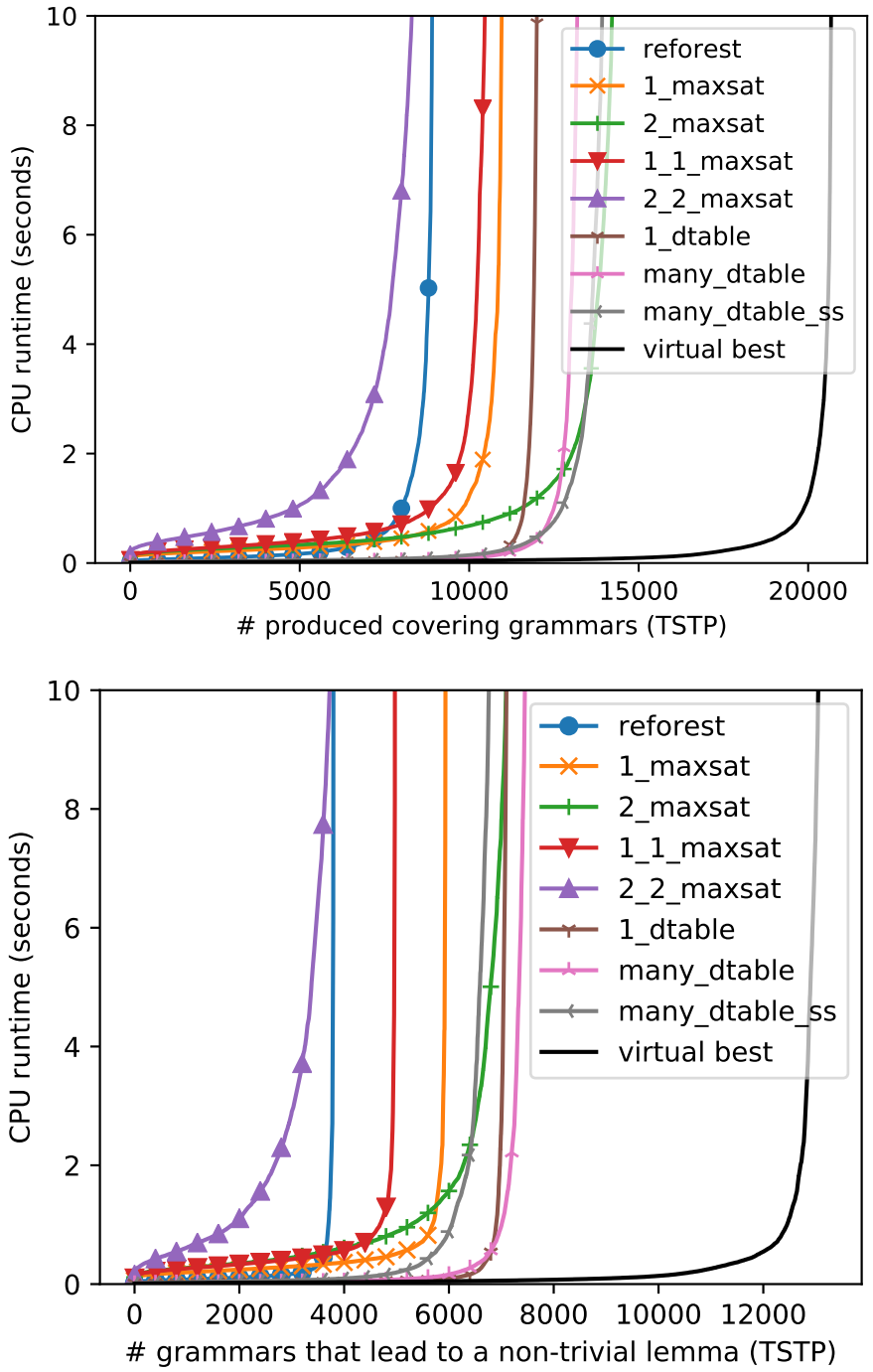


Figure 4.3: Cactus plot of grammar generation runtime on proofs from the TSTP.

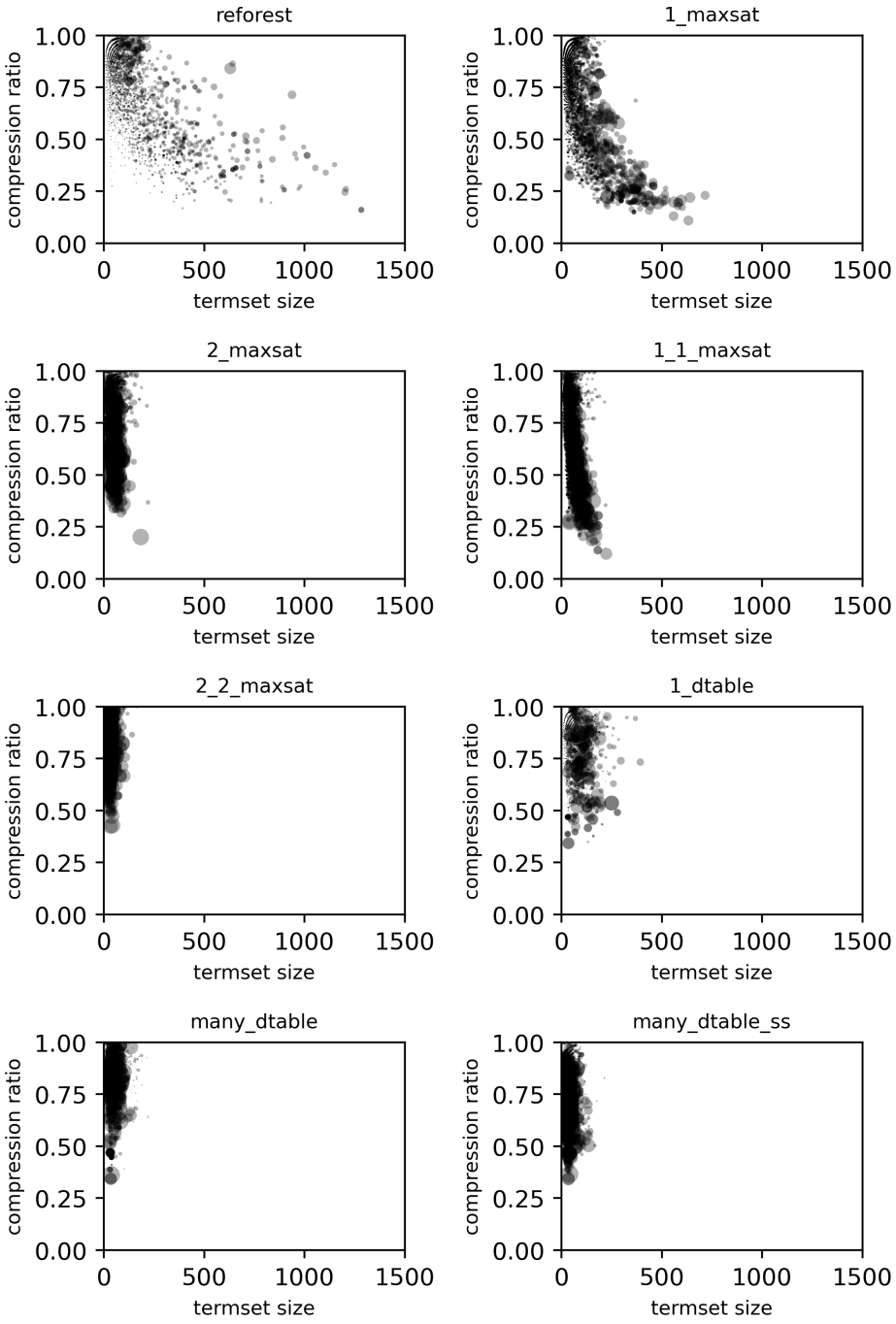


Figure 4.4: Runtime performance comparison on the TSTP proofs. Each point represents an input term set, and its size is proportional to the runtime of each algorithm.

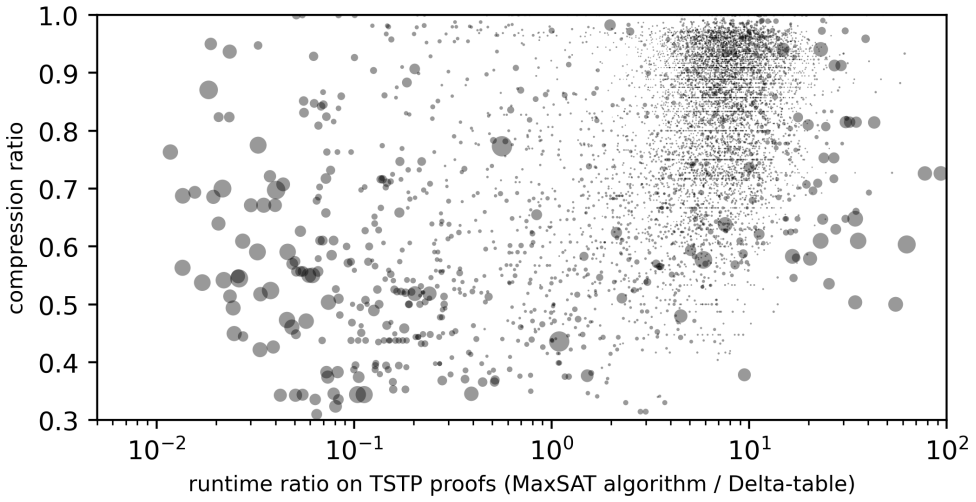


Figure 4.5: Comparison of the performance difference to the compression ratio and overall runtime. Each point represents an input term set from the TSTP data set. The size of a point is proportional to the combined runtime of both algorithms.

which occur in many of the synthetic examples.

For easy term sets—those which take a short time to compress—the delta-table algorithm is much faster than the MaxSAT algorithm; there the curve for the delta-table algorithm is below the one for the MaxSAT algorithm. As far as we can tell, the performance difference on the easy examples is due to the overhead of constructing the stable grammar and the minimization formula. On average, this reduction makes up for 72% of the runtime for term sets where the total runtime is less than 1 second, whereas the situation is almost reversed for term sets where the total runtime is more than 10 seconds: there about 69% of the runtime is spent inside the MaxSAT solver.

This dichotomy between easy and hard problems is not only apparent on the aggregate data set, but also for individual term sets: Figure 4.5 shows the CPU runtime ratio (x-axis) in comparison to the compression ratio (y-axis) and total runtime (size of the points). On the top right there is a cluster of small points: these are the easy problems, and while the delta-table algorithm is

faster by a factor of 10, both algorithms can solve the problems in a reasonable time. Then there is a large cluster on the left: these are the hard problems, which generally admit a better compression, and here the MaxSAT algorithm is faster by a factor of 10.

Figure 4.4 shows a scatter plot of term set size against achieved compression ratio (ratio of grammar size to term set size, i.e., number of productions to number of terms). The size of each point in Figure 4.4 is proportional to the runtime of the algorithm: we see that Reforest is typically much faster than the other algorithms. The reason for the difference between Figures 4.3 and 4.4 is mainly that the TSTP contains many small proofs, for which it is important to find even very small opportunities for compression. Hence for the small term sets it is more advantageous to use algorithmically more complex methods as they can exploit these subtle features in a more effective way and find compressing grammars at all.

Only the MaxSAT algorithm and Reforest can find grammars whose size is less than 25% of the input term set; however Reforest only achieves these compression ratios for larger term sets. For the MaxSAT algorithm, the choice of parameter is crucial: while a larger parameter N would theoretically allow us to find more grammars (and hence lower compression ratios), in practice these larger parameters just increase the runtime and we hence find fewer grammars in the given time limit.

With Reforest we have an interesting algorithm to find covering VTRATGs that uses very different techniques to the delta-table and MaxSAT algorithm. Due to its roots in grammar-based compressed, it is better from an algorithmic perspective: the introduction of a single nonterminal could potentially be done in linear time using the implementation tricks used in TreeRePair [65]. (The Reforest implementation takes quadratic time.) Even iterating this process to introduce multiple nonterminals only has cubic cost. In comparison, both the MaxSAT and delta-table algorithms rely on algorithms for NP-complete optimization problems (MaxSAT and set cover, respectively). The crucial trick that allows Reforest to find covering VTRATGs without relying on backtracking search is the heuristic of most frequent digrams, which allows us to deterministically choose a compression.

However it is not obvious how to generalize Reforest to produce induction

grammars. Digram compression exponentially compresses sequences of the form $\{r(c), \ldots, r(f^{2^n-1}(c))\}$ using the VTRATG $A \rightarrow r(B_2) \mid r(f(B_2)), B_2 \rightarrow B_3 \mid f^{2^2}(B_3), \ldots, B_{n-2} \rightarrow B_{n-1} \mid f^{2^{n-2}}(B_{n-1}), B_{n-1} \rightarrow c \mid f^{2^{n-1}}(c)$. While for induction grammars, we would hope to obtain a production $\tau \rightarrow r(v)$. Since digram compression is central to the algorithm, this behavior seems to be hard to avoid.

5 Formula equations and decidability

In the previous chapters we have studied the grammars of simple proofs and simple induction proofs. The grammars contain only the quantifier inference terms in these proofs. Generating their language corresponds to cut-elimination. However not every grammar is the grammar of a proof. For example, a grammar generating the empty language will not correspond to a proof (since the empty sequent is not derivable). In this chapter we will study the conditions under which grammars can be extended to proofs, and what properties the proposed cut formula and induction formulas need to satisfy. These conditions will be collected in a so-called *formula equation*.

The algorithms devised in Chapter 4 allow us to find grammars that cover a given set of terms. If we start from a cut-free proof π and find a VTRATG G that covers $L(\pi)$, then we know the quantifier inference terms for a small proof with cuts (i.e., a proof with a small quantifier complexity). It only remains to algorithmically find suitable cut formulas, i.e., a solution to the formula equation. This approach of introducing cuts into an existing cut-free proof is called cut-introduction [51, 48, 50, 49, 36].

A procedurally similar approach can also be used for automated inductive theorem proving [31]. There we start from cut-free proofs of instances of the sequent and find a covering induction grammar. In both applications it is crucial to automatically solve the induced formula equations. Hence we will study two algorithms that solve formula equations in this chapter. The first one in Section 5.6 will be based on forgetful inference, while the second one in Section 5.7 will be based on interpolation.

5.1 Formula equations

We have already seen the definition of a formula equation in Section 2.9. An easy observation is that if the formula equation is solvable, then it is valid in higher-order logic, both in Henkin semantics as well as full semantics.

Example 5.1.1. The formula equation $\exists X\, \Psi$ with $\Psi = \forall x\, ((P(x) \to X) \wedge (X \to P(x))$ has no solution (modulo $T = \emptyset$) since Ψ implies $\forall x\, (P(x) \to X)$ and $\forall y\, (X \to P(y))$ and therefore $\forall x \forall y\, (P(x) \to P(y))$, which is not a valid formula. If the quantifier $\forall x$ was not present in the formula equation, it would have the solution $X(x)$. One way to view this situation is that the quantifier $\forall x$ in the formula equation has the effect that x may not occur in the solution.

The concept of formula equations can be traced back to the 19th century, see [102] for an overview. Formula equations are connected to second-order quantifier elimination. In this context, Ackermann proved the following lemma which allows us to eliminate a single second-order quantifier.

Lemma 5.1.1 (Ackermann's lemma [1])**.** *Let Φ be a formula equation such that $\Phi = \exists \overline{X}\, \exists Y\ (\forall \overline{x}\, (\theta(\overline{x}) \to Y(\overline{x})) \wedge \psi)$ where Y does not occur in θ and only occurs negatively in ψ. Then:*

1. *Φ is T-solvable iff $\Phi' = \exists \overline{X}\, \psi[Y \backslash \theta]$ is solvable*
2. *If Φ has the T-solution σ, then Φ' has the T-solution $\sigma \restriction \overline{X}$.*
3. *If Φ' has the T-solution σ then Φ has the T-solution $[Y \backslash \theta]\sigma$.*

Proof. Let us first show $T \vdash \forall \overline{x}\, (\theta(\overline{x}) \to Y(\overline{x})) \wedge \psi \to \psi[Y \backslash \lambda \overline{x}\, \theta(\overline{x})]$. This is true because Y only occurs negatively in ψ. Whenever we can prove $\chi' \to \chi$ for some formulas χ, χ' then we can replace all negative occurrences of χ by χ' in ψ while preserving provability (this is an easy induction on ψ). From this observation we conclude 2.

For 3, assume $T \vdash \psi[Y \backslash \theta]\sigma$. We need to show $T \vdash (\forall \overline{x}\, (\theta(\overline{x}) \to Y(\overline{x})) \wedge \psi)[Y \backslash \theta]\sigma$. By moving $[Y \backslash \theta]\sigma$, this is equivalent to $T \vdash \forall \overline{x}\, (\theta\sigma(\overline{x}) \to \theta\sigma(\overline{x})) \wedge \psi[Y \backslash \theta]\sigma$, which directly follows from the assumption. Finally, 1 follows from 2 and 3. □

We can relax the syntactic restriction in Lemma 5.1.1 a bit to formula equations of the form $\Phi = \exists \overline{X}\ \exists Y\ \forall \overline{x}\ (\theta(\overline{x}) \to Y(\overline{t})) \wedge \psi$ where Y does not occur in θ and only negatively in ψ. This formula equation is equivalent to $\tilde{\Phi} = \exists \overline{X}\ \forall \overline{y}\ (\exists \overline{x}\ (\theta(\overline{x}) \wedge \overline{y} = \overline{t}) \to Y(\overline{y})) \wedge \psi$. Therefore Φ is solvable iff $\exists \overline{X}\ \psi[Y \backslash \lambda \overline{y}\ \exists \overline{x}\ (\theta(\overline{x}) \wedge \overline{y} = \overline{t})]$ is solvable.

Example 5.1.2. Let $\Phi = \exists X\ (\forall x\ (P(x) \to X(x)) \wedge (X(a) \to P(b)))$. Then Φ is solvable iff $P(a) \to P(b)$ is solvable. And $P(a) \to P(b)$ is not $\emptyset$-solvable.

Example 5.1.3. Let $\Phi = \exists X\ (\forall x\ (X(x) \to X(s(x))) \wedge (X(a) \to P(b)))$. Then Lemma 5.1.1 does not apply since $\theta(x) = X(x)$ contains X.

5.2 Solvability of VTRATGs

Let us first study the case of VTRATGs and simple proofs. That is: given a decodable VTRATG G such that every nonempty nonterminal vector has a production, is there a simple proof π such that $G(\pi) = G$?

Remark 5.2.1. Requiring a production of every nonterminal vector is a subtle condition, but does not pose a large restriction in practice: given a VTRATG G such that there is no production for the nonterminal vector $\overline{B}$, we can add a production $\overline{B} \to (c, \ldots, c)$ with dummy constants c. This only slightly increases the size of the grammar. Furthermore the language of the extended grammar G' is a superset of $L(G)$, so if $L(G)$ is tautological, then $L(G')$ is tautological as well.

This situation is of course a small mismatch: we can have a simple proof π such that $G(\pi)$ has a nonterminal vector without productions, and then the formula equation for $G(\pi)$ is not defined—even though we could reasonably expect it to exist, and furthermore have a solution consisting of the cut-formulas in π. The fundamental reason for this mismatch stems from the different treatment of weakening inferences in proofs and VTRATGs. As we have seen in Lemma 2.7.5 and Theorem 3.9.1, all steps in cut-elimination except for the ones for weakening preserve the language of the grammar exactly. Only the steps for weakening can make the language smaller. The case that a nonterminal vector has no productions corresponds to a cut where the cut formula is completely introduced by a weakening on the weak side. During

cut-elimination, all quantifier instances on the strong side are hence discarded. However on the level of the grammar we essentially only discard those terms which contain nonterminals. The language of a grammar explicitly excludes terms with nonterminals—recall Definition 2.6.6.

There are other approaches as well. For example, we could make the function $G(\cdot)$ partial, and only define $G(\pi)$ if π does not introduce cut formulas using weakening. This would break the correspondence between instance grammars and grammars of instance proofs in Theorem 2.8.1: the instance grammar of an induction grammar can introduce (some or all) cut formulas using weakening, so $G(I(\pi, t))$ we be undefined. Another choice would be to preprocess the simple proofs by permuting weakening inferences downward, assigning to every simple proof π a proof π' where no cut-formula is introduced using weakening. This breaks Theorem 2.8.1 in a different way: now $G(I(\pi, t)) \simeq G(I(\pi, t)')$ may have fewer nonterminal vectors than $I(G(\pi), t)$. Eliminating these weakening inferences from the proof with induction directly is in general not possible. Yet another option is to change how the language of a grammar is defined, by iteratively pruning nonterminal vectors without productions and productions that reference these nonterminals. This complicated definition would be highly unnatural from the point of view of formal languages.

Let the nonterminals of such a VTRATG G be $A \prec \overline{B_1} \prec \ldots \overline{B_n}$. Then a proof π with $G(\pi) = G$ has n cuts; in the general case these cuts are nested in a linear way as shown in Figure 5.1.

The sequent $\Gamma_i \vdash \Delta_i$ consists of the instances of $\Gamma \vdash \Delta$ (corresponding to the productions from the nonterminal A) that only contain eigenvariables from $\overline{\alpha_{i+1}}, \ldots, \overline{\alpha_n}$. That is, if $A \to s$ is a production in G and s only contains nonterminals from $\overline{B_{i+1}}, \ldots, \overline{B_n}$, then $\Gamma_i \vdash \Delta_i$ contains the corresponding formula instance. Similarly, the terms $\overline{t_{i,j}}$ range over all right-hand sides of productions of $\overline{B_i}$.

The formula equation Φ_G in the following Definition 5.2.1 is just the conjunction of all the leaves in Figure 5.1. If this formula equation is solvable, then there is a simple proof π with $G(\pi) = G$ (namely the one in Figure 5.1). And also the other way around, as we will see in Theorem 5.2.1: if $L(G)$ is

$$\dfrac{\dfrac{\dfrac{\Gamma_0 \vdash \Delta_0, \varphi_n(\overline{\alpha_n}), \ldots, \varphi_2(\overline{\alpha_2}), \varphi_1(\overline{\alpha_1})}{\Gamma \vdash \Delta, \varphi_n(\overline{\alpha_n}), \ldots, \varphi_2(\overline{\alpha_2}), \varphi_1(\overline{\alpha_1})}}{\Gamma \vdash \Delta, \varphi_n(\overline{\alpha_n}), \ldots, \varphi_2(\overline{\alpha_2}), \forall\overline{x_1}\, \varphi_1(\overline{x_1})} \qquad \dfrac{\Gamma_1, \varphi_1(\overline{t_{1,1}}), \cdots \vdash \Delta_1, \varphi_n(\overline{\alpha_n}), \ldots, \varphi_2(\overline{\alpha_2})}{\Gamma, \forall\overline{x_1}\, \varphi_1(\overline{x_1}) \vdash \Delta, \varphi_n(\overline{\alpha_n}), \ldots, \varphi_2(\overline{\alpha_2})}}{\Gamma \vdash \Delta, \varphi_n(\overline{\alpha_n}), \ldots, \varphi_2(\overline{\alpha_2})}$$

$$\vdots$$

$$\dfrac{\dfrac{\Gamma \vdash \Delta, \varphi_n(\overline{\alpha_n})}{\Gamma \vdash \Delta, \forall\overline{x_n}\, \varphi_n(\overline{x_n})} \qquad \dfrac{\Gamma_n, \varphi_n(\overline{t_{n,1}}), \cdots \vdash \Delta_n}{\Gamma, \forall\overline{x_n}\, \varphi_n(\overline{x_n}) \vdash \Delta}}{\Gamma \vdash \Delta}$$

Figure 5.1: General case of a simple proof with n cuts corresponding to a VTRATG with $n + 1$ nonterminal vectors.

tautological, then the formula equation is solvable.

Definition 5.2.1. Let $G = (N, \Sigma, A, P)$ be a decodable VTRATG with $N = \{A \prec \overline{B_1} \prec \ldots \overline{B_n}\}$. We define the formula equation $\Phi_G = \exists\overline{X}\, \Psi_G$, where Ψ_G is the conjunction of the following formulas:

- $\forall\overline{\alpha_1} \ldots \forall\overline{\alpha_n}\big(\bigwedge \Gamma_0 \to \bigvee \Delta_0 \vee X_n(\overline{\alpha_n}) \vee \cdots \vee X_1(\overline{\alpha_n}, \ldots, \overline{\alpha_1})\big)$
- $\forall\overline{\alpha_{i+1}} \ldots \forall\overline{\alpha_n}\big(\bigwedge \Gamma_i \wedge \bigwedge_{\overline{t} \in P_{\overline{B_i}}} X_i(\overline{\alpha_n}, \ldots, \overline{\alpha_{i+1}}, \overline{t}) \to \bigvee \Delta_i \vee X_n(\overline{\alpha_n}) \vee \cdots \vee X_{i+1}(\overline{\alpha_n}, \ldots, \overline{\alpha_{i+1}})\big)$ for $1 \leq i \leq n$

where:

- $P_{\overline{C}} = \{\overline{t} \mid \overline{C} \to \overline{t} \in P\}$ for every nonterminal vector $\overline{C} \in N$.
- $T_i = \{t \in P_A \mid \mathrm{FV}(t) \subseteq \bigcup \overline{B_{i+1}} \cup \cdots \cup \bigcup \overline{B_n}\}$ for $0 \leq i \leq n$.
- $\Gamma_i \vdash \Delta_i$ is the sequent consisting of all formula instances corresponding to terms in T_i.

The formula equation for a VTRATG is directly of the form required by an iterated application of Lemma 5.1.1: if we eliminate $X_1, \ldots, X_n$ in this order, then there is always exactly one positive occurrence of X_i in the i-th iteration, and the other negative occurrences are in other implications. What we obtain in this way is the so-called "canonical solution". Whenever the formula equation

is solvable or even just valid, this canonical solution will solve the formula equation:

Definition 5.2.2. Let $G = (N, \Sigma, A, P)$ be a decodable VTRATG with $N = \{A \prec \overline{B_1} \prec \ldots \overline{B_n}\}$. Then the canonical solution $\overline{C^G} = [X_1 \backslash C_1^G, \ldots, X_n \backslash C_n^G]$ is defined as follows:

$$C_1^G(\overline{\alpha_n}, \ldots, \overline{\alpha_1}) = \bigwedge \Gamma_0 \wedge \neg \bigvee \Delta_0$$
$$C_{i+1}^G(\overline{\alpha_n}, \ldots, \overline{\alpha_{i+1}}) = \left(\bigwedge \Gamma_i \wedge \neg \bigvee \Delta_i\right) \wedge \bigwedge_{\overline{t} \in P_{\overline{B_i}}} C_i(\overline{\alpha_n}, \ldots, \overline{\alpha_{i+1}}, \overline{t})$$

Lemma 5.2.1. *Let G be a decodable VTRATG such that there is a production of every nonempty nonterminal vector. Then $\bigwedge L(G) \leftrightarrow \Psi_G[\overline{X} \backslash \overline{C^G}]$.*

Proof. By iterated application of Lemma 5.1.1. □

In a sense, the canonical solution C_i^G is the conjunction of all instances of the end-sequent that are available in the proof at the i-th cut. We could hope that this means that the canonical solution is maximal, that it implies every other solution. However this is not the case:

Example 5.2.1. For a VTRATG G, the canonical solution is not necessarily maximal in the sense that it implies any other solution to the FE Φ_G. Consider for example the sequent:

$$\forall x\, P(x), \forall x\, (Q(x) \to Q(s(x))) \vdash Q(0) \to Q(s(s(0)))$$

And the following VTRATG G:

$$A \to r_1(B) \mid r_2(C) \mid r_3$$
$$B \to C$$
$$C \to 0 \mid s(0)$$

This VTRATG induces the formula equation Φ_G:

$$\begin{aligned}\exists X \exists Y \big(&\forall \beta\, \forall \gamma\, (P(\beta) \wedge (Q(\gamma) \to Q(s(\gamma))) \to \\ &\qquad (Q(0) \to Q(s(s(0)))) \vee Y(\gamma) \vee X(\gamma, \beta)) \\ &\wedge \forall \gamma\, ((Q(\gamma) \to Q(s(\gamma))) \wedge X(\gamma, \gamma) \to (Q(0) \to Q(s(s(0)))) \vee Y(\gamma)) \\ &\wedge (Y(0) \wedge Y(s(0)) \to (Q(0) \to Q(s(s(0)))))\big)\end{aligned}$$

Then $[X\backslash\lambda\gamma\lambda\beta\ \bot, Y\backslash\lambda\gamma\ (Q(\gamma) \to Q(s(\gamma)))]$ is a solution for Φ_G. But $\bot$ is not implied by $C_1^G(\gamma,\beta) = (P(\beta) \wedge (Q(\gamma) \to Q(s(\gamma))) \wedge \neg(Q(0) \to Q(s(s(0)))))$.

For VTRATGs, all reasonable ways to define the solvability of the associated formula equation are equivalent:

Theorem 5.2.1. *Let G be a decodable VTRATG such that there is a production of every nonempty nonterminal vector, then the following are equivalent:*

1. *There exists a simple proof π such that $G(\pi) = G$*
2. *$L(G)$ is T-tautological*
3. *$\overline{C^G}$ is a T-solution for Φ_G*
4. *Φ_G has a quantifier-free T-solution*
5. *Φ_G is T-solvable*
6. *$T \models \Phi_G$ in Henkin semantics*
7. *$T \models \Phi_G$ in full higher-order semantics*

Proof. We have $1 \Rightarrow 2$ by Corollary 2.7.1, and $2 \Rightarrow 3$ by Lemma 5.2.1. Since $\overline{C^G}$ is quantifier-free we also have $3 \Rightarrow 4 \Rightarrow 5$. As we have a concrete formula for the solution, the solution also exists in all Henkin models (and a fortiori also in all full higher-order models), hence $5 \Rightarrow 6 \Rightarrow 7$. Since the equivalence $L(G) \leftrightarrow \Psi_G[\overline{X}\backslash\overline{C^G}]$ of Lemma 5.2.1 is also valid in Henkin semantics, we get $7 \Rightarrow 2$. Furthermore Φ_G consists of the initial sequents of a proof π with $G(\pi) = G$, so $3 \Rightarrow 1$. □

Corollary 5.2.1. *Let G be a decodable VTRATG such that every nonempty nonterminal vector has a production. Then it is decidable whether G is $\emptyset$-solvable or not.*

5.3 Solvability of induction grammars

Similar to Definition 5.2.1 for VTRATGs, we can also collect the conditions necessary to construct a simple induction proof with a given induction grammar into a formula equation:

Definition 5.3.1. Let $G = (\tau, \alpha, (\overline{v}_c)_c, \overline{\gamma}, P)$ be an induction grammar. We define the following sets where $\overline{\kappa} \in \{\tau, \overline{\gamma}\}$, i is the index of a constructor, and j is c or an index of a constructor:

- $P^i_{\overline{\kappa}} = \{\overline{t} \mid \overline{\kappa} \to \overline{t} \in P \wedge \mathrm{FV}(\overline{t}) \subseteq \{\alpha\} \cup \overline{v}_{c_i} \cup \overline{\gamma}\}$
- $P^c_{\overline{\kappa}} = \{\overline{t} \mid \overline{\kappa} \to \overline{t} \in P \wedge \mathrm{FV}(\overline{t}) \subseteq \{\alpha\}\}$
- $\Gamma_j = P^j_\tau$
- $T_j = P^j_{\overline{\gamma}}$ if $P^j_{\overline{\gamma}} \neq \emptyset$, otherwise $T_j = \{\overline{\gamma}\}$

Definition 5.3.2. Let G be a decodable induction grammar for a simple induction problem $\Gamma \vdash \forall x\, \varphi(x)$, then the corresponding FE is defined as $\Phi_G = \exists X\, \Psi_G$ where Ψ_G is the conjunction of the following formulas:

1. $\forall \overline{v_{c_i}} \forall \overline{\gamma}\ \left(\bigwedge \Gamma_i \wedge \bigwedge_l \bigwedge_{\overline{t} \in T_i} X(\alpha, v_{c_i,l}, \overline{t}) \to X(\alpha, c_i(\overline{v}_{c_i}), \overline{\gamma})\right)$
   where i is the index of a constructor
2. $\neg(\bigwedge \Gamma_c \wedge \bigwedge_{\overline{t} \in T_c} X(\alpha, \alpha, \overline{t}))$

Unlike VTRATGs, the solvability of the FE is not equivalent to the languages being tautological. For induction grammars, we have two distinct notions instead: solvability and validity (referring to the associated formula equation). Solvability clearly always implies validity:

Lemma 5.3.1. *Let G be a decodable induction grammar. If Φ_G is T-solvable, then Φ_G is T-valid.*

Constructing a counterexample for the converse is a bit involved, and we will postpone it until Section 5.4. Solvability, however, is equivalent to the existence of a simple induction proof:

Theorem 5.3.1. *Let G be a decodable induction grammar. Then the following are equivalent:*

1. *Φ_G has a quantifier-free solution modulo T*
2. *There exists a simple induction proof π such that $G(\pi) = G$*

Proof. The formulas in Definition 5.3.2 are equivalent to the initial sequents in the simple induction proof in Definition 2.8.4. Hence whenever the FE Φ_G has a quantifier-free solution modulo the background theory T, we can construct a simple induction proof π with the solution as induction formula such that $G(\pi) = G$. Vice versa, if we have a simple induction proof π then its induction formula is a solution for Φ_G. □

Example 5.3.1. The induction grammar G in Example 2.8.5 induces the following FE:

$$\begin{aligned}\exists X \Big(& X(\alpha, 0, \gamma) \wedge (X(\alpha, \alpha, 0) \to d(\alpha) = \alpha + \alpha) \wedge \\ & \forall \nu \forall \gamma \left(X(\alpha, \nu, \gamma) \wedge X(\alpha, \nu, \nu) \wedge X(\alpha, \nu, 0) \wedge s(\gamma) + s(\nu) = s(s(\gamma) + \nu) \wedge \right. \\ & \left. \gamma + s(\nu) = s(\gamma + \nu) \wedge s(\nu) + s(\nu) = s(s(\nu) + \nu) \to X(\alpha, s(\nu), \gamma) \right) \Big)\end{aligned}$$

This FE has the solution $\varphi(\alpha, \nu, \gamma) := (s(\gamma) + \nu = s(\gamma + \nu) \wedge d(\nu) = \nu + \nu)$.

Let us now show that the language is tautological iff the formula equation is *valid*. In the following Section 5.4, we will then discuss that validity in general does not imply solvability for the formula equations induced by induction grammars. The following technical condition just ensures that $I(G, t)$ has a production for every nonempty nonterminal vector:

Definition 5.3.3. An induction grammar G is called *suitable* iff $P^c_{\overline{\gamma}} \neq \emptyset$ or $\overline{\gamma} = ()$.

Unlike VTRATGs, we cannot give a uniform solution to the formula equations induced by induction grammars. (Since the problem of finding a solution is undecidable, as we will see in Theorem 5.4.2.) However the analogous definition is still useful, as we can use it characterize the *validity* of the formula equation.

Definition 5.3.4. Let G be a decodable induction grammar. Then for every free constructor term t, we define the *canonical solution* $C_t(\alpha, \overline{\gamma})$ recursively as follows:

$$C_{c_i(\overline{s})}(\alpha, \overline{\gamma}) \quad = \quad \bigwedge \Gamma_i[\overline{\nu}_i \backslash \overline{s}] \wedge \bigwedge_l \bigwedge_{\overline{t} \in T_i} C_{s_l}(\alpha, \overline{t})$$

Theorem 5.3.2. *Let G be a suitable decodable induction grammar. Then the following are equivalent:*

1. *G is T-tautological.*
2. *$L(G,t)$ is T-tautological for all free constructor terms t.*
3. *$\Phi_{I(G,t)}$ is T-solvable for all free constructor terms t.*
4. *$\bigwedge_{\overline{s}\in P^c_{\overline{\gamma}}} C_t(t,\overline{s}) \to \varphi(t)$ is T-tautological for all free constructor terms t.*
5. *$T \models \Phi_G$ in full semantics.*

Proof. $1 \Leftrightarrow 2$ by Definition 2.8.9; and $2 \Leftrightarrow 3$ by Theorem 5.2.1. For $5 \Rightarrow 3$ note that $\models \Phi_G \to \Phi_{I(G,t)}$ even in Henkin semantics. For $2 \Leftrightarrow 4$ note that the formula $\bigwedge_{\overline{s}\in P^c_{\overline{\gamma}}} C_t(t,\overline{s}) \to \varphi(t)$ is equivalent to $\bigwedge L(G,t)$. It remains to show $3 \Rightarrow 5$. So assume $T \models \Phi_{I(G,t)}$ for all free construct terms t and let $\mathcal{M}$ be a full higher-order model of T. In a full higher-order model, we can clearly evaluate infinite conjunctions and disjunctions, and define predicates that contain these infinite connectives. A direct way to define the witness for the existential quantifier in Φ_G is by case distinction over all free constructor terms t: we set $X = \lambda\alpha\lambda\nu\lambda\overline{\gamma}\ \bigwedge_t(\nu = t \to C_t(\alpha,\overline{\gamma}))$. (This definition does not require or imply that the constructors are injective.) Every formula listed in Definition 5.3.2 is satisfied: 1. due to the definition of canonical solution, and 2. since $L(G,t)$ is T-tautological for all t. □

Remark 5.3.1. Theorem 5.3.2 constructs the set X in the model using a least fixed-point construction.

5.4 Decidability and existence of solutions

On a practical level, we want to algorithmically solve the formula equations induced by VTRATGs and induction grammars. For VTRATGs, we have seen in Theorem 5.2.1 that this is possible (as long as the language is tautological), and we can even give a nice (albeit exponentially large) canonical solution. For induction grammars, the situation is more complicated:

1. The formula equation may not have a solution, even if the language is tautological:

   Theorem 5.4.1 ([31]). *There exists a suitable decodable induction grammar G with $|\overline{\gamma}| = 1$ such that Φ_G is valid but Φ_G does not have a quantifier-free solution modulo $\emptyset$.*

2. It is undecidable whether the formula equation induced by a given induction grammar is solvable:

   Theorem 5.4.2 ([33]). *There exists a computable sequence of suitable decodable induction grammars $(G_n)_{n\geq 0}$ with $|\overline{\gamma}| = 1$ such that Φ_{G_n} is valid for all n, but the set $\{n \mid \Phi_{G_n}$ has a quantifier-free solution modulo $\emptyset\}$ is undecidable.*

In short, there is no algorithm to solve formula equations, even if we assume as a precondition that the FE is valid. The proofs of Theorems 5.4.1 and 5.4.2 are both based on a similar construction: the following induction grammar G for the sequent $\forall x\ (P(x) \to P(s(x))) \vdash \forall x\ (P(0) \to P(x))$:

$$\tau \to r_1(\gamma)$$
$$\gamma \to s(\gamma) \mid 0$$

A more straightforward induction grammar would use the single production $\tau \to r_1(\nu)$. The trick of replacing ν by γ prevents the straightforward solution $P(0) \to P(\nu)$ to the formula equation. The proofs of Theorems 5.4.1 and 5.4.2 then analyse a potential solution and deduce that such a solution needs to contain arbitrarily large numerals $s^N(\dots)$, a contradiction. This approach crucially depends on the fact that the solution is quantifier-free modulo an empty theory.

It turns out to be practically hard to algorithmically (or even manually) find solutions to formula equations such as the ones induced by induction grammars. We could hope that this problem becomes easier by restricting the induction grammars, for example only considering induction grammars with $|\overline{\gamma}| = 0$—precluding the grammar used by Theorems 5.4.1 and 5.4.2. Another

option is to consider more solutions, for example quantified solutions: for some applications it is not absolutely important that we find a simple induction proof with the specified induction grammar (a quantified solution to the FE would in general result in a proof that is not a simple induction proof).

In the rest of this section we will hence work towards a generalization of Theorems 5.4.1 and 5.4.2, showing that solving the formula equations induced by induction grammars is hard even if we restrict the class of induction grammars and relax the notion of solution. Concretely, we generalize the unsolvability result of Theorem 5.4.1 in both of these directions: we will construct an induction grammar with $|\overline{\gamma}| = 0$ such that the induced formula equation is valid modulo the empty theory, but is not T-solvable for a fixed theory T. It is unfortunately not yet clear how to extend the approach of this section to an undecidability result like Theorem 5.4.2.

Theorem 5.4.3. *Let T be a computably enumerable consistent extension of PA. Then there exists an induction grammar $G = (\tau, \alpha, (\overline{\nu}_c)_c, \overline{\gamma}, P)$ with $\overline{\gamma} = ()$ such that Φ_G is $\emptyset$-valid but Φ_G is not T-solvable.*

The rest of this section is devoted to a proof of Theorem 5.4.3. We will work in an arithmetical theory with function symbols such as d (which in our case doubles the input). In the sequent $\Gamma \vdash \forall x\, \varphi(x)$ that we will construct, the antecedent Γ will only consist of formulas that are provable in $\mathsf{IOpen} \subseteq \mathsf{PA}$ (a weak fragment of Peano arithmetic where induction is restricted to quantifier-free formulas). We will choose the formula $\forall x\, \varphi(x)$ such that it expresses the consistency of a fixed computable enumerable theory T. On a very abstracted level, the general plan could be described as follows: we consider the sequent $\mathsf{PA} \vdash \mathrm{Con}(T)$, and use Gödel's second incompleteness theorem to show that there can be no simple induction proof with the constructed induction grammar (and hence no solution to the formula equation) if $T \supseteq \mathsf{PA}$.

Remark 5.4.1. Most of this argument also works if we replace PA by IOpen. In fact, the condition $T \supseteq \mathsf{IOpen}$ would suffice to show that there are no *quantifier-free* T-solutions to the constructed formula equation. The use of PA is only necessary to show the stronger statement that there no *quantified* solutions as well.

One requirement for this approach is that the constructed induction grammar is tautological. As a small example, consider the sequent $\forall x\, d(s(x)) = s(s(d(x))), \forall x\, s(x) > 0 \vdash \forall x\, (x \neq 0 \to d(x) > 0)$ with the following induction grammar G:

$$\tau \to r_1(v) \mid r_2(v)$$

Then $\Phi_{I(G,n)}$ is equivalent to the following formula, saying that the first n instances of the antecedent imply the conclusion. By Theorem 5.3.2, G is tautological iff this formula is tautological for all n:

$$\underbrace{\bigwedge_{0 \leq i < n} \Big(d(s^{i+1}(0)) = s(s(d(s^i(0)))) \wedge s^{i+1}(0) > 0 \Big)}_{C_n(\alpha)} \to (s^n(0) \neq 0 \to d(s^n(0)) > 0)$$

This formula happens to be *not* tautological for $n > 0$, since the instances for $s(x) > 0$ are not enumerated "fast enough"—we would need the instance for $2n$, but we only get the instances until n. And this brings us to the main technical challenges of choosing the Γ in $\Gamma \vdash \mathrm{Con}(T)$: 1) we need to produce the instances "fast enough" so that $\Phi_{I(G,n)}$ is tautological, and 2) each formula in Γ can only have a *single* universal quantifier. (Syntactically we can of course have multiple quantifiers, but the instances can only vary in one "direction": namely v. Hence we might as well assume that there is a single universal quantifier instantiated with v.) That is, we cannot directly use all instances of a formula such as $\forall x \forall y\, (x + y = y + x)$.

Particularly the restriction of a single universal quantifier makes the choice of Γ interesting. Definition 5.4.1 defines part of the formula that we pick. As usual, 0 is zero and s is the successor. The function $d(x) = 2x$ doubles its argument, and $h(x) = \lfloor x/2 \rfloor$ halves its argument. The predicate $e(x)$ is true iff x is even. The function c codes pairs, and π_1, π_2 project to the first and second component, resp. The coding of pairs is defined by interleaving the digits in a base 2 representation, i.e. $\pi_1((a_n \ldots a_1)_2) = (\ldots a_3 a_1)_2$ and $\pi_2((a_n \ldots a_1)_2) = (\ldots a_4 a_2)_2$. Note that as a binary function, $c(x, y)$ is only defined for $x < M$ where M is a fixed bound since we only have a single universal quantifier. We will use c to encode a word $w_1 \ldots w_n$ as $c(w_1, c(w_2, \ldots, c(w_n, 0)))$ (i.e., as a linked list with c as the "cons" operation).

Definition 5.4.1. Let $M > 0$. The formula $\theta_M(x)$ is defined as:

$$
\begin{aligned}
& 0 \neq s(0) \\
\wedge\ & d(0) = 0 \wedge d(s(x)) = s(s(d(x))) \\
\wedge\ & h(0) = 0 \wedge h(s(0)) = 0 \wedge h(s(s(x))) = s(h(x)) \\
\wedge\ & e(0) \wedge \neg e(s(0)) \wedge (e(s(x)) \leftrightarrow \neg e(x)) \\
\wedge\ & (e(x) \rightarrow \pi_1(x) = d(\pi_1(h(h(x))))) \\
\wedge\ & (\neg e(x) \rightarrow \pi_1(x) = s(d(\pi_1(h(h(x)))))) \\
\wedge\ & \pi_2(x) = \pi_1(h(x)) \\
\wedge\ & c(0,0) = 0 \\
\wedge\ & \bigwedge_{2i<M} (e(x) \rightarrow c(s^{2i}(0), x) = d(d(c(s^i(0), h(x))))) \\
\wedge\ & \bigwedge_{2i<M} (\neg e(x) \rightarrow c(s^{2i}(0), x) = d(s(d(c(i, h(x)))))) \\
\wedge\ & \bigwedge_{2i+1<M} c(s^{2i+1}(0), x) = s(c(s^{2i}(0), x))
\end{aligned}
$$

The instances $\bigwedge_{i<n} \theta_M(i)$ in Definition 5.4.1 suffice to compute $=, d, h, e, \pi_1$, and π_2 for input values up to n:

Lemma 5.4.1. *Let us make conversion of numbers to numerals explicit by writing $\underline{n} = s^n(0)$, and extend this notation to truth values via $\underline{\bot} = \bot$ and $\underline{\top} = \top$ as well. Then for any $M, n > 0$:*

$$\bigwedge_{i<n} \theta_M(\underline{i}) \vdash \underline{a} = \underline{b} \leftrightarrow \underline{a = b} \qquad \text{for all } a, b < n$$

$$\bigwedge_{i<n} \theta_M(\underline{i}) \vdash d(\underline{a}) = \underline{d(a)} \qquad \text{for all } a < n$$

$$\bigwedge_{i<n} \theta_M(\underline{i}) \vdash h(\underline{a}) = \underline{h(a)} \qquad \text{for all } a < n$$

$$\bigwedge_{i<n} \theta_M(\underline{i}) \vdash e(\underline{a}) \leftrightarrow \underline{e(a)} \qquad \text{for all } a < n$$

$$\bigwedge_{i<n} \theta_M(\underline{i}) \vdash \pi_1(\underline{a}) = \underline{\pi_1(a)} \qquad \text{for all } a < n$$

$$\bigwedge_{i<n} \theta_M(\underline{i}) \vdash \pi_2(\underline{a}) = \underline{\pi_2(a)} \qquad \text{for all } a < n$$

$$\bigwedge_{i<n} \theta_M(\underline{i}) \vdash c(\underline{a}, \underline{b}) = \underline{c(a,b)} \qquad \text{for all } a < M \text{ and } a^2 + 2b^2 < n$$

We call the functions d, h, π_1, π_2 and the predicates $=, e$ *quickly representable* since they can be computed for all arguments less than n.

Proof. All of the claims are proven by recursion on a and b; the arguments always strictly decrease in the recursive definition of the functions and predicates. For example for h, we can prove $h(0) = 0$ and $h(s(0)) = 0$ since this already occurs literally in $\theta_M(0)$. And we can prove $h(\underline{a+2}) = s(h(\underline{a}))$ for any $a < n$; by the induction hypothesis we can already prove $h(\underline{a}) = \underline{h(a)}$.

The case for $=$ is interesting since it is the only *binary* predicate where we can show quick representation. Clearly if $a = b$, then we can prove $\underline{a} = \underline{b}$ by reflexivity. Otherwise $e(h^i(\underline{a})) \nleftrightarrow e(h^i(\underline{b}))$ for some suitably chosen i. □

Note that some combinations may not be quickly representable: for example we cannot prove $h(d(\underline{a})) = \underline{a}$ for $0 < a \leq n$ since the intermediate result $d(\underline{a})$ is too large.

We can now turn towards defining the formula that will express the consistency of the theory T. We will formalize the assertion that a given program P does not terminate. (Later on in the proof of Theorem 5.4.3, we will instantiate P by a program that searches for a proof of a contradiction from T.) Concretely, the program P will be represented as an instance of Post-Correspondence, i.e., $P = ((w_1^1, \ldots, w_n^1), (w_1^2, \ldots, w_n^2))$. Recall that a solution is a non-empty finite sequence $(i_1, \ldots, i_k)$ such that $w_{i_1}^1 \ldots w_{i_k}^1 = w_{i_1}^2 \ldots w_{i_k}^2$. The predicate $I(x)$ is true iff x is a non-empty sequence of the form $(i_1, \ldots, i_k)$. The sequence is represented as a list using the c function. The function $f^1(c(i_1, \ldots c(i_n, 0) \ldots))$ (and analogously f^2) maps the input sequence to the resulting concatenation $w_{i_1}^1 \ldots w_{i_k}^1$ (also represented as a list).

Definition 5.4.2. Let $P = ((w_1^1, \ldots, w_n^1), (w_1^2, \ldots, w_n^2))$ be an instance of Post-Correspondence. Define $w_{i,j}^h$ as the j-th letter of w_i^h for $1 \leq j \leq |w_i^h|$, and assume that letters are natural numbers such that $0 < w_{i,j}^h < M$ for some M.

Define the following sequent:

$$\forall x\ \theta_M(x),$$
$$\forall x \bigwedge_{h\in\{1,2\}} \bigwedge_{i\le n} \Big(\pi_1(x) = i \to f^h(x) = c(w^h_{i,1}, \dots c(w^h_{i,|w^h_i|}, f^h(\pi_2(x))))\Big),$$
$$\forall x \bigwedge_{h\in\{1,2\}} \bigwedge_{i\le n} \Big(\pi_1(x) = i \to \pi_1(f^h(x)) = w^h_{i,1} \wedge \dots \wedge \pi_1(\pi_2^{|w^h_i|-1}(f^h(x))) = w^h_{i,|w^h_i|} \wedge \pi_2^{|w^h_i|}(f^h(x)) = f^h(\pi_2(x))\Big),$$
$$\neg I(0), \forall x\ (I(x) \leftrightarrow (\pi_2(x) = 0 \vee I(\pi_2(x))) \wedge \bigvee_{i\le n} \pi_1(x) = i)$$
$$\vdash \forall x\ (I(x) \to f^1(x) \neq f^2(x))$$

And define the induction grammar Solve_P:

$$\tau \to r_1(0) \mid \dots \mid r_1(M) \mid r_1(v) \mid r_2(v) \mid r_3 \mid r_4(v)$$

Lemma 5.4.2. *Let P be an instance of Post-Correspondence. Then Φ_{Solve_P} is $\emptyset$-valid iff P has no solution.*

Proof. We need to show that $C_n(\alpha) \to I(n) \to f^1(n) \neq f^2(n)$ is provable. In a similar way as in Lemma 5.4.1, we can easily see that I is quickly representable. To see that $f^1(n) = f^2(n)$ is quickly representable as well, we make a case distinction. If the two sides of the equality are different, then the strings they represent are different at position j and we can prove $\pi_1(\pi_2^j(f^1(n))) \neq \pi_1(\pi_2^j(f^1(n)))$ for this position j. (This is the reason we added the instances up to M in the induction grammar: so that we can prove all inequalities between letters.) If $f^1(n) = f^2(n)$ are equal, then we can unfold both sides to identical terms of the form $c(s^{\cdots}(0), \dots c(s^{\cdots}(0), 0))$. Hence $\Phi_{I(\mathrm{Solve}_P,n)}$ is $\emptyset$-valid iff $I(n) \to f^1(n) \neq f^2(n)$ is true in the standard model, and this is the case iff n does not code a solution to P. □

Proof of Theorem 5.4.3. There exists a program that terminates if and only if T is inconsistent—e.g. by searching for a proof of false. That such a program exists is provable in PA. Since Post-Correspondence is Turing-complete, there is an instance P of Post-Correspondence such that P is solvable iff T

is inconsistent; and this is provable in PA as well. Set $G = \mathrm{Solve}_P$. Since T is consistent by assumption, Φ_G is $\emptyset$-valid by Lemma 5.4.2. If Φ_G were T-solvable, then T proves that P has no solution and hence $T \vdash \mathrm{Con}(T)$, a contradiction to Gödel's second incompleteness theorem. □

5.5 Examples of difficult formula equations

Before we come to algorithms that solve formula equations, let us first look at some concrete practical examples. Recall that our motivation for formula equations was to, ultimately, find the matrix of the induction formula for a simple induction proof. We have already figured out the quantifier inferences by finding an induction grammar that covers some finite family of instance languages. Solving the formula equation induced by this induction grammar is hence the only step missing to the construction of a proof with induction.

Many of these induced formula equations are surprisingly hard to solve in practice. It appears to be even harder to show for a concrete formula equation that it has no solutions (unless it is not valid). In this section we will show several such formula equations that are induced by induction grammars for real-world problems from the TIP library ([21]) and a formalization of the fundamental theorem of arithmetic in GAPT (we will come back to these in Section 7.4). All formula equations have $T = \emptyset$. For none of these formula equations could we find a solution or conversely, show that there is no solution.

`subpl`

Here we consider a property of the predecessor function p on natural numbers, namely that it commutes with the truncating subtraction $(-)$. (The constant c comes from Skolemization.)

$$\begin{aligned} &\forall x\ (x - 0 = x), \\ &\forall x\ \forall y\ (x - s(y) = p(x) - y), \\ \vdash\ &\forall x\ (p(c) - x = p(c - x)) \end{aligned}$$

The MaxSAT algorithm produces the following induction grammar (covering

the languages of proofs generated by an automatic theorem prover):

$$\tau \to r_1(\gamma) \mid r_2(\gamma, \nu) \mid r_3$$
$$\gamma \to p^2(\gamma) \mid p(c) \mid c$$

It is easy to see that this induction grammar is tautological, and that the induced formula equation is hence valid. Fix the parameter n. First observe that for $i > 0$, the instance grammar generates the instances $p^{i+1}(c) - s^{n-i}(0) = p^{i+2}(c) - s^{n-i-1}(0)$. However this happens in a refreshingly different way: first the instance production corresponding to $\gamma \to p^2(\gamma)$ is applied $\lfloor i/2 \rfloor$ times, and then $\gamma \to c$ or $\gamma \to p(c)$ depending on the parity of i. (A more natural and human way would be to use a production $\gamma \to p(\gamma)$. We would need to use that production $i - 1$ times, followed by $\gamma \to p(c)$.) Hence we can derive $p(c) - s^n(0) = p^{n+1}(c) - 0$ from the instance language.

In a similar way, the instances $p^i(c) - s^{n-i}(0) = p^{i+1}(c) - s^{n-i-1}(0)$ are generated. This allows us to derive $c - s^n(0) = p^n(c) - 0$. Since also $p^n(c) - 0 = 0$ is generated (again by a similar argument), we get $p(c) - s^n(0) = p(c - s^n(0))$ and the instance language is tautological.

However because of the unconventional way the instances are generated, there is no obvious choice for a solution to the induced formula equation:

$$\begin{aligned}
&\exists X \big(\forall \gamma \, (\gamma - 0 = \gamma \to X(\alpha, 0, \gamma)) \wedge \\
&\quad \forall \nu \, \forall \gamma \, (X(\alpha, \nu, p^2(\gamma)) \wedge X(\alpha, \nu, c) \wedge X(\alpha, \nu, p(c)) \wedge \gamma - 0 = \gamma \wedge \\
&\qquad \gamma - s(\nu) = p(\gamma) - \nu \to X(\alpha, s(\nu), \gamma)) \wedge \\
&\qquad (X(\alpha, \alpha, c) \wedge X(\alpha, \alpha, p(c)) \to p(c) - \alpha = p(c - \alpha)) \big)
\end{aligned}$$

subps

This formula equation is induced by an induction grammar for similar lemma as `subpl`, also produced by the MaxSAT algorithm as a covering induction

grammar for proofs produced by an ATP.

$$\begin{aligned}
&\forall x\ (x - 0 = x),\\
&\forall x\ \forall y\ (x - s(y) = p(x) - y),\\
&\forall x\ (p(s(x)) = x),\\
\vdash\ &\forall x\ (p(s(c) - x) = c - x)
\end{aligned}$$

It is easy to see that every solution for the FE in `subpl` also solves the following FE, and it is just as unclear whether the FE has a solution or not:

$$\begin{aligned}
\exists X\ \big(&\forall \gamma\ (\gamma - 0 = \gamma \wedge p(s(c)) = c \rightarrow X(\alpha, 0, \gamma)) \wedge\\
&\forall \nu\ \forall \gamma\ (X(\alpha, \nu, p^2(\gamma)) \wedge X(\alpha, \nu, s(c)) \wedge X(\alpha, \nu, p(s(c))) \wedge \gamma - 0 = \gamma \wedge\\
&\qquad p(s(c)) = c \wedge \gamma - s(\nu) = p(\gamma) - \nu \rightarrow X(\alpha, s(\nu), \gamma)) \wedge\\
&\quad (X(\alpha, \alpha, s(c)) \wedge X(\alpha, \alpha, p(s(c))) \wedge p(s(c)) = c \rightarrow\\
&\qquad p(s(c) - \alpha) = c - \alpha)\big)
\end{aligned}$$

`prod/prop_13`

This problem from the TIP library (hence the different names for zero and successor) states a property of the halving function $h(x) = \lfloor \frac{x}{2} \rfloor$:

$$\begin{aligned}
&\forall x\ (Z + x = x),\\
&\forall x\ \forall y\ (S(x) + y = S(x + y)),\\
&h(Z) = Z \wedge h(S(Z)) = Z,\\
&\forall x\ (h(S(S(x))) = S(h(x))),\\
\vdash\ &\forall x\ (h(x + x) = x)
\end{aligned}$$

The MaxSAT algorithm produces the following tautological induction grammar:

$$\begin{aligned}
&\tau \rightarrow r_1(\gamma) \mid r_2(\nu, \gamma) \mid r_3 \mid r_4(S^2(\nu + \nu))\\
&\gamma \rightarrow \alpha \mid \gamma \mid \nu
\end{aligned}$$

As a first guess, we might think that $h(\nu + \nu) = \nu$ could be a solution to the induced formula equation. Alas, this does not work because we do not

have the instance $v + S(v) = S(v + v)$. Another potential "solution" that comes to mind is $v + \gamma = S^v(\gamma) \wedge h(S^{2v}(0)) = S^v(0)$, this would work if not for the fact that is not a first-order formula. So it is still open whether the following formula equation has a solution or not:

$$\begin{aligned}
&\exists X \big(\forall \gamma\ (Z + \gamma = \gamma \wedge h(Z) = Z \rightarrow X(\alpha, Z, \gamma)) \wedge \\
&\quad \forall v\ \forall \gamma\ (X(\alpha, v, \alpha) \wedge X(\alpha, v, \gamma) \wedge X(\alpha, v, v) \wedge Z + \gamma = \gamma \wedge h(Z) = Z \wedge \\
&\qquad h(S^2(v + v)) = S(h(v + v)) \wedge S(v) + \gamma = S(v + \gamma) \rightarrow \\
&\qquad X(\alpha, S(v), \gamma)) \wedge \\
&\qquad (X(\alpha, \alpha, \alpha) \wedge h(Z) = Z \rightarrow h(\alpha + \alpha) = \alpha)\big)
\end{aligned}$$

isaplanner/prop_03

Here we show that the number of occurrences of an element in a list increases if we append something to the list. Lists are an inductive data type with the constructors *nil* and *cons*; e.g. $cons(a, cons(b, nil))$ is a two-element list. The function $c(n, l)$ counts how often n occurs in the list l; concatenation of lists is denoted by $a(l_1, l_2)$, $e(m, n)$ is a predicate expressing the equality of elements m and n. The constants n and ys are obtained from Skolemization.

$$\begin{aligned}
&\forall x\ le(Z, x), \\
&\forall x\ \forall y\ (le(S(x), S(y)) \leftrightarrow le(x, y)), \\
&\forall w\ (c(w, nil) = Z) \\
&\forall w\ \forall x\ \forall l\ (\neg e(w, x) \rightarrow c(w, cons(x, l)) = c(w, l)) \\
&\forall w\ \forall x\ \forall l\ (e(w, x) \rightarrow c(w, cons(x, l)) = S(c(w, l))) \\
&\forall x\ (a(nil, x) = x) \\
&\forall x\ \forall y\ \forall z\ (a(cons(x, y), z) = cons(x, a(y, z))) \\
\vdash\ &\forall x\ (le(c(n, x), c(n, a(x, ys))))
\end{aligned}$$

The following formula equation is induced by an induction grammar found

by the MaxSAT algorithm. It is unknown whether it has a solution or not:

$$
\begin{aligned}
\exists X\ \big(\forall \gamma_0\ \forall \gamma_1\ \big(&le(Z, c(n, ys)) \wedge c(n, nil) = Z \wedge \\
&(\neg e(n, \gamma_1) \rightarrow c(n, cons(\gamma_1, \gamma_0)) = c(n, \gamma_0)) \wedge \\
&(e(n, \gamma_1) \rightarrow c(n, cons(\gamma_1, \gamma_0)) = S(c(n, \gamma_0))) \wedge \\
&a(nil, ys) = ys \rightarrow X(\alpha, nil, \gamma_0, \gamma_1)) \wedge \\
\forall \nu\ \forall \nu_0\ \forall \gamma_0\ \forall \gamma_1\ \big(&X(\alpha, \nu_0, a(\nu_0, ys), \nu) \wedge X(\alpha, \nu_0, \nu_0, \nu) \wedge \\
&le(Z, c(n, ys)) \wedge c(n, nil) = Z \wedge \\
&(\neg e(n, \gamma_1) \rightarrow c(n, cons(\gamma_1, \gamma_0)) = c(n, \gamma_0)) \wedge \\
&(e(n, \gamma_1) \rightarrow c(n, cons(\gamma_1, \gamma_0)) = S(c(n, \gamma_0))) \wedge \\
&(le(S(c(n, \nu_0)), S(c(n, a(\nu_0, ys)))) \leftrightarrow \\
&\quad le(c(n, \nu_0), c(n, a(\nu_0, ys)))) \wedge \\
&a(nil, ys) = ys \wedge a(cons(\nu, \nu_0), ys) = cons(\nu, a(\nu_0, ys)) \rightarrow \\
&X(\alpha, cons(\nu, \nu_0), \gamma_0, \gamma_1)) \wedge \\
\forall \gamma_0\ \forall \gamma_1\ \big(&X(\alpha, \alpha, \gamma_0, \gamma_1) \wedge le(Z, c(n, ys)) \wedge c(n, nil) = Z \wedge \\
&a(nil, ys) = ys \rightarrow le(c(n, \alpha), c(n, a(\alpha, ys))))\big)
\end{aligned}
$$

5.6 Solution algorithm using forgetful inference

On a practical level, given a tautological grammar we want to algorithmically generate a proof with that grammar. If we recall Theorems 5.2.1 and 5.3.1, this means that we need to find a solution to the associated formula equation for the grammar. For VTRATGs, this problem is decidable as we can always use the (in general exponentially large) canonical solution as given by Definition 5.2.2. For induction grammars, this problem is undecidable by Theorem 5.4.2. An impractical algorithm to solve the formula equations induced by induction grammars would be to enumerate all formulas, and for every formula check whether it is a solution and return it if it is indeed a solution.

A practically more effective (however incomplete in the sense that it will not find a solution for every solvable formula equation) algorithm was introduced

in [31] and is based on an algorithm that improves the solutions for formula equations induced by VTRATGs which was introduced in [51]. (See also [36] for a newer presentation that is closer to the formalism of this section.)

The central idea behind this improvement algorithm is forgetful inference: we regard the solution to the formula equation as a formula in conjunctive normal form, and then apply inferences such as resolution to these clauses. It is called forgetful because whenever we do an inference we remove the premises from the CNF.

Definition 5.6.1 (simplification of CNFs)**.** We define the binary relations $\rhd_p, \rhd_r, \rhd_f$ on the set of clauses (where a clause is a set of literals) as the smallest relations containing the following:

$$C \cup \{C \cup \{l = r\}, D[l]\} \rhd_p C \cup \{C \cup D[r]\}$$
$$C \cup \{C \cup \{l = r\}, D[r]\} \rhd_p C \cup \{C \cup D[l]\}$$
$$C \cup \{C \cup \{l\}, \{\neg l\} \cup D\} \rhd_r C \cup \{C \cup D\}$$
$$C \cup \{C\} \rhd_f C$$

The relations $\rhd_p, \rhd_r, \rhd_f$ are called simplification by forgetful resolution, forgetful paramodulation, and forgetting, respectively. We define the relation $\rhd = \rhd_r \cup \rhd_p \cup \rhd_f$ as their union.

We also lift the simplification relation $\rhd$ to solutions of formula equations, by rewriting a single formula at a time:

Definition 5.6.2 (simplification of solutions of formula equations)**.** Let $\overline{\varphi} = (\varphi_1, \ldots, \varphi_n)$ and $\overline{\psi} = (\psi_1, \ldots, \psi_n)$ be a sequences of formulas. Then $\overline{\varphi} \rhd \overline{\psi}$ iff there exists a j such that $\varphi_j \rhd \psi_j$ and $\varphi_i = \psi_i$ for all $i \neq j$.

Example 5.6.1. Consider the sequent $\forall x\ (P(x) \to P(s(x))) \vdash P(0) \to P(s^4(0))$ and the following tautological VTRATG G:

$$A \to r_1(B) \mid r_1(s(B)) \mid r_2$$
$$B \to 0 \mid s^2(0)$$

Then the canonical solution C^G is given by:

$$(P(\alpha) \to P(s(\alpha))) \wedge (P(s(\alpha)) \to P(s^2(\alpha))) \wedge \neg(P(0) \to P(s^4(\alpha)))$$

Via simplification we get a much nicer solution:

$$\begin{aligned} C^G &\rhd_f (P(\alpha) \to P(s(\alpha))) \wedge (P(s(\alpha)) \to P(s^2(\alpha))) \\ &\rhd_r P(\alpha) \to P(s^2(\alpha)) \end{aligned}$$

Of course the relation $\rhd$ can also lead to formulas that are no longer solutions of the formula equation. In the actual algorithm that simplifies the solution we hence need to check that the formula is still a solution. The function FindConsequences$(\Phi, \overline{\psi})$ in Algorithm 4 computes the $\rhd$-simplified solutions for a formula equation Φ, skipping the branches where the simplified formula is not a solution. The way we use this function to compute a "nice" solution to the formula equations induced by a VTRATG G is as follows: we compute the $\rhd$-consequences using FindConsequences$(\Phi_G, \overline{C^G})$ and then return the smallest solution we have found (e.g. counting the number of logical connectives).

Algorithm 4 Compute $\rhd$-consequences for solution improvement

```
function FindConsequences(Φ: formula equation, φ̄: solution for Φ)
    yield φ̄
    for each ψ̄ such that φ̄ ▷ ψ̄ do
        if ψ̄ is a solution for Φ and we did not process ψ̄ before then
            FindConsequences(Φ, ψ̄)
        end if
    end for
end function
```

The FindConsequences algorithm is practically effective; starting from the analytic content of the canonical solution, which essentially is just a conjunction of instances of the end-sequent, it generates short and interesting non-analytic solutions. In an evaluation on the TPTP [36], we got meaningful solutions such as $\mathsf{complement}(\mathsf{complement}(x)) = x$ (this solution was found for the formula equation induced by a small covering VTRATG for the SET190-6 problem).

An obvious and practical optimization in Algorithm 4 is to perform the simplification steps $\rhd_p, \rhd_r, \rhd_f$ in a particular order: we can always move the $\rhd_f$ steps to the end, without decreasing the set of found consequences. Practically,

we first compute all $(\rhd_p \cup \rhd_r)$-consequences, and then forget clauses. Another step-permutation concerns formula equations for VTRATGs with more than two non-terminals (and hence more than one predicate variable). Clearly, performing simplification on the different predicate variables is independent. However it may be the case that one permutation of the steps may produce a sequence of formulas that is not a solution as an intermediate step. Let us consider the case for two predicate variables as an example:

$$\begin{aligned}\exists X_1 \exists X_2 \big(&\forall \beta\, \forall \gamma\, (\Gamma_0 \to X_2(\gamma) \vee X_1(\gamma, \beta)) \\ &\wedge \forall \gamma\, (\Gamma_1 \wedge X_1(\gamma, t_1) \wedge \cdots \wedge X_1(\gamma, t_m) \to X_2(\gamma)) \\ &\wedge (\Gamma_2 \wedge X_2(s_1) \wedge \cdots \wedge X_2(s_n) \to \bot)\big)\end{aligned}$$

Observe that X_i for $i \in \{1, 2\}$ only occurs negatively in one conjunct, the only other occurrences are positive and before that conjunct. For this reason, we can always permute the simplification steps in such a way that we first perform all simplifications for X_2 and then the ones for X_1.

A possible extension is to include consequences modulo an equational background theory T. We add an additional simplification step $\rhd_e$ such that $C \cup \{D[l\sigma]\} \rhd_e C \cup \{D[r\sigma]\}$ for $l = r \in T$, and only search for consequences where we use an equation $l = r \in T$ at most once. That is, FindConsequences accepts the list of equations T as an additional argument and removes the used equation in the recursive call.

The solution algorithm for formula equations induced by induction grammars is now based on the improvement algorithm described above. In a certain sense, we run FindConsequences on an instance of the formula equation for a particular free constructor term:

Definition 5.6.3. Let G be an induction grammar and t a free constructor term. Then define the formula equation $\tilde{\Phi}_{G,t}$ as follows:

$$\begin{aligned}\tilde{\Phi}_{G,t} \;=\; \exists X\, \tilde{\Psi}_{G,t} \;=\; \exists X \Big(&\forall \overline{\gamma}\, (C_t(\alpha, \overline{\gamma}) \to X(\alpha, t, \overline{\gamma})) \wedge \\ &\Big(\bigwedge \Gamma_c \wedge \bigwedge_{\overline{s} \in T_c} X(\alpha, \alpha, \overline{s}) \to \varphi(\alpha)\Big)\Big)\end{aligned}$$

Let us collect some easy observations about this instantiated formula equation $\tilde{\Phi}_{G,t}$:

Lemma 5.6.1. *Let G be an induction grammar and t a free constructor term. Then:*

1. $\vdash \Psi_G \to \tilde{\Psi}_{G,t}$
2. *If Φ_G has φ as a solution, then $\tilde{\Phi}_{G,t}$ also has the solution φ.*
3. *If $\varphi(\alpha, v, \overline{\gamma})$ is a solution for $\tilde{\Phi}_{G,t}$, then so is $\varphi(\alpha, t, \overline{\gamma})$.*
4. *If G is tautological, then $\tilde{\Phi}_{G,t}$ has the solution $C_t(\alpha, \overline{\gamma})$.*

So to sum up Lemma 5.6.1, the solutions of $\tilde{\Phi}_{G,t}$ and Φ_G differ in two ways: $\tilde{\Phi}_{G,t}$ has more solutions, and the solutions of $\tilde{\Phi}_{G,t}$ do not distinguish between v and t. We will need to take both of these issues into account.

Algorithm 5 Heuristic solution for formula equations induced by induction grammars

```
function SolveIndFE(G: induction grammar, t: free constructor term)
    for each φ yielded by FindConsequences(Φ̃_{G,t}, C_t(α, γ̄)) do
        for every φ′ such that φ′[ν\t] = φ do
            if φ′ is a solution of Φ_G then
                return φ′
            end if
        end for
    end for
    return none
end function
```

Algorithm 5 is obviously incomplete since it always terminates: there exist induction grammars whose induced formula equation is solvable, yet the algorithm will not return a solution. Let us consider an easy example where it fails to find a solution:

Example 5.6.2. Consider the following sequent stating that a number is even if

and only it is not odd:

$$\begin{aligned} & e(0) \wedge \neg o(0), \\ & \forall x\ (e(s(x)) \leftrightarrow \neg e(x)), \\ & \forall x\ (o(s(x)) \leftrightarrow \neg o(x)) \\ \vdash\ & \forall x\ (e(x) \leftrightarrow \neg o(x)) \end{aligned}$$

The following induction grammar for this sequent is tautological:

$$\tau \longrightarrow r_1 \mid r_2(\nu) \mid r_3(\nu)$$

We can also solve the induced formula equation using the solution $e(\nu) \leftrightarrow \neg o(\nu)$. However if try to solve it using Algorithm 5, then we first compute the following canonical solution $C_t(\alpha)$ for example for $t = s(0)$, consisting of the following clauses in CNF:

$$\begin{aligned} e(0) \\ \neg o(0) \\ e(s(0)) \vee e(0) \\ \neg e(s(0)) \vee \neg e(0) \\ o(s(0)) \vee o(0) \\ \neg o(s(0)) \vee \neg o(0) \end{aligned}$$

Resolution of these clauses (no matter the choice of t) can only produce clauses of the form $\pm o(s^n(0))$, $\pm e(s^n(0))$, $\pm o(s^n(0)) \vee \pm o(s^m(0))$, or $\pm e(s^n(0)) \vee \pm e(s^m(0))$. There are no clauses that mix e and o as in the natural solution. An important observation is that any solution to the formula equation will be true in the standard model (the natural numbers with e interpreted as even, and o as odd). If we build a solution as a conjunction of these clauses, then every clause will be valid. For example, $e(s(0))$, $e(\nu) \vee e(s(s(\nu)))$ cannot be clauses in a solution. So assume that we have such a CNF $C(\nu)$ where $C(\alpha) \wedge e(0) \wedge \neg o(0) \longrightarrow (e(\alpha) \leftrightarrow \neg o(\alpha))$ is provable. Without loss of generality $C(\nu)$ contains no clause of the form $\pm e(s^m(\nu)) \vee \pm e(s^n(0))$ or $\pm o(s^m(\nu)) \vee \pm o(s^n(0))$: if such a clause is true in the standard model, then one of its literal is valid as well and we can replace the clause by the literal.

Now every clause in $C(v)$ has the property that either all literals contain v, or else none contain v. We can now construct a counter-model M for $C(\alpha) \wedge e(0) \wedge \neg o(0) \rightarrow (e(\alpha) \leftrightarrow \neg o(\alpha))$: we use the standard model, but interpret o differently. Interpret α as the smallest number larger than all numerals occurring in the formula. This has the effect that the subterms containing α and the subterms not containing α are interpreted as disjoint subsets. The predicate e is interpreted as the even numbers, however the predicate o is interpreted differently for large numbers: interpret $o(n)$ as n is odd if $n < \alpha^M$ and as n is even if $n \geq \alpha^M$.

5.7 Solution algorithm using interpolation

Formula equations are closely related to the concept of constrained Horn clauses studied in program verification [12]. Similarly to how the formula equations of this chapter capture the conditions for formulas to be cut formulas and induction formulas, sets of constrained Horn clauses capture the necessary conditions for formulas to be program invariants (such as loop invariants or pre-/postconditions). Solving a set of constrained Horn clauses hence amounts to finding the correct invariants, and thus proving a given correctness property of the program.

A constrained Horn clause is a formula of the form $C(\overline{x_1}, \ldots, \overline{x_n}, \overline{y}) \wedge X_1(\overline{x_1}) \wedge \cdots \wedge X_n(\overline{x_n}) \rightarrow Y(\overline{y})$ or $C(\overline{x_1}, \ldots, \overline{x_n}) \wedge X_1(\overline{x_1}) \wedge \cdots \wedge X_n(\overline{x_n}) \rightarrow \bot$ where $X_1, \ldots, X_n, Y$ are predicate variables and the constraint C is a formula that does not contain any predicate variables. The term "Horn" refers to the structure of the predicate variables, the formula C can be arbitrarily complex and also contain additional negations, etc. Just as with formula equations, a solution is a substitution for the predicate variables that validates all clauses. Given a set of constrained Horn clauses $\mathcal{C}$, we will regard it as the formula equation $\exists \overline{X} \bigwedge_{C \in \mathcal{C}} \forall \overline{x} \bigvee C$. We will also call formula equations of this form constrained Horn clause problems.

There is a considerable amount research on the automatic solution of constrained Horn clause problems, using a plethora of techniques ranging from predicate abstraction [44, 57], randomized enumeration [38], machine learn-

ing [103], abstract interpretation [56], to even tree automata [62]. Most of these approaches rely on SMT solvers in their implementations.

The typical setting for these solvers is to consider constrained Horn clauses in an expressive background theory that is well supported by SMT solvers (such as e.g. Presburger arithmetic), and to disallow uninterpreted function symbols. A typical constrained Horn clause might be $y < x \wedge X(x, y) \rightarrow X(x + 1, x + y)$ where x and y range over the integers. For this setting, there is a large number of state-of-the-art provers that can solve even large industrial problems, cf. the results of the yearly CHC competition [20].

When solving the formula equations induced by induction grammars or VTRATGs, we typically work with a weaker background theory (usually just equality) but with uninterpreted function symbols. In this section we will use an approach for constrained Horn clauses to solve formula equations as induced by VTRATGs and induction grammars. The Duality solver [69] uses interpolation at its core to find the solutions.

From a practical point of view, the Duality implementation in Z3 was unfortunately removed in version 4.8.0[1] as the underlying interpolation code was unmaintained. Even before the removal, the solver often failed for our formula equations due to unsupported inferences in the proofs generated by Z3. Since we use the solver in a very different setting compared to what it was intended to solve (many uninterpreted function symbols modulo equality as opposed to none modulo Presburger arithmetic), it is not unexpected to encounter edge cases in the implementation. Hence we opted for a reimplementation of the algorithm in GAPT, specialized to the class of formula equations that we consider.

Each constrained Horn clause problem induces a directed graph with the occurrences of the predicate variables (including $\bot$ if there is no predicate variable on the right side) as vertices. There is an edge from an occurrence $X^1(\bar{t})$ to $Y^2(\bar{s})$ (using superscripts to disambiguate occurrences) iff either:

a) Both occurrences are in the same clause such that $X^1(\bar{t})$ is on the left and $Y^2(\bar{s})$ on the right side, or

b) $X = Y$, $X^1(\bar{t})$ occurs on the right side, and $Y^2(\bar{s})$ occurs on the left side.

[1] `https://github.com/Z3Prover/z3/pull/1646`

Constrained Horn clause problems are called recursive if the graph is cyclic, daglike if the graph is acyclic, and treelike if the graph is a tree. Treelike problems can be easily solved using interpolation, daglike problems will require a potentially exponential unrolling step; the real difficulty lies in the recursive case. (Cf. the difficulty in solving formula equations for VTRATGs and induction grammars, resp.)

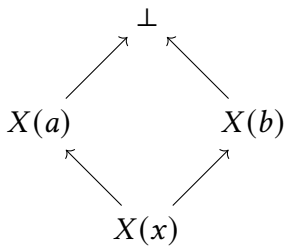
Example 5.7.1. The following constrained Horn clause problem is treelike since its graph $X(x) \to X(a) \to \bot$ is a tree (even a path in this case):

$$\exists X \left(\forall x \, (P(x) \to X(x)) \wedge (X(a) \wedge \neg P(a) \to \bot) \right)$$

Example 5.7.2. The following constrained Horn clause problem is daglike but not treelike:

$$\exists X \left(\forall x \, (P(x) \to X(x)) \wedge (X(a) \wedge X(b) \wedge \neg (P(a) \wedge P(b)) \to \bot) \right)$$

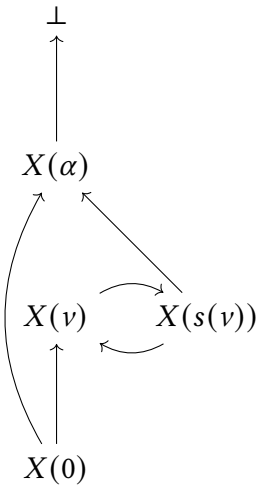
It has the following graph, which is acyclic:



Example 5.7.3. The formula equations induced by induction grammars are constrained Horn clause problems (after applying some propositional equivalences to bring them into the correct syntactic form—mainly adding the $\bot$). For example consider the following (not particularly meaningful but small) one:

$$\exists X \left((P(a) \to X(0)) \wedge \forall v \, (P(a) \wedge X(v) \to X(s(v))) \wedge (X(\alpha) \wedge \neg P(a) \to \bot) \right)$$

It has the following graph, which contains a cycle at $X(v)$:



Conceptually, on a very abstracted level, Duality solves formula equations using two unrolling operations. The first unrolling turns a recursive formula equation into a daglike one, the second unrolling step turns a daglike one into a treelike one. There is a straightforward way to solve treelike formula equations using interpolation. The solutions to the unrolled treelike formula equations can then always be turned into a solution for the daglike formula equation. Only the last step is hard, where we need to convert a solution to the unrolled daglike formula equation to a solution to the original recursive one.

Solving treelike formula equations. Given a treelike formula equation we can always bring it in a form where all occurrences of a predicate variables are applied to the same variables. This is accomplished by adding additional equalities:

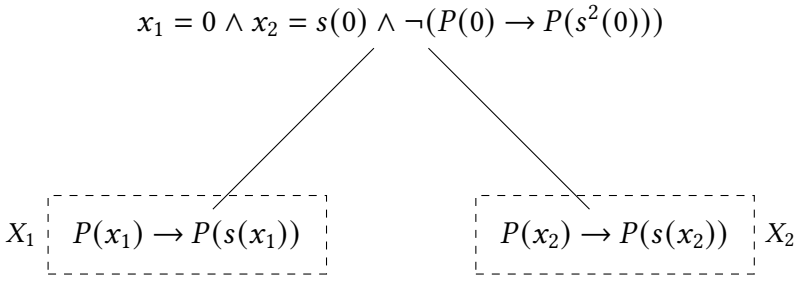
$$\begin{aligned}\exists X_1 \exists X_2 \Big(& \forall x\, ((P(x) \to P(s(x))) \to X_1(x)) \wedge \\ & \forall x\, ((P(x) \to P(s(x))) \to X_2(x)) \wedge \\ & (X_1(0) \wedge X_2(s(0)) \wedge P(0) \to P(s^2(0)))\Big)\end{aligned}$$

For the occurrence $X_1(0)$, we abstract the term 0 into the variable x_1 and

then replace $X_1(0)$ by $X(x_1)$ and add the equality $x_1 = 0$:

$$\begin{aligned}\exists X_1 \exists X_2 \Big(& \forall x_1 \; ((P(x_1) \to P(s(x_1))) \to X_1(x_1)) \wedge \\ & \forall x_2 \; ((P(x_2) \to P(s(x_2))) \to X_2(x_2)) \wedge \\ & \forall x_1 \, \forall x_2 \; (X_1(x_1) \wedge x_1 = 0 \wedge X_2(x_2) \wedge x_2 = s(0) \wedge P(0) \to P(s^2(0)))\Big)\end{aligned}$$

Such a formula equation directly corresponds to a tree. This tree contains one node for every clause in the formula equation, which is labelled with the quantifier-free part of the clause. The conjunction of the formulas in this tree is then unsatisfiable if and only if the formula equation was valid.



By Craig interpolation [24], if we have a proof of $\Gamma, \Pi \vdash \Delta, \Lambda$ then we can compute a formula I and proofs of $\Gamma \vdash \Delta, I$ and $I, \Pi \vdash \Lambda$ in polynomial time where I only contains function and predicate symbols that occur in both $\Gamma \cup \Delta$ and $\Pi \cup \Lambda$. For quantifier-free formulas with equality as a background theory, this interpolant is quantifier-free as well [68].

In order to compute the solution for the predicate variable X_1 we divide the tree into two parts: the part below the X_1 node, and the rest. The only common symbols are the variables of the predicate variable (i.e., x_1), and hence the solution is directly given by the interpolant. The collection of all of these interpolants—one for every subtree—is also known as a *tree interpolant* (a term introduced by the Duality paper [69]).

Solving daglike formula equations. We can now extend the approach to the daglike case by unrolling the formula equation into a treelike one by

duplicating clauses as needed.

$$\exists X \Big(\forall x\ ((P(x) \to P(s(x))) \to X(x)) \wedge \\ (X(0) \wedge X(s(0)) \wedge P(0) \to P(s^2(0)))\Big)$$

The predicate variable has two occurrences on the left side of the last clause, $X(0)$ and $X(s(0))$. Hence we need to make two copies of it:

$$\exists X_1 \exists X_2 \Big(\forall x\ ((P(x) \to P(s(x))) \to X_1(x)) \wedge \\ \forall x\ ((P(x) \to P(s(x))) \to X_2(x)) \wedge \\ (X_1(0) \wedge X_2(s(0)) \wedge P(0) \to P(s^2(0)))\Big)$$

The important part about this unrolling is that we can easily translate solutions of the unrolled problem to solutions of the original daglike problem: if we have a solution φ_1, φ_2 for X_1, X_2, resp., then we get a solution for the original X by taking the conjunction of the solution for the duplicated predicate variables. That is, $\varphi_1 \wedge \varphi_2$ is a solution for the daglike formula equation.

Solving recursive formula equations. Finally, recursive formula equations are handled by unrolling them into non-recursive ones. As before, this is achieved by duplicating predicate variables and clauses. In the Duality solver, this is actually handled in an incremental way so that new clauses are generated on demand and the decision which clauses to generate is driven by the counter-model computed by the SMT solver. This is important to scale to large problems with many predicate variables. For the formula equations induced by induction grammars however there is little to decide, since we only have a single predicate variable and a very small number of clauses.

$$\exists X \Big((P(0) \to X(0)) \wedge \\ \forall x\ ((P(x) \to P(s(x))) \wedge X(x) \to X(s(x))) \wedge \\ (X(c) \to P(c))\Big)$$

Here we duplicate the predicate variable X into the variables $X_1, \dots X_n$:

$$\begin{aligned}\exists X \Big(& (P(0) \to X_1(0)) \wedge \\ & \forall x\ ((P(x) \to P(s(x))) \wedge X_1(x) \to X_2(s(x))) \wedge \\ & \vdots \\ & \forall x\ ((P(x) \to P(s(x))) \wedge X_{n-1}(x) \to X_n(s(x))) \wedge \\ & (X_1(c) \to P(c)) \wedge \dots \wedge (X_n(c) \to P(c)) \Big)\end{aligned}$$

The solution for the unrolled formula equation is then translated back to the original formula equation by combining it in a *disjunction*. In the example, a possible solution would be $P(0) \wedge x = 0, P(x), \dots, P(x)$. We would then get a candidate solution $(P(0) \wedge x = 0) \vee P(x)$ for the original formula equation. In this case, it actually solves the formula equation. But this is of course not true in general.

Constrained Horn clauses for VTRATGs. The attentive reader will have noticed that while the formula equations induced by induction grammars are directly constrained Horn clause problems, the formula equations induced by VTRATGs are not of this form, in general. If there are more than two nonterminal vectors, then the induced formula equation will not be a constrained Horn clause problem (because there will be more than one predicate variable on the right side of a clause). However we can define a different version of the formula equation Φ_G which is Horn:

Definition 5.7.1. Let $G = (N, \Sigma, A, P)$ be a decodable VTRATG with $N = \{A \prec \overline{B_1} \prec \dots \overline{B_n}\}$. We define the formula equation $\Phi'_G = \exists \overline{X}\ \Psi'_G$, where Ψ'_G is the conjunction of the following formulas:

- $\forall \overline{\alpha_1} \dots \forall \overline{\alpha_n} \left(\bigwedge \Gamma_0 \to \bigvee \Delta_0 \vee X_1(\overline{\alpha_n}, \dots, \overline{\alpha_1}) \right)$
- $\forall \overline{\alpha_{i+1}} \dots \forall \overline{\alpha_n} \big(\bigwedge \Gamma_i \wedge \bigwedge_{\bar{t} \in P_{\overline{B_i}}} X_i(\overline{\alpha_n}, \dots, \overline{\alpha_{i+1}}, \bar{t}) \to \bigvee \Delta_i \vee X_{i+1}(\overline{\alpha_n}, \dots, \overline{\alpha_{i+1}}) \big)$ for $1 \leq i \leq n$

where $P_{\overline{C}}$, T_i, and Γ_i are as in Definition 5.2.1.

This definition differs from Definition 5.2.1 only in that there is just a single predicate variable on the right-hand side of the implication.

Lemma 5.7.1. *Let G be a decodable VTRATG. Then $\Psi'_G \to \Psi_G$ is provable, and every T-solution of Φ'_G is a T-solution of Φ_G as well.*

Proof. Observe that Ψ'_G is obtained from Ψ_G by replacing positive subformulas by $\top$ and propositional equivalences. □

Lemma 5.7.2. *Let G be a decodable VTRATG such that there is a production of every nonterminal vector. If $L(G)$ is T-tautological, then $\overline{C^G}$ is a T-solution for Φ'_G.*

Proof. Analogously to the proof of Lemma 5.2.1. □

Theorem 5.7.1. *Let G be a decodable VTRATG such that there is a production of every nonterminal vector. Then Φ_G is T-solvable if and only if Φ'_G is T-solvable.*

Proof. If Φ_G is solvable, then G is tautological by Theorem 5.2.1 and the canonical solution solves Φ'_G by Lemma 5.7.2. On the other hand, if Φ'_G is solvable, then the same solution works for Φ_G by Lemma 5.7.1. □

The formula equation Φ'_G is then a daglike constrained Horn clause problem that can be solved using the interpolation-based approach.

6 Algorithm for proof import

Cut-free proofs have a variety of applications. In the preceding chapters, we have seen one of them: given a cut-free proof in the form of a Herbrand sequent, we can algorithmically introduce structure in the form of cuts or induction inferences. Starting from proofs obtained from automated theorem provers, we can introduce these inferences to structure these automatically generated proofs. In this chapter we will investigate a practical way to get expansion proofs from theorem provers. These expansion trees were introduced in [71] to generalize Herbrand's theorem to higher-order logic in the form of elementary type theory (simple type theory without choice or extensionality). In first-order logic, they provide a technically convenient formalism to store the tautological instances of non-prenex formulas.

There is a generic way to solve the problem of importing proofs by automated theorem provers: since every inference generated by the theorem prover is valid in first-order logic we could translate the proof into the calculus LK (with say, one cut per inference). We could then apply a general-purpose cut-elimination algorithm to this proof and extract an expansion proof from the resulting cut-free proof.

However for our applications we are only interested in the quantifier instance terms as contained in an Herbrand disjunction, or expansion tree. Given that we are only interested in this specific aspect of the proofs, we can hope to devise more efficient proof import procedures that do not produce a full proof including propositional reasoning, but only the expansion proof. In this chapter we will present such an algorithm and evaluate its implementation in GAPT.

Some of the most effective theorem provers in classical first-order logic are based on the resolution and superposition calculi; therefore we aim to import proofs generated by these provers. Algorithms to translate automatically

generated resolution proofs to expansion proofs have already been proposed in the context of the TPS [78] and GAPT [52] systems.

The approach in [52] transforms a resolution refutation first into a sequent calculus proof with atomic cuts, and then extracts an expansion proof in a second step; Skolemization is not handled in the algorithm and is instead treated as a preprocessing step of the formula.

More closely related to the approach in this chapter is the algorithm in [78]: it directly transforms a resolution refutation into an expansion proof. Skolemization is handled explicitly during the translation, and Skolem terms are automatically converted to variables. However this treatment has two unfortunate consequences: first, only outer Skolemization can be used in this way—and this conflicts with the goal to reduce the number of Skolem functions and the number of the arguments, which is necessary for efficient proof search. Additionally, the deskolemization is incompatible with built-in equational inferences which implicitly make use of congruences for Skolem functions.

Both of the approaches require a naive distributivity-based transformation of the input formula into clause normal form (CNF), which can increase the size of the problem exponentially. Moreover, neither approach can handle splitting inferences as used by SPASS [100] or Vampire [99].

The solution we present in this chapter includes clausification and splitting steps as inferences of the resolution proof in a similar way as higher-order resolution calculi [4], conveniently keeping track of introduced subformula definitions for structural clausification and Skolem functions in the same data structure as the resolution refutation.

The algorithm as shown in Algorithm 6 proceeds in three main steps: first we extract an expansion proof of an extended sequent, possible containing cuts. If the resolution proof did not contain subformula definitions or Avatar splitting, then this is already the final expansion proof. To eliminate the cuts corresponding to Avatar splitting, the next step is a cut-elimination on expansion proof. The subformula definitions are then removed in the third and final step. Skolemization remains in the output proof (and could be eliminated afterwards as well, if desired).

In Sections 6.1 and 6.2 we describe the resolution and expansion proof calculi used, respectively. Basic operations on expansion proofs, merge- and

Algorithm 6 Transforming expansion proofs to resolution proofs

```
function Transform(π)
E1 ← Extract(π)                          (see Section 6.3)
E2 ← EliminateCuts(E1)                   (see Lemma 6.2.2)
E3 ← EliminateDefinitions(E2)            (see Lemmas 6.4.1 and 6.4.2)
return E3
```

cut-reduction, are introduced in Section 6.2 as well. Section 6.3 then explains the extraction of expansion proofs from resolution proofs. The subformula definitions are eliminated in Section 6.4. In Section 6.6 we then evaluate the performance and clausification quality of an implementation of this algorithm. At the end in Section 6.7, we revisit the splitting inferences and show how to eliminate them directly from resolution proofs.

6.1 Resolution proofs

As a calculus for resolution proofs we consider a first-order variant of Andrew's system $\mathcal{R}$ [4], extended with inferences to support subformula definitions and Avatar splitting. This system $\mathcal{R}$ is a system for reasoning in higher-order logic, we use it in first-order logic for practical reasons: inference rules for clausification are included as part of the calculus. If we were to consider a first-order calculus where every inference step ended in a clause, then we would need to keep track of an additional data structure that describes how the clauses are derived from the input formula given to the prover. With the calculus considered in this section, this information is part of the proof.

The inferences in this calculus as shown in Figure 6.1 operate on sequents, which are pairs of multisets of formulas. Let φ be the closed formula that is to be proven, a resolution proof then starts with the sequent $\varphi \vdash$ (using the Input inference) and ends with the empty sequent $\vdash$.

Resolution proofs of first-order formulas can typically be roughly divided into an upper part starting with $\varphi \vdash$ that performs clausification inferences applied during preprocessing, and a lower part that ends with $\vdash$ and corresponds to the relevant steps taken during the actual proof search with resolution,

Clausification inferences:

$$\frac{}{\varphi \vdash}\ \text{Input} \qquad \frac{\top, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \text{TopL} \qquad \frac{\Gamma \vdash \Delta, \bot}{\Gamma \vdash \Delta}\ \text{BotR}$$

$$\frac{\neg\varphi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \varphi}\ \text{NegL} \qquad \frac{\Gamma \vdash \Delta, \neg\varphi}{\varphi, \Gamma \vdash \Delta}\ \text{NegR}$$

$$\frac{\varphi \wedge \psi, \Gamma \vdash \Delta}{\varphi, \psi, \Gamma \vdash \Delta}\ \text{AndL} \qquad \frac{\Gamma \vdash \Delta, \varphi \wedge \psi}{\Gamma \vdash \Delta, \varphi}\ \text{AndR1} \qquad \frac{\Gamma \vdash \Delta, \varphi \wedge \psi}{\Gamma \vdash \Delta, \psi}\ \text{AndR2}$$

$$\frac{\Gamma \vdash \Delta, \varphi \vee \psi}{\Gamma \vdash \Delta, \varphi, \psi}\ \text{OrR} \qquad \frac{\varphi \vee \psi, \Gamma \vdash \Delta}{\varphi, \Gamma \vdash \Delta}\ \text{OrL1} \qquad \frac{\varphi \vee \psi, \Gamma \vdash \Delta}{\psi, \Gamma \vdash \Delta}\ \text{OrL2}$$

$$\frac{\Gamma \vdash \Delta, \varphi \rightarrow \psi}{\varphi, \Gamma \vdash \Delta, \psi}\ \text{ImpR} \qquad \frac{\varphi \rightarrow \psi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \varphi}\ \text{ImpL1} \qquad \frac{\varphi \rightarrow \psi, \Gamma \vdash \Delta}{\psi, \Gamma \vdash \Delta}\ \text{ImpL2}$$

$$\frac{\Gamma \vdash \Delta, \forall x\, \varphi}{\Gamma \vdash \Delta, \varphi}\ \text{AllR} \qquad \frac{\forall x\, \varphi, \Gamma \vdash \Delta}{\varphi[x\backslash t], \Gamma \vdash \Delta}\ \text{AllL (where } t \overset{\text{sk}}{\mapsto} \forall x\, \varphi)$$

$$\frac{\exists x\, \varphi, \Gamma \vdash \Delta}{\varphi, \Gamma \vdash \Delta}\ \text{ExL} \qquad \frac{\Gamma \vdash \Delta, \exists x\, \varphi}{\Gamma \vdash \Delta, \varphi[x\backslash t]}\ \text{ExR (where } t \overset{\text{sk}}{\mapsto} \exists x\, \varphi)$$

Subformula definition inferences:

$$\frac{\varphi, \Gamma \vdash \Delta}{D(\bar{t}), \Gamma \vdash \Delta}\ \text{AbbrL} \qquad \frac{\Gamma \vdash \Delta, \varphi}{\Gamma \vdash \Delta, D(\bar{t})}\ \text{AbbrR} \qquad (\text{where } D(\bar{t}) \overset{\text{def}}{\mapsto} \varphi)$$

$$\frac{}{\varphi \vdash D(\bar{t})}\ \text{DefL} \qquad \frac{}{D(\bar{t}) \vdash \varphi}\ \text{DefR} \qquad (\text{where } D(\bar{t}) \overset{\text{def}}{\mapsto} \varphi)$$

Logical inferences:

$$\frac{\Gamma \vdash \Delta, A \qquad A, \Pi \vdash \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\ \text{Res} \qquad \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\ \text{Factor} \qquad \frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\ \text{Factor}$$

$$\frac{}{A \vdash A}\ \text{Taut} \qquad \frac{\Gamma \vdash \Delta}{\Gamma\sigma \vdash \Delta\sigma}\ \text{Subst} \qquad \frac{}{\vdash t = t}\ \text{Refl}$$

$$\frac{\Gamma \vdash \Delta, t = s \qquad A[t], \Pi \vdash \Lambda}{A[s], \Gamma, \Pi \vdash \Delta, \Lambda}\ \text{Rw} \qquad \frac{\Gamma \vdash \Delta, t = s \qquad A[s], \Pi \vdash \Lambda}{A[t], \Gamma, \Pi \vdash \Delta, \Lambda}\ \text{Rw}$$

$$\frac{\Gamma \vdash \Delta, t = s \qquad \Pi \vdash \Lambda, A[t]}{\Gamma, \Pi \vdash \Delta, \Lambda, A[s]}\ \text{Rw} \qquad \frac{\Gamma \vdash \Delta, t = s \qquad \Pi \vdash \Lambda, A[s]}{\Gamma, \Pi \vdash \Delta, \Lambda, A[t]}\ \text{Rw}$$

Avatar splitting inference:

$$\frac{\mathcal{S}_1 \mathbin{+\!\!+} \mathcal{S}_2}{\mathcal{S}_1 \mathbin{:\!+} D}\ \text{AvSplit} \qquad \frac{}{D \mathbin{+\!:} \mathcal{S}_2}\ \text{AvIntro}$$

$$(\text{where } D \overset{\text{def}}{\mapsto} \forall \overline{x}\, \mathcal{S}_2 \text{ and } \mathrm{FV}(\mathcal{S}_1) \cap \mathrm{FV}(\mathcal{S}_2) = \emptyset)$$

Figure 6.1: Calculus for resolution proofs.

rewriting, factoring, and other inferences. There is no implicit unification in the inferences, substitution is handled explicitly using an extra inference.

Both Skolemization and subformula definition inferences make use of global dictionaries that store the interpretation of the defined atoms and Skolem functions, respectively. Defined atoms and Skolem functions are fresh symbols that do not occur in the problem. For definitions, the entry $D(\overline{x}) \overset{\text{def}}{\mapsto} \psi$ means that the atom $D(\overline{x})$ is defined to be ψ, that is, $\forall \overline{x}\ (D(\overline{x}) \leftrightarrow \psi)$ is assumed. For Skolem functions, the entry $s(\overline{x}) \overset{\text{sk}}{\mapsto} \forall y\ \varphi[y]$ means that $s(\overline{x})$ is the Skolem term used to instantiate $\forall y\ \varphi[y]$, that is, $\forall \overline{x}\ (\varphi[s(\overline{x})] \rightarrow \forall y\ \varphi[y])$ is assumed. We use both relations also for substitution instances of the definitions, i.e. we write $D(\overline{t}) \overset{\text{def}}{\mapsto} \psi[\overline{x}\backslash\overline{t}]$ as well as $s(\overline{t}) \overset{\text{sk}}{\mapsto} (\forall y\ \varphi[y])[\overline{x}\backslash\overline{t}]$. Furthermore, we require these definitions to be acyclic.

The AvSplit and AvIntro inferences here are the minimal version necessary to represent Avatar splitting [99] in the resulting proofs. This calculus augments the clauses derived in the theorem prover with a finite set of so-called assertions. Splitting in general is based on the observation that if the literals of a clause can be partitioned into two parts with disjoint variables, then the clause is equivalent to a disjunction of two clauses. For example:

$$\forall x\ \forall y\ (P(x) \vee Q(y)) \quad \leftrightarrow \quad (\forall x\ P(x)) \vee (\forall y\ Q(y))$$

A minimal nonempty subset of the literals whose variables are disjoint from the rest of the clause is called a component. It is typically beneficial to keep the number of literals in a clause small. So if we derive such a splittable clause, then we could consider two cases: first $\forall x\ P(x)$, and then $\forall y\ Q(y)$. In this way, we would subdivide the proof search like a tree if we had multiple clauses to split. (This is essentially the approach taken by the SPASS prover [101].) The Avatar approach in contrast lets a SAT solver determine which of these cases to consider at any moment. The assertions that are added to the clauses hence indicate to which case they belong. Let us consider what happens when the A-clause $P(x) \vee Q(y) \ \leftarrow\ C_1, C_2$ (which already has two assertions) is split[1]:

$$\frac{P(x) \vee Q(y) \ \leftarrow\ C_1, C_2}{P(x) \ \leftarrow\ C_3 \qquad Q(y) \ \leftarrow\ C_4 \qquad \neg[C_1] \vee \neg[C_2] \vee [C_3] \vee [C_4]}$$

[1]The precise details may be subtly different in different implementations.

Two new assertions are introduced: C_3 and C_4. The prover derives two first-order clauses that it keeps in the saturation loop. The first one is $P(x) \leftarrow C_3$. We can read this as: if C_3, then $\forall x P(x)$. It is useful to think of the assertion C_3 as an abbreviation for $\forall x\, P(x)$. The third derived clause is not used for saturation but is passed to the SAT solver. The syntax $[C_1]$ refers to the propositional atom associated with the component C_1. The clause for the SAT solver is equivalent to the original clause if we replace C_3 by $\forall x\, P(x)$ and C_4 by $\forall y\, Q(y)$.

The distinction between the clause and its assertions is only relevant for proof search. Inferences (such as resolution or superposition) are only performed on the clause, but not on the assertions. However for proof transformations, such as the one in this chapter, this distinction is irrelevant: here we represent A-clauses by combining the assertion and the clause part into a single clause $[C_1], \ldots, [C_k], a_1, \ldots, a_m \vdash b_1, \ldots b_n$.

We define prepending an element to the antecedent as $\varphi +\!: (\Gamma \vdash \Delta) = (\Gamma \cup \{\varphi\} \vdash \Delta)$, appending an element to the succedent as $(\Gamma \vdash \Delta) :\!+ \varphi = (\Gamma \vdash \Delta \cup \{\varphi\})$, and concatenating sequents as $(\Gamma \vdash \Delta) +\!+ (\Pi \vdash \Lambda) = (\Gamma \cup \Pi) \vdash (\Delta \cup \Lambda)$.

During proof search, splitting takes a clause $\mathcal{S}_1 +\!+ \mathcal{S}_2$ that can be partitioned into two (or more) clauses $\mathcal{S}_1$ and $\mathcal{S}_2$ with pairwise disjoint free variables, and splits it into three clauses: $D_1 +\!: \mathcal{S}_1$ and $D_2 +\!: \mathcal{S}_2$ plus an additional clause $\vdash D_1, D_2$. (This third clause is ground. During proof search it is directly passed to the SAT solver, and is not considered by the first-order reasoning part of the prover.) In our calculus we represent this step as follows:

$$\dfrac{\dfrac{\mathcal{S}_1 +\!+ \mathcal{S}_2}{\mathcal{S}_1 :\!+ D_2}\,\text{AvSplit}}{\vdash D_1, D_2}\,\text{AvSplit} \qquad \dfrac{}{D_i +\!: \mathcal{S}_i}\,\text{AvIntro}$$

Other splitting schemes such as the one implemented in SPASS [100] can be simulated using these inferences as well. The splitting dependencies a clause implicitly depends on are simply translated to explicit splitting atoms. For each split we then take the resulting proofs of the two branches and resolve on the opposing splitting atoms.

We assume a few minor technical restrictions on resolution proofs: there are no AbbrL inferences below Rw, Refl, and Taut, and DefR with the same defined atom. These are not particularly large restrictions, as they simply reflect the

fact that clausification typically happens as a preprocessing step before proof search. It is also important to note that we only allow subformula definitions in a single polarity here—if we want to abbreviate a formula φ that occurs on both sequent sides, then we need to introduce a different D_i atom for each side. This is a larger restriction since it precludes certain kinds of preprocessing: for example such definitions would occur if we wanted to abbreviate a subformula of the input formula that occurs multiple times (and in both polarities) by the same atom.

Example 6.1.1. The following is natural proof of a variant of the Drinker's formula. We label the subproofs on the left-hand side for later reference.

(π_0) $\exists x \forall y\ (P(x) \to P(y)) \vdash$ (Input)

(π_1) $\forall y\ (P(x) \to P(y)) \vdash$ (ExL(π_0))

(π_2) $P(x) \to P(s(x)) \vdash$ (AllL(π_1))

(π_3) $\vdash P(x)$ (ImpL(π_2))

(π_4) $P(s(x)) \vdash$ (ImpR(π_2))

(π_5) $\vdash P(s(x))$ (Subst(π_3))

(π_6) $\vdash$ (Res(π_4, π_5))

6.2 Expansion proofs

The proof formalism of expansion trees was introduced in [71] to describe Herbrand disjunctions in higher-order logic. We use them in first-order logic as well, since they are an elegant and convenient data structure. The central idea is that each expansion tree E comes with a *shallow formula* $\mathrm{sh}(E)$ and a quantifier-free *deep formula* $\mathrm{dp}(E)$. The deep formula corresponds to the Herbrand disjunction, the shallow formula is the quantified formula. If the deep formula is a quasi-tautology (a tautology modulo equality), then the shallow formula is valid.

Expansion trees have two polarities, $-$ and $+$. We write $-p$ for the inverse polarity of p, i.e. $--\ =\ +$ and $-+\ =\ -$. Polarity only changes on the left

side of the connective $\rightarrow$ and inside the connective $\neg$. This distinction is important since we must instantiate positive occurrences of $\forall$ (resp. negative occurrences of $\exists$, called "strong quantifiers") with an eigenvariable or Skolem term, while we can instantiate the negative ones with whatever terms we want ("weak quantifiers"). An atom is a predicate such as $P(x, y)$ or an equality; the formulas $\top, \bot$ are not atoms. For example, consider the case that we want to prove the formula $(\forall x\ \varphi) \wedge \neg(\forall x\ \varphi)$. Here, the first universal quantifier has positive polarity, and the second one negative polarity.

We base the development on [54] and include several extensions here: first, there are subformula definition nodes to represent the AbbrL inferences in the resolution calculus. We also add cut nodes as in [54], these cuts are similar to cuts in a sequent calculus and will be used to represent Avatar splitting inferences. Finally we also add Skolem nodes to represent the Skolemization steps.

Definition 6.2.1. We inductively define the set $\mathrm{ET}^p(\varphi)$ of *expansion trees* with polarity $p \in \{+, -\}$ and *shallow formula* φ:

$$\frac{\varphi \text{ formula}}{\mathrm{wk}^p(\varphi) \in \mathrm{ET}^p(\varphi)} \qquad \frac{E_1 \in \mathrm{ET}^p(\varphi) \qquad E_2 \in \mathrm{ET}^p(\varphi)}{E_1 \sqcup E_2 \in \mathrm{ET}^p(\varphi)}$$

$$\frac{A \text{ atom}/\top/\bot}{A^p \in \mathrm{ET}^p(A)} \qquad \frac{E \in \mathrm{ET}^p(\varphi)}{\neg E \in \mathrm{ET}^{-p}(\neg\varphi)} \qquad \frac{E_1 \in \mathrm{ET}^p(\varphi) \qquad E_2 \in \mathrm{ET}^p(\psi)}{E_1 \wedge E_2 \in \mathrm{ET}^p(\varphi \wedge \psi)}$$

$$\frac{E_1 \in \mathrm{ET}^p(\varphi) \qquad E_2 \in \mathrm{ET}^p(\psi)}{E_1 \wedge E_2 \in \mathrm{ET}^p(\varphi \vee \psi)} \qquad \frac{E_1 \in \mathrm{ET}^{-p}(\varphi) \qquad E_2 \in \mathrm{ET}^p(\psi)}{E_1 \wedge E_2 \in \mathrm{ET}^p(\varphi \rightarrow \psi)}$$

$$\frac{D(\bar{t}) \overset{\mathrm{def}}{\mapsto} \varphi}{\varphi +_{\mathrm{def}} D^p(\bar{t}) \ \in\ \mathrm{ET}^p(\varphi)} \qquad \frac{E_1 \in \mathrm{ET}^+(\varphi) \qquad E_2 \in \mathrm{ET}^-(\varphi)}{\mathrm{Cut}(E_1, E_2) \in \mathrm{ET}^-(\top)}$$

$$\frac{E \in \mathrm{ET}^+(\varphi[x\backslash y])}{\forall x\ \varphi +^{y}_{\mathrm{ev}} E \ \in\ \mathrm{ET}^+(\forall x\ \varphi)} \qquad \frac{E \in \mathrm{ET}^+(\varphi[x\backslash t]) \qquad t \overset{\mathrm{sk}}{\mapsto} \forall x\ \varphi}{\forall x\ \varphi +^{t}_{\mathrm{sk}} E \ \in\ \mathrm{ET}^+(\forall x\ \varphi)}$$

$$\frac{E_1 \in \mathrm{ET}^-(\varphi[x\backslash t_1]) \qquad \cdots \qquad E_n \in \mathrm{ET}^-(\varphi[x\backslash t_n])}{\forall x\ \varphi +^{t_1} E_1 \cdots +^{t_n} E_n \ \in\ \mathrm{ET}^-(\forall x\ \varphi)}$$

$$\frac{E \in \mathrm{ET}^-(\varphi[x\backslash y])}{\exists x\ \varphi +^{y}_{\mathrm{ev}} E \ \in\ \mathrm{ET}^-(\exists x\ \varphi)} \qquad \frac{E \in \mathrm{ET}^-(\varphi[x\backslash t]) \qquad t \overset{\mathrm{sk}}{\mapsto} \exists x\ \varphi}{\exists x\ \varphi +^{t}_{\mathrm{sk}} E \ \in\ \mathrm{ET}^-(\exists x\ \varphi)}$$

$$\frac{E_1 \in \mathrm{ET}^+(\varphi[x\backslash t_1]) \qquad \cdots \qquad E_n \in \mathrm{ET}^+(\varphi[x\backslash t_n])}{\exists x\ \varphi +^{t_1} E_1 \cdots +^{t_n} E_n \ \in\ \mathrm{ET}^+(\exists x\ \varphi)}$$

The tree $\mathrm{wk}^p(\varphi)$ is called "weakening", $E_1 \sqcup E_2$ is called merge, $\forall x\, \varphi +^y_{\mathrm{ev}} E \in \mathrm{ET}^+(\forall x\, \varphi)$ or $\exists x\, \varphi +^y_{\mathrm{ev}} E$ is an eigenvariable node, and $\forall x\, \varphi +^t_{\mathrm{sk}} E \in \mathrm{ET}^+(\forall x\, \varphi)$ or $\exists x \varphi +^y_{\mathrm{sk}} E$ is a Skolem node. Each expansion tree E has a uniquely determined shallow formula and polarity, we write $\mathrm{sh}(E)$ for its shallow formula, and $\mathrm{pol}(E)$ for its polarity. The *size* $|E|$ of an expansion tree is the number of its leaves counted as in a tree. Given an expansion tree $E = \forall x \varphi +^y_{\mathrm{ev}} E'$ or $E = \exists x \varphi +^y_{\mathrm{ev}} E'$, we say that y is the eigenvariable of E. The set $\mathrm{EV}(E)$ contains all eigenvariables of subtrees in E, including E. We also use the notation for blocks of quantifiers with expansion trees, that is, we write $\forall \overline{x}\ \varphi +^{\overline{x}}_{\mathrm{ev}} E$ as an abbreviation for $\forall \overline{x}\ \varphi +^{x_1}_{\mathrm{ev}} \cdots +^{x_n}_{\mathrm{ev}} E$.

Example 6.2.1. The following expansion tree $E \in \mathrm{ET}^+(\exists x \forall y\ (P(x) \to P(y)))$ has our variant of the Drinker's formula as shallow formula:

$$\begin{aligned}
&\exists x \forall y\ (P(x) \to P(y)) \\
&+^x \forall y\ (P(x) \to P(y)) +^{s(x)}_{\mathrm{sk}} (\mathrm{wk}^-(P(x)) \to P(s(x))^+) \\
&+^{s(x)} \forall y\ (P(s(x)) \to P(y)) +^{s(s(x))}_{\mathrm{sk}} (P(s(x))^- \to \mathrm{wk}^+(P(s(s(x)))))
\end{aligned}$$

While the shallow formula describes the quantified formula to be proven, the deep formula is a quantifier-free formula corresponding to the Herbrand disjunction:

Definition 6.2.2. Let E be an expansion tree, we define the *deep formula* $\mathrm{dp}(E)$ recursively as follows:

$$\mathrm{dp}(\mathrm{wk}^+(\varphi)) = \bot, \qquad \mathrm{dp}(\mathrm{wk}^-(\varphi)) = \top$$

$$\mathrm{dp}(E_1 \sqcup E_2) = \mathrm{dp}(E_1) \vee \mathrm{dp}(E_2) \quad \text{if } \mathrm{pol}(E_1 \sqcup E_2) = +$$

$$\mathrm{dp}(E_1 \sqcup E_2) = \mathrm{dp}(E_1) \wedge \mathrm{dp}(E_2) \quad \text{if } \mathrm{pol}(E_1 \sqcup E_2) = -$$

$$\mathrm{dp}(A^p) = A, \quad \mathrm{dp}(\neg E) = \neg \mathrm{dp}(E), \quad \mathrm{dp}(E_1 \wedge E_2) = \mathrm{dp}(E_1) \wedge \mathrm{dp}(E_2)$$

$$\mathrm{dp}(E_1 \vee E_2) = \mathrm{dp}(E_1) \vee \mathrm{dp}(E_2), \quad \mathrm{dp}(E_1 \to E_2) = \mathrm{dp}(E_1) \to \mathrm{dp}(E_2)$$

$$\mathrm{dp}(\varphi +_{\mathrm{def}} D^p(\overline{t})) = D^p(\overline{t})$$

$$\mathrm{dp}(\forall x\ \varphi +^y_{\mathrm{ev}} E) = \mathrm{dp}(E), \quad \mathrm{dp}(\forall x\ \varphi +^t_{\mathrm{sk}} E) = \mathrm{dp}(E)$$

$$\mathrm{dp}(\forall x\ \varphi +^{t_1} E_1 \cdots +^{t_n} E_n) = \mathrm{dp}(E_1) \wedge \cdots \wedge \mathrm{dp}(E_n)$$

$$\mathrm{dp}(\exists x\ \varphi +^{y}_{\mathrm{ev}} E) = \mathrm{dp}(E), \quad \mathrm{dp}(\exists x\ \varphi +^{t}_{\mathrm{sk}} E) = \mathrm{dp}(E)$$
$$\mathrm{dp}(\exists x\ \varphi +^{t_1} E_1 \cdots +^{t_n} E_n) = \mathrm{dp}(E_1) \wedge \cdots \wedge \mathrm{dp}(E_n)$$

Example 6.2.2. The expansion tree E in Example 6.2.1 has the following tautological deep formula: $\mathrm{dp}(E) = (\top \to P(s(x))) \vee (P(s(x)) \to \bot)$

Eigenvariable nodes add a small technical complication to the definition of an expansion proof. (But we need them since they arise during the extraction of splitting inferences.) Not only is it necessary that the deep formula is quasi-tautological, but the eigenvariables also need to be acyclic in a certain sense (this is a similar restriction to the eigenvariable condition in sequent calculi). We formalize this acyclicity using a dependency relation, which we will require to be acyclic:

Definition 6.2.3. Let E be an expansion tree. The dependency relation $<_E$ is a binary relation on variables where $x <_E y$ iff E contains a subtree E' such that $x \in \mathrm{FV}(\mathrm{sh}(E'))$ and $y \in \mathrm{EV}(E')$.

Definition 6.2.4. An *expansion sequent* $\mathcal{E}$ is a sequent of expansion trees. Its dependency relation $<_{\mathcal{E}} = \bigcup_{E \in \mathcal{E}} <_E$ is the union of the dependency relations of its trees. Its shallow sequent and deep sequent consists of the shallow and deep formulas of its expansion trees, respectively.

An expansion sequent is *positive(negative)* iff the trees in the succedent have positive(negative) polarity and the trees in the antecedent have negative(positive) polarity. The *size* $|\mathcal{E}|$ of an expansion sequent is the sum of the sizes of its expansion trees.

Definition 6.2.5. An *expansion proof* $\mathcal{E}$ is a positive expansion sequent such that:

1. $<_{\mathcal{E}}$ is acyclic (i.e., can be extended to a linear order) and there are no duplicate eigenvariables,
2. $\mathrm{dp}(\mathcal{E})$ is a quasi-tautology

Example 6.2.3. Let E be as in Example 6.2.1. The sequent $\mathrm{dp}(\vdash E)$ is a tautology, and the dependency relation of $\vdash E$ is empty and hence acyclic. So $\vdash E$ is an expansion proof.

Expansion proofs are sound and complete for first-order logic. That is, if $\mathcal{S}$ is a sequent, then there exists an expansion proof $\mathcal{E}$ with $\mathcal{S}$ as the shallow sequent if and only if $\mathcal{S}$ is valid. Completeness follows easily from cut-free completeness of the sequent calculus LK: we can recursively assign to every proof π in LK with end-sequent $\mathcal{S}$ an expansion proof $\mathcal{E}$ such that $\mathrm{sh}(\mathcal{E}) = \mathcal{S}$ (as done for example in [71]).

Given a substitution σ that maps eigenvariables to variables, we can apply it to expansion trees and expansion sequents, written $E\sigma$ and $\mathcal{E}\sigma$, respectively. The calculus for expansion proofs we presented is redundant: it is still complete even without the $\sqcup$, $+_{\mathrm{def}}$, and Cut nodes. We also only need either the eigenvariable or Skolem nodes.

Two important reductions were introduced in [54] that will eliminate the $\sqcup$ and Cut nodes. The elimination of $+_{\mathrm{def}}$ will be discussed in Section 6.4.

The relation $\overset{\sqcup}{\rightsquigarrow}$ pushes merge nodes to the leaves of an expansion tree until they disappear, for example $(E_1 \wedge E_2) \sqcup (E_3 \wedge E_4) \overset{\sqcup}{\rightsquigarrow} (E_1 \sqcup E_3) \wedge (E_2 \sqcup E_4)$ reduces merges on conjunctions.

Lemma 6.2.1. *The relation $\overset{\sqcup}{\rightsquigarrow}$ on expansion sequents has the following properties:*

1. *$\overset{\sqcup}{\rightsquigarrow}$ is terminating.*
2. *Whenever $\mathcal{E} \overset{\sqcup}{\rightsquigarrow} \mathcal{E}'$, then $\mathrm{dp}(\mathcal{E}) \to \mathrm{dp}(\mathcal{E}')$ is a tautology.*
3. *$\overset{\sqcup}{\rightsquigarrow}$ preserves acyclicity of the dependency relation.*
4. *$\overset{\sqcup}{\rightsquigarrow}$ preserves polarity and the shallow formula.*
5. *If $\mathcal{E}$ does not contain Skolem or definition nodes, then its $\overset{\sqcup}{\rightsquigarrow}$-normal forms do not contain merge nodes.*

In particular the $\overset{\sqcup}{\rightsquigarrow}$-normal forms of expansion proofs without Skolem or definition nodes are merge-free expansion proofs.

Proof. Analogous to Lemmas 12 and 13 in [54]. Note that merge reduction can get stuck on merges of two Skolem nodes with different Skolem terms, or two definition nodes with different definitions, hence the condition in point 5. □

Cuts can be reduced and eventually eliminated as well. The cut-reduction relation $\overset{\text{cut}}{\rightsquigarrow}$ (written as $\mapsto$ in [54]) extends merge-reduction and reduces quantified cuts via substitution.

Lemma 6.2.2. *$\overset{\text{cut}}{\rightsquigarrow}$ preserves quasi-tautology of the deep formula, and is weakly normalizing. If no definition or Skolem nodes appear in cuts, then the $\overset{\text{cut}}{\rightsquigarrow}$-normal forms are cut-free.*

Proof. See Lemma 16 and Theorem 33 in [54]. Just as in Lemma 6.2.1, cut-reduction can get stuck on cuts with Skolem nodes such as $\text{Cut}(\forall x\ \varphi +^{x}_{\text{sk}} \ldots, \forall x\ \varphi +^{t} \ldots)$. □

The following lemma will bound the complexity introduced by Avatar splitting inferences, since they will be translated to cuts on formulas of the form $\forall \overline{x}\, C(\overline{x})$ where $C(\overline{x})$ is a clause.

Lemma 6.2.3. *Let $|\mathcal{E}|$ be an expansion proof with n cuts such that all cut formulas are universally quantified closed prenex formulas, and let $\mathcal{E} \overset{\text{cut}^*}{\rightsquigarrow} \mathcal{E}^*$ such that $\mathcal{E}^*$ is cut-free. Then $|\mathcal{E}^*| \leq |\mathcal{E}|^{2^n}$.*

Proof. Merge reduction and propositional reduction steps reduce the size of the expansion proof and keep the number of cuts constants. Each quantifier reduction step decreases the number of cuts by one and increases the size of the proof at most quadratically: essentially we duplicate the proof m times, where m is the number of weak quantifier term blocks in the cut—and m is less than the size of the expansion proof. □

6.3 Extraction

In this section we convert a resolution proof of a formula φ into an expansion proof with a shallow sequent of the following form:

$$\forall \overline{x}\ (\varphi_1(\overline{x}) \to D_1(\overline{x})), \cdots, \forall \overline{x}\ (\varphi_n(\overline{x}) \to D_n(\overline{x})),$$
$$\forall \overline{x}\ (D_{n+1}(\overline{x}) \to \varphi_{n+1}(\overline{x})), \cdots, \forall \overline{x}\ (D_m(\overline{x}) \to \varphi_m(\overline{x})) \vdash \varphi$$

That is, the expansions in the antecedent describe subformula definitions and have definition axioms $\forall \overline{x}\ (\varphi(\overline{x}) \to D(\overline{x}))$ or $\forall \overline{x}\ (D(\overline{x}) \to \varphi(\overline{x}))$ as shallow

formulas where $D(\overline{x}) \overset{\text{def}}{\mapsto} \varphi(\overline{x})$. Furthermore the resulting expansion proof will not have any eigenvariable nodes. We will eliminate the additional expansion trees for definitions in Section 6.4.

The extraction proceeds bottom-up, starting from the proof ending in the empty clause, propagating expansions trees upward until they are at the Input-rules. At every point, each subproof is assigned a finite set of expansion sequents. Formally, we describe the extraction as a binary relation on these assignments of expansion sequents, called extraction states:

Definition 6.3.1. An extraction state $\mathcal{P}; \mathcal{S}$ is a pair consisting of a set $\mathcal{P}$ and an expansion sequent $\mathcal{S}$. Each element of $\mathcal{P}$ is a pair $(\pi, \mathcal{E})$, where π is a subproof ending in $\mathcal{T}$ and $\mathcal{E}$ is a negative expansion sequent such that there exists a substitution σ with $\mathcal{T}\sigma = \mathrm{sh}(\mathcal{E})$. The *deep formula of the extraction state* is defined as:

$$\mathrm{dp}(\mathcal{P}; \mathcal{S}) = \left(\bigwedge_{(\pi,\mathcal{E})\in\mathcal{P}} \mathrm{dp}(\mathcal{E}) \right) \to \mathrm{dp}(\mathcal{S})$$

Note that the trees of the *negative* expansion sequents here have the opposite polarity as in the expansion sequents in Section 6.2. This is due to the fact that resolution proofs are proofs by refutation. The resolution proof starts with $\varphi \vdash$ while the expansion proof has $\vdash \varphi$ as the shallow sequent—observe that φ occurs in opposite polarities here.

Given a resolution proof π, the *initial extraction state* is $C_\pi = (\{(\pi, \vdash)\}; \vdash)$. We can now define a relation $\rightsquigarrow$ on extraction states, that transforms the initial extraction state into extraction state of the form $\emptyset; \mathcal{E}$, where $\mathcal{E}$ is the desired expansion proof. This relation preserves the quasi-tautologyhood of the deep formula—we will prove this crucial property in Lemma 6.3.2. The relation $\rightsquigarrow$ is the smallest relation containing the following cases:

$$\begin{aligned}
\mathcal{P}, (\pi, \mathcal{E}_1), (\pi, \mathcal{E}_2); \mathcal{S} &\rightsquigarrow \mathcal{P}, (\pi, \mathcal{E}_1 \sqcup \mathcal{E}_2); \mathcal{S} \quad \text{if } \mathrm{sh}(\mathcal{E}_1) = \mathrm{sh}(\mathcal{E}_2) \\
\mathcal{P}; E_1 +\!\!: E_2 +\!\!: \mathcal{S} &\rightsquigarrow \mathcal{P}; E_1 \sqcup E_2 +\!\!: \mathcal{S} \quad \text{if } \mathrm{sh}(E_1) = \mathrm{sh}(E_2) \\
\mathcal{P}; \mathcal{S} :\!+ E_1 :\!+ E_2 &\rightsquigarrow \mathcal{P}; \mathcal{S} :\!+ E_1 \sqcup E_2 \quad \text{if } \mathrm{sh}(E_1) = \mathrm{sh}(E_2)
\end{aligned}$$

Logical connectives:

$$\mathcal{P}, (\text{Input}, E \vdash); \mathcal{S} \rightsquigarrow \mathcal{P}; \mathcal{S} :+ E$$
$$\mathcal{P}, (\text{TopL}(\pi), \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \top^+ +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{BotR}(\pi), \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ \bot^-); \mathcal{S}$$
$$\mathcal{P}, (\text{NegL}(\pi), \mathcal{E} :+ E_1); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \neg E_1 +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{NegR}(\pi), E_1 +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ \neg E_1); \mathcal{S}$$
$$\mathcal{P}, (\text{AndL}(\pi), E_1 +: E_2 +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (E_1 \wedge E_2) :+ \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{AndR1}(\pi), \mathcal{E} :+ E_1); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ (E_1 \wedge \text{wk}^-(\dots))); \mathcal{S}$$
$$\mathcal{P}, (\text{AndR2}(\pi), \mathcal{E} :+ E_2); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ (\text{wk}^-(\dots) \wedge E_2)); \mathcal{S}$$
$$\mathcal{P}, (\text{OrR}(\pi), \mathcal{E} :+ E_1 :+ E_2); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ (E_1 \vee E_2)); \mathcal{S}$$
$$\mathcal{P}, (\text{OrL1}(\pi), E_1 +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (E_1 \vee \text{wk}^+(\dots)) +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{OrL2}(\pi), E_2 +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (\text{wk}^+(\dots) \vee E_2) +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{ImpR}(\pi), E_1 +: \mathcal{E} :+ E_2); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ (E_1 \to E_2)); \mathcal{S}$$
$$\mathcal{P}, (\text{ImpL1}(\pi), \mathcal{E} :+ E_1); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (E_1 \to \text{wk}^+(\dots)) +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{ImpL2}(\pi), E_2 +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (\text{wk}^-(\dots) \to E_2) +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{AllR}(\pi), \mathcal{E} :+ E_1); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ (\forall x\ \varphi +^{x\sigma} E_1)); \mathcal{S}$$
$$\mathcal{P}, (\text{AllL}(\pi), E_1 +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (\forall x\ \varphi +^{t}_{\text{sk}} E_1) +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{ExL}(\pi), E_1 +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (\exists x\ \varphi +^{x\sigma} E_1) +: \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\text{ExR}(\pi), \mathcal{E} :+ E_1); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ (\exists x\ \varphi +^{t}_{\text{sk}} E_1)); \mathcal{S}$$

(where $\varphi\sigma = \text{sh}(E_1)$)

Subformula definitions:

$$\mathcal{P}, (\text{AbbrL}(\pi), D^+(\bar{t}) +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, (\varphi +_{\text{def}} D^+(\bar{t})) +: \mathcal{E})); \mathcal{S}$$
$$\mathcal{P}, (\text{AbbrR}(\pi), \mathcal{E} :+ D^-(\bar{t})); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ (\varphi +_{\text{def}} D^-(\bar{t}))); \mathcal{S}$$
$$\mathcal{P}, (\text{DefR}(D(\bar{t})), E_D \vdash E_\varphi); \mathcal{S} \rightsquigarrow \mathcal{P}; \forall \bar{x}(D(\bar{x}) \to \varphi) +^{\bar{t}} (E_D \to E_\varphi) +: \mathcal{S}$$
$$\mathcal{P}, (\text{DefL}(D(\bar{t})), E_\varphi \vdash E_D); \mathcal{S} \rightsquigarrow \mathcal{P}; \forall \bar{x}(\varphi \to D(\bar{x})) +^{\bar{t}} (E_\varphi \to E_D) +: \mathcal{S}$$

Avatar splitting:

$$\mathcal{P}, (\mathrm{AvSplit}(\pi), \mathcal{E}_1 :+ D^-); \mathcal{S} \rightsquigarrow \\ \mathcal{P}, (\pi, \mathcal{E}_1 +\!\!+ \mathcal{S}_2^-[\overline{x}\backslash\overline{y}]); (\forall \overline{x}\, \mathcal{S}_2 +^{\overline{y}} \mathcal{S}_2^+[\overline{x}\backslash\overline{y}] \to D^-) +: \mathcal{S}$$

(where $\mathcal{S}_2$ is as in the inference rule, and $\overline{y}$ are fresh variables)

$$\emptyset; E_1 \to D^-, D^+ \to E_2, \mathcal{S} \rightsquigarrow \emptyset; \mathrm{Cut}(E_1, E_2) +: \mathcal{S} \quad \text{if } D \text{ does not occur in } \mathcal{S}$$

$$\mathcal{P}, (\mathrm{AvIntro}(D), E_D +: \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}; D^+ \to (\forall \overline{x}\, \mathcal{S}_2 +^{\overline{t}} \mathcal{E}) +: \mathcal{S}$$

Logical inferences:

$$\mathcal{P}, (\mathrm{Res}(\pi_1, \pi_2), \mathcal{E}_1\ \mathcal{E}_2); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi_1, \mathcal{E}_1 :+ \varphi^-), (\pi_2, \varphi^+ +: \mathcal{E}_2); \mathcal{S}$$
$$\mathcal{P}, (\mathrm{Factor}(\pi), \mathcal{E} :+ E_1); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E} :+ E_1 :+ E_1); \mathcal{S}$$
$$\mathcal{P}, (\mathrm{Subst}(\pi), \mathcal{E}); \mathcal{S} \rightsquigarrow \mathcal{P}, (\pi, \mathcal{E}); \mathcal{S}$$
$$\mathcal{P}, (\mathrm{Taut}(\varphi), E_1 \vdash E_2); \mathcal{S} \rightsquigarrow \mathcal{P}; \mathcal{S}$$
$$\mathcal{P}, (\mathrm{Refl}, \vdash (t = t)^-) \rightsquigarrow \mathcal{P}; \mathcal{S}$$

And the following transition for Rw:

$$\mathcal{P}, (\mathrm{Rw}(\pi_1, \pi_2), \mathcal{E}_1 +\!\!+ \mathcal{E}_2 :+ E_3[s]); \rightsquigarrow \\ \mathcal{P}, (\pi_1, \mathcal{E}_1 :+ (t = s)^-), (\pi_2, \mathcal{E}_2 :+ E_3[t]); \mathcal{S}$$

Example 6.3.1. Let π_6 be the resolution proof in Example 6.1.1. Then the relation $\rightsquigarrow$ extracts the expansion proof as follows, where $\varphi_x = \forall y(P(x) \to P(y))$:

$$(\pi_6, \vdash); \vdash \ \rightsquigarrow\ (\pi_4, \underbrace{P(s(x))}_{=E_1} \vdash), (\pi_5, \vdash E_1); \vdash \ \rightsquigarrow\ (\pi_4, E_1 \vdash), (\pi_3, \vdash E_1); \vdash$$
$$\rightsquigarrow^2 (\pi_2, \underbrace{\mathrm{wk}^-(P(x)) \to E_1}_{=E_2} \vdash), (\pi_2, \underbrace{E_1 \to \mathrm{wk}^+(P(s(s(x))))}_{=E_3} \vdash); \vdash$$
$$\rightsquigarrow^2 (\pi_1, \varphi_x +^{s(x)}_{\mathrm{sk}} E_2 \vdash), (\pi_1, \varphi_{s(x)} +^{s(s(x))}_{\mathrm{sk}} E_3 \vdash); \vdash$$
$$\rightsquigarrow^2 (\pi_1, \underbrace{\forall x \varphi_x +^x \varphi_x +^x_{\mathrm{sk}} E_2}_{=E_4} \vdash), (\pi_1, \underbrace{\forall x \varphi_x +^{s(x)} \varphi_{s(x)} +^{s(x)}_{\mathrm{sk}} E_3}_{=E_5} \vdash); \vdash$$
$$\rightsquigarrow (\pi_1, E_4 \sqcup E_5 \vdash); \vdash \ \rightsquigarrow\ \emptyset; \vdash E_4 \sqcup E_5$$

Definition 6.3.2. Let π be a resolution proof. Then $|\pi|_t$ and $|\pi|_d$ denote the number of its inferences when counted as a tree or a DAG, respectively.

Lemma 6.3.1. *$\rightsquigarrow$ is terminating.*

Proof. Let $\mathcal{P}; \mathcal{S}$ be a extraction state, we define a function $|\cdot|_n$ as its termination measure by $|\mathcal{P}; \mathcal{S}|_n = |\mathcal{S}|_c + 2\sum_{(\pi,\mathcal{E})\in\mathcal{P}} |\pi|_t$ where $|\mathcal{S}|_c$ is the number of expansion trees in $\mathcal{S}$. Each case of the relation $\rightsquigarrow$ decreases this termination measure. □

Lemma 6.3.2. *Let $C_1 \rightsquigarrow C_2$, then $\mathrm{dp}(C_1) \longrightarrow \mathrm{dp}(C_2)$ is a quasi-tautology. If C_1 is acyclic, then so is C_2.*

Proof. Straightforward induction on $\rightsquigarrow$. □

Lemma 6.3.3. *Let π be a resolution proof where all Input inferences have the formula φ, and let $C_\pi \rightsquigarrow^* (\mathcal{P}, \mathcal{S}_1 \vdash \mathcal{S}_2)$. Then $\mathcal{P} = \emptyset$, and:*

1. *for every $E \in \mathcal{S}_1$, $\mathrm{sh}(E)$ is either $\top$ or $\forall \overline{x}\ (D(\overline{x}) \longrightarrow \varphi)$ where $D(\overline{t}) \overset{\mathrm{def}}{\mapsto} \varphi$, and*

2. *for every $E \in \mathcal{S}_2$, $\mathrm{sh}(E) = \varphi$.*

Proof. If $\mathcal{P} \neq \emptyset$, then we can apply one of the cases of $\rightsquigarrow$. Properties 1. and 2. are preserved in every case. □

Using the relation $\rightsquigarrow$ we obtain an expansion proof with cuts. Since there are no Skolem or definition nodes in these cuts, we can eliminate them (see Lemma 6.2.2) to obtain a cut-free expansion proof $\mathcal{E}^*$ with definitions: $C_\pi \rightsquigarrow^* (\emptyset, \mathcal{E}) \overset{\mathrm{cut}^*}{\rightsquigarrow} (\emptyset, \mathcal{E}^*)$

6.4 Definition elimination

Consider now a cut-free expansion proof $\mathcal{E}$ without eigenvariable nodes where the shallow sequent contains only definition axioms in the antecedent (this is the form of expansion proofs that are produced by the extraction in Section 6.3).

Assume that all definitions axioms are in the same polarity, that is, the introduced atom is on the left side. (The other polarity is treated analogously.)

$$\mathcal{E} = (E_1^D, \cdots, E_n^D \vdash E_\varphi), \quad \mathrm{sh}(E_i^D) = \forall \overline{x}\ (\varphi_i(\overline{x}) \to D_i(\overline{x})), \quad \mathrm{sh}(E_\varphi) = \varphi \tag{6.1}$$

The expansions of $\varphi_i(\overline{t})$ may contain definition nodes as well. Due to the acyclicity of the AbbrL-inferences, we can assume that each expansion of $\varphi_i(\overline{t})$ only contains definition nodes $+_{\mathrm{def}}\ D_j(\dots)$ where $j < i$. We will now successively eliminate each of the definition axioms, starting with the one for n. The expansion tree for $\forall \overline{x}\ (\varphi_n(\overline{x}) \to D_n(\overline{x}))$ has the following form:

$$\forall \overline{x}\ (\varphi_n(\overline{x}) \to D_n(\overline{x})) +^{\overline{t_1}} \left(E_1 \to D_n(\overline{t_1})^-\right) \cdots +^{\overline{t_k}} \left(E_k \to D_n(\overline{t_k})^-\right)$$

For performance reasons, we consider two cases here: in the first case the deep sequent is tautological, in the second case the deep sequent is only quasi-tautological. In both cases, we replace definition nodes $+_{\mathrm{def}}\ D_n(\dots)$ in the expansion proof by subtrees of E_n^D. However in the second case we perform many duplications, and we want to avoid this if possible.

Tautological deep sequent. If $\mathrm{dp}(\mathcal{E})$ is not just quasi-tautological but tautological (this is the case if the resolution proof did not use the built-in equational inference Rw and Refl), then we merely need to replace each occurrence of a definition node $+_{\mathrm{def}}\ D_n(\overline{t_i})^+$ with the corresponding expansion tree E_i. We define a function $R[\cdot]$ that performs this replacement on definition nodes for D_n, and recursively maps over the other possible nodes:

$$R[\varphi_n(\overline{t}) +_{\mathrm{def}} D_n(\overline{t})^+] = \begin{cases} E_i & \text{if there exists } i \text{ such that } \overline{t} = \overline{t_i} \\ \mathrm{wk}^+(\varphi_n(\overline{t_i})) & \text{otherwise} \end{cases}$$

Lemma 6.4.1. *Let $\mathcal{E}$ be an expansion sequent as in Equation* (6.1)*. If* $\mathrm{dp}(\mathcal{E})$ *is tautological, then* $\mathrm{dp}\left(E_1^D, \cdots, E_{n-1}^D \vdash R[\varphi]\right)$ *is tautological as well.*

Proof. Let I be a counter-model for $\mathrm{dp}\left(E_1^D, \cdots, E_{n-1}^D \vdash R[\varphi]\right)$. Assume without loss of generality that I is not defined for any of the atoms $D_n(\overline{t})$; we extend I by setting $I(D_n(\overline{t_i})) = I(\mathrm{dp}(E_i))$ for all i, and $I(D_n(\overline{t})) = 0$ otherwise. We now have $I(\mathrm{dp}(E_n^D)) = 1$ as well as $I(\mathrm{dp}(R[\varphi_n(\overline{t}) +_{\mathrm{def}} D_n(\overline{t})^+])) = I(D_n(\overline{t}))$ and thus $I(\mathrm{dp}(R[E_\varphi])) = I(\mathrm{dp}(E_\varphi))$ since D_n only occurs in E_φ in definition nodes. Hence I is a counter-model for $\mathrm{dp}(\mathcal{E})$ as well. □

Quasi-tautological deep sequent. In general, the sequent $\mathrm{dp}(\mathcal{E})$ may be just quasi-tautological. In particular the following congruence scheme is quasi-tautological and not tautological:

$$t_1 = s_1 \wedge \cdots \wedge t_m = s_m \to (D(t_1, \cdots, t_m) \leftrightarrow D(s_1, \cdots, s_m))$$

Hence it is no longer sufficient to replace a definition node $+_{\mathrm{def}}\ D_n(\overline{t_i})^+$ by just the expansion tree where the arguments are syntactically equal. We replace it by *all* the expansion trees $E_1, \cdots, E_n$, suitably replacing the term vectors $\overline{t_i}$ so that the shallow formulas match. To perform this replacement, we define a generalization operation G. We first define the generalized tree E, and then define the replacement operation R^{eq} uniformly for all definition atoms:

$$E = G_{\varphi(\overline{x})}(E_1) \sqcup \cdots \sqcup G_{\varphi(\overline{x})}(E_n)$$

$$R^{\mathrm{eq}}[\varphi_n(\overline{t}) +_{\mathrm{def}} D_n(\overline{t})^+] = E[\overline{x} \backslash \overline{t}]$$

Definition 6.4.1. Let φ, ψ be first-order formulas such that $\psi\sigma = \varphi$ for some substitution σ, and $E \in \mathrm{ET}^p(\varphi)$ an expansion tree without definition nodes. Then we recursively define its generalization $G_\psi(E) \in \mathrm{ET}^p(\psi)$:

$$G_\psi(A^p) = \psi^p \qquad G_{\neg\psi}(\neg E) = \neg G_\psi(E) \qquad G_\psi(\mathrm{wk}^p(\varphi)) = \mathrm{wk}^p(\psi)$$
$$G_\psi(E_1 \sqcup E_2) = G_\psi(E_1) \sqcup G_\psi(E_2)$$
$$G_{\psi_1 \wedge \psi_2}(E_1 \wedge E_2) = G_{\psi_1}(E_1) \wedge G_{\psi_2}(E_2)$$
$$G_{\forall x \psi}(\forall x \varphi +^y_{\mathrm{ev}} E) = \forall x \varphi +^y_{\mathrm{ev}} G_{\psi[x\backslash y]}(E)$$
$$G_{\forall x \psi}(\forall x \varphi +^t_{\mathrm{sk}} E) = \forall x \varphi +^t_{\mathrm{sk}} G_{\psi[x\backslash t]}(E)$$
$$G_{\forall x \psi}(\forall x \varphi +^{t_1} E_1 \cdots +^{t_n} E_n) = \forall x \varphi +^{t_1} G_{\psi[x\backslash t_1]}(E_1) \cdots +^{t_n} G_{\psi[x\backslash t_n]}(E_n)$$

Lemma 6.4.2. *Let $\mathcal{E}$ be an expansion sequent as in Equation* (6.1)*. If* $\mathrm{dp}(\mathcal{E})$ *is quasi-tautological, then* $\mathrm{dp}\left(E_1^D, \cdots, E_{n-1}^D \vdash R^{\mathrm{eq}}[\varphi]\right)$ *is quasi-tautological as well.*

Proof. Similar to Lemma 6.4.1, except we now set $I(D_n(\overline{a})) = I(\mathrm{dp}(E)[\overline{x}\backslash\overline{a}])$ for all $\overline{a}$ in the domain of the counter-model. □

6.5 Complexity

We will now give a double-exponential upper bound on the complexity of the whole algorithm as summarized in Algorithm 6. Without Avatar inferences and definition inferences, this bound would be just single-exponential. We do not know whether this double-exponential bound is actually attained in the worst case, the best known lower bound is single-exponential (as is necessary in any conversion from resolution proofs to expansion proofs).

Lemma 6.5.1. *Whenever* $C_\pi \rightsquigarrow^* (\emptyset; \mathcal{E})$*, then* $|\mathcal{E}| \leq 4^{|\pi|_d+1}$.

Proof. Let $\mathcal{P}; \mathcal{S}$ be a extraction state, we define its size as $|\mathcal{P}; \mathcal{S}| = |\mathcal{S}| + \sum_{(\pi,\mathcal{E})\in\mathcal{P}} 2|\pi|_t|\pi|_m + |\mathcal{E}|$, where $|\pi|_m$ is the maximum length of a sequent occurring in π. No case of the relation $\rightsquigarrow$ increases this size. Observe that $|\pi|_m \leq 2|\pi|_d$, and hence $|\mathcal{E}| \leq |C_\pi| \leq 2|\pi|_t|\pi|_m \leq 2^{|\pi|_d+2}|\pi|_d \leq 4^{|\pi|_d+1}$. □

Lemma 6.5.2. *Let* π *be a resolution proof, then* $|\textsc{Transform}(\pi)| \leq 2^{4^{|\pi|_d}}$.

Proof. Let E_1, E_2, E_3 be as in Algorithm 6. From Lemma 6.5.1 we know that $|E_1| \leq 4^{|\pi|_d+1}$. Note that there are fewer than $|\pi|_d$ splitting and definition inferences in π, and hence fewer than $|\pi|_d$ cuts and definitions in E_1 and E_2. The cost of cut-elimination is shown in Lemma 6.2.3, we have $|E_2| \leq |E_1|^{2^c}$ where $c < |\pi|_d$ is the number of cuts. We have a similar bound for definition elimination: each step increases the size at most quadratically and we have $|E_3| \leq |E_2|^{2^d}$ where $d < |\pi|_d$ is the number of definitions. Hence $|\textsc{Transform}(\pi)| \leq 4^{(|\pi|_d+1)2^{c+d}} \leq 2^{4^{|\pi|_d}}$. □

6.6 Empirical evaluation

We evaluated the implementation in three experiments using GAPT 2.9, which contains Algorithm 6 as the `ResolutionToExpansionProof` function.

First, we compared it against the LK-based method for expansion proof extraction from resolution proofs described in [52], which is also implemented in GAPT. The TSTP library (Thousands of Solutions from Theorem Provers, see [94]) contains proofs from a variety of automated theorem provers. We loaded all 6341 Prover9 proofs into GAPT, and measured the runtime of each

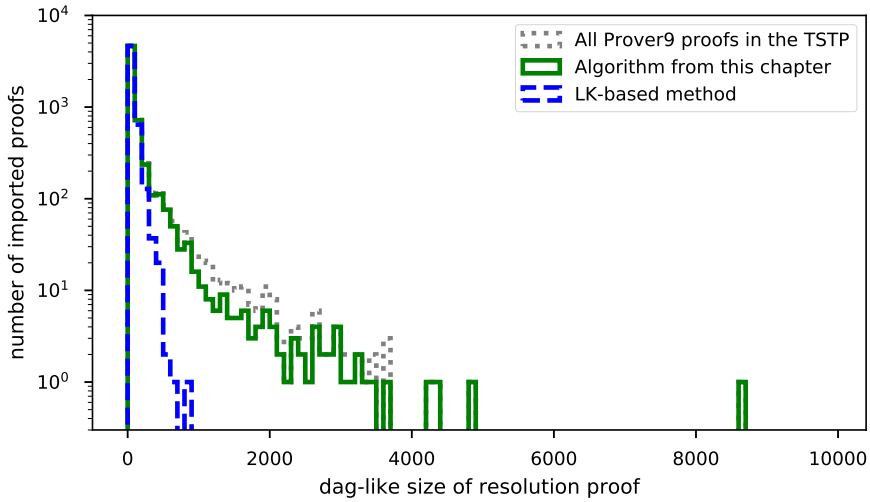


Figure 6.2: Comparison of the number of successfully imported Prover9 proofs in the TSTP using Algorithm 6 and the LK-based algorithm in [52].

of the algorithms with a timeout of 120 seconds. GAPT is unable to read 40 of the proofs at all since Prover9's `prooftrans` executable fails. Of the remaining proofs, the LK-based method in [52] imports 5479 proofs (87.0%). Algorithm 6 imports 6125 proofs (97.2%), it only fails on 176 proofs due to the timeout. There is no proof where the LK-based method is successful, but Algorithm 6 fails.

Figure 6.2 plots the number of successfully imported proofs using each of the algorithms against the DAG-like size of the resolution proof. The algorithm in this chapter manages to import almost all Prover9 proofs in the TSTP including very large proofs; proofs between 1000 and 10000 inferences can not be imported by the LK-based method at all.

As the second experiment, we evaluated the quality of the clausification allowed by the resolution calculus used in this chapter. We took the 662 problems in the first-order FEQ, FNE, FNN, and FNQ divisions of the CASC-26 competition whose size was less than one megabyte after including the separate axiom files. On these 662 problems, we compared the performance of the E theorem prover [87] version 2.1 (as submitted to CASC) when directly

running on the problems, and when running on the clausification produced by GAPT.

GAPT fails to clausify 19 of the problems due to excessive runtime. These problems (e.g. `HWV053+1`) have blocks of more than a thousand quantifiers. On the remaining ones we ran E in both the default configuration, and the auto-scheduling mode (as used in the CASC), both with a timeout of 60 seconds. In the default mode the E clausification results in 81 found proofs, the GAPT clausification in 89. With auto-scheduling enabled the result is reversed but still close, and E's own clausification produces more proofs (308) than GAPT's clausification (285). These results show that the clausification allowed by the calculus presented in this chapter is competitive with the ones implemented in state-of-the-art automated theorem provers. We believe that the reversed results for the auto-scheduling mode are indicative of a larger trend in first-order theorem provers: many provers train their strategy selection algorithms on the exact problems from the TPTP and thus rely very closely on the syntactic features of these problems.

Finally, we wanted to compare the runtime of Algorithm 6 with the runtime of the first-order prover. To this end, we again used the same 662 first-order problems from CASC-26 and used GAPT's external prover interface to obtain expansion proofs from E. This prover interface uses the clausification of the resolution calculus described in this chapter to create a CNF, sends this CNF to E, then parses and reconstructs a resolution proof using proof replay, and finally constructs an expansion proof using Algorithm 6. We measured the mean runtime of each phase on the successfully imported proofs (only 7 proofs could not be imported in the time limit of 2 minutes). The E prover itself takes up 62%, the largest part of the runtime. Algorithm 6 only takes 7.2% of the total runtime. The rest of the runtime is spent mainly in proof replay (15.6%) and clausification (4.6%).

Since the expansion proof extraction is only a fraction of the prover runtime, we believe that it is practically feasible to generate expansion proofs instead of resolution proofs. Even though expansion proofs can be exponentially larger in the worst case, this situation seems to occur only rarely in practice.

6.7 Direct elimination of Avatar-inferences

When importing resolution proofs with Avatar splitting using the procedure described in this chapter, each cut in the expansion proof corresponds to an atom introduced using Avatar splitting. The cut-elimination of the expansion proof then corresponds to the elimination of the splitting inferences.

This naturally leads to the question whether and how we can eliminate Avatar inferences directly on the resolution proof itself. There are application for resolution proofs which require that the proofs do not contain splitting inferences: for example we can extract interpolants from resolution proofs [58], or use them in the cut-elimination method CERES [8]. In this section we will present a method to eliminate splitting inferences from resolution proofs at exponential cost.

The basic idea is that we eliminate all resolution inferences on splitting atoms. That is, resolution inferences where the resolved atom $d \overset{\text{def}}{\mapsto} \forall \overline{x}\, \mathcal{S}_d$ is a defined atom as used in the inferences AvSplit and AvIntro:

$$\dfrac{\begin{array}{cc}(\pi_1) & (\pi_2) \\ \mathcal{S}_1 :+ D & D +: \mathcal{S}_2\end{array}}{\mathcal{S}_1 +\!\!+ \mathcal{S}_2}\ \text{Res}$$

Once we have eliminated all such resolution steps, the resulting proof will no longer contain splitting inferences as the introduced splitting atoms can only be removed using the resolution inference. We eliminate each resolution step separately. As a first step, we show how to transform the left sub-proof (π_1) into a proof of $\mathcal{S}_1 +\!\!+ \mathcal{S}_D$:

Lemma 6.7.1. *Let $d \overset{\text{def}}{\mapsto} \forall \overline{x}\, \mathcal{S}_d$ be a splitting atom, and π a resolution proof ending in $\mathcal{S} :+ D \cdots :+ D$. We implicitly assume that $\mathcal{S}$ and $\mathcal{S}_D$ have disjoint sets of free variables. Then there exists a resolution proof $P(\pi, D)$ of $\mathcal{S} +\!\!+ \mathcal{S}_D$ such that $|P(\pi, D)|_d \leq |\pi|_d |\mathcal{S}_D|$.*

Proof. The interesting case is when π is an AvSplit-inference, so let π end in an AvSplit inference:

$$\dfrac{\begin{array}{c}(\pi') \\ \mathcal{S} +\!\!+ \mathcal{S}_D\end{array}}{\mathcal{S} :+ D}\ \text{AvSplit}$$

In this case we can just leave out the AvSplit-inference and set $P(\pi, D) = \pi'$. If D is not the main formula of the inference, then we can proceed recursively. Consider representatively the case where π is a Res-inference:

$$\dfrac{\begin{array}{cc}(\pi_1) & (\pi_2)\\ \mathcal{S}_1 :+ D \cdots :+ D :+ \varphi & \varphi +: \mathcal{S}_2 :+ D \cdots :+ D\end{array}}{\mathcal{S}_1 +\!+ \mathcal{S}_2 :+ D \cdots :+ D}\ \text{Res}$$

Then we construct $P(\pi, D)$ as follows:

$$\dfrac{\dfrac{\begin{array}{cc}(P(\pi_1, D)) & (P(\pi_2, D))\\ \mathcal{S}_1 +\!+ \mathcal{S}_D :+ \varphi & \varphi +: \mathcal{S}_2 +\!+ \mathcal{S}_D\end{array}}{\mathcal{S}_1 +\!+ \mathcal{S}_2 +\!+ \mathcal{S}_D +\!+ \mathcal{S}_D}\ \text{Res}}{\mathcal{S}_1 +\!+ \mathcal{S}_2 +\!+ \mathcal{S}_D}\ \text{Factor}^+$$

We can permute $P(\cdot, D)$ and Subst inferences. We can also permute Factor/Rw inferences where D is the main formula, by repeating the Factor/Rw inference for every atom in $\mathcal{S}_D$. If π is an AbbrR-inference, we replace it by AllR and OrR inferences. Note that D is not the main formula of a clausification inference, since splitting atoms do not occur in the input formula or in defined subformulas. Furthermore D cannot be the main formula of a Refl-inference since it is not an equation. □

We can now plug the left subproof into all the AvIntro inferences in the right subproof:

Lemma 6.7.2. *Let $D \overset{\text{def}}{\mapsto} \forall \overline{x}\, \mathcal{S}_D$ be a splitting atom, π_D a resolution proof ending in $\mathcal{S}' +\!+ \mathcal{S}_D$, and π a resolution proof ending in $D +: \ldots D +: \mathcal{S}$. We implicitly assume that $\mathcal{S}$, $\mathcal{S}_D$, and $\mathcal{S}'$ have pairwise disjoint sets of free variables. Then there exists a resolution proof $R(\pi, \pi_D, D)$ of $\mathcal{S} +\!+ \mathcal{S}'$ such that $|R(\pi, \pi_D, D)|_d \leq |\pi|_d |\mathcal{S}_D| + |\pi_D|_d$.*

Proof. The interesting case is when π is a AvIntro-inference with D as a main formula:

$$\dfrac{}{D +: S_D}\ \text{AvIntro}$$

In this case we can set $R(\pi, \pi_D, D)$ to π_D. The other cases are as in Lemma 6.7.1. □

Combining the two previous Lemmas 6.7.1 and 6.7.2, we can now eliminate a single resolution inference on a splitting atom:

Lemma 6.7.3. *Let π be a top-most resolution inference on a splitting atom $D \overset{\text{def}}{\mapsto} \forall \overline{x}\, \mathcal{S}_D$ (that is, π_1 and π_2 do not contain any resolution inferences on splitting atoms):*

$$\frac{\begin{array}{cc}(\pi_1) & (\pi_2)\\ \mathcal{S}_1 :\!+ D & D :\!+ \mathcal{S}_2\end{array}}{\mathcal{S}_1 +\!+ \mathcal{S}_2}\, Res$$

Then $\pi' = R(\pi_2, P(\pi_1, D), D)$ is a resolution proof of $\mathcal{S}_1 +\!+ \mathcal{S}_2$ without resolution inferences on splitting atoms such that $|\pi'|_d \leq 2|\pi|_d|\mathcal{S}_D|$.

Proof. Neither $P(\cdot,\cdot)$ nor $R(\cdot,\cdot,\cdot)$ introduce resolution inferences on new atoms, hence π' does not contain resolution inferences on splitting atoms. The size bound is the combination of the results from Lemmas 6.7.1 and 6.7.2. □

Finally, we eliminate all resolution inferences one after another:

Theorem 6.7.1. *Let π be a resolution proof ending in the empty sequent, then there is a resolution proof without splitting π' also ending in the empty sequent such that $|\pi'|_d \leq c^{|\pi|_d}$ for some constant c that only depends on the maximum size of a splitting component.*

Proof. If there is an AvSplit or AvIntro inference in π then then there is a resolution inference on a splitting atom below it. This is because the proof ends in the empty sequent and there is no other way for the splitting atom to go away.

Hence it suffices to eliminate all resolution inferences on splitting atoms. By Lemma 6.7.3 we know how to eliminate a single such resolution inference. We now iteratively eliminate these inferences going from top to bottom, in each iteration the size of the proof is at most increased by a factor of $c = 2 \max_{D \overset{\text{def}}{\mapsto} \forall \overline{x}\, \mathcal{S}_D} |\mathcal{S}_D|$. □

Theorem 6.7.1 gives a exponential bound on the size of a resolution proof without splitting in terms of a resolution proof with splitting.

Open Problem 6.7.1. Is the bound in Theorem 6.7.1 optimal? Can Avatar splitting inferences be removed at subexponential cost?

7 Implementation and evaluation

We set out in Section 2.9 to reverse induction-elimination. In the previous chapters we have evaluated individual algorithms that are parts of this reversal: Chapter 6 shows how to obtain instance proofs, Chapter 4 and Section 4.4 describe an algorithm to find an induction grammar covering the proofs, and Chapter 5 explains how to solve the induced formula equation. In this chapter we will evaluate the main algorithm, and its implementation in the GAPT system.

The implementation and empirical evaluation of proof-theoretic algorithms such as this one provides valuable practical insight. Proof-theoretic algorithms often have large worst-case runtime. For example, cut-elimination in first-order logic necessarily has non-elementary runtime complexity. This result would lead one to believe at first that cut-elimination is utterly infeasible. Yet cut-elimination is perfectly practical for many proofs that occur in practice, at least if implemented efficiently [34].

A similar performance question also arises for the algorithms that we have seen so far. All the major individual algorithms we have seen so far have at least exponential worst-case runtime. From a purely theoretical point of view, we can only say that they work. But it is possible that the worst-case is actually common in practice. Thus we need to conduct experiments to evaluate the performance on actual examples.

There is an even more fundamental model assumption that we have briefly touched upon in Section 2.10: that the instance proofs have a certain form of regularity. The whole approach depends crucially on this feature: that instance proofs are similar to proofs obtained by induction-elimination. If the instance proofs were completely different, then there would be little hope of

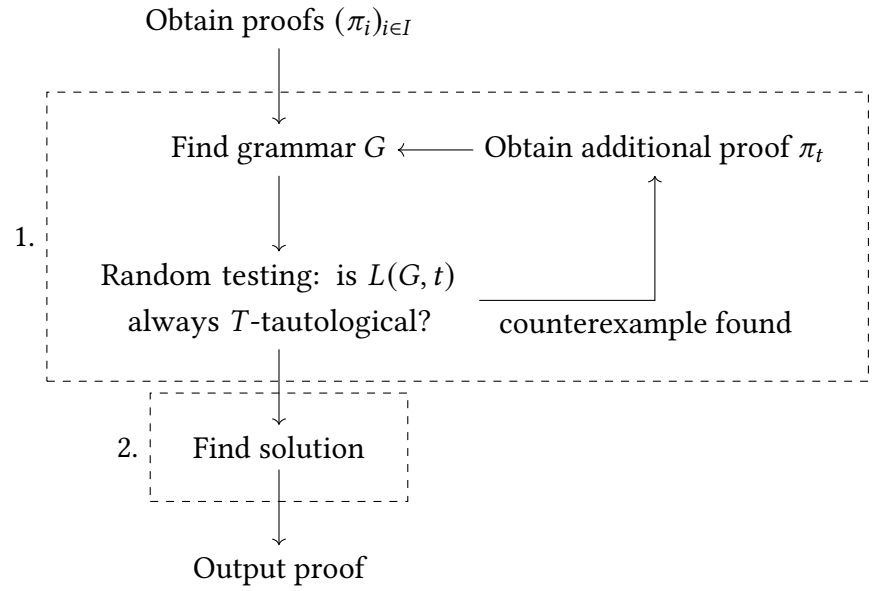


Figure 7.1: Refinement loop to find induction grammars

generalizing them to a proof with induction. A priori, we might even expect instance proofs to have this regularity. Whether this is actually the case for automatically generated proofs is a question that can only be answered by practical experiments. We will answer this question in Sections 7.4 and 7.5.

7.1 Refinement loop to find induction grammars

The algorithm we saw in Section 4.4 produces an induction grammar of minimal size that covers a given (finite) family of languages $(L_t)_{t\in I}$. However there is no obvious choice for this set I. We could e.g. take I to be the set of all free constructor terms of size less than 2. However in practice, this choice often results in induction grammars G such that $L(G,t)$ is not T-tautological for some t of size larger than 2. In this case, Φ_G cannot be T-solvable (remember Lemma 5.3.1 and Theorem 5.3.2).

Therefore, we use a refinement loop to find induction grammars. In each iteration we check the generated induction grammar G and verify that $L(G,t)$

is T-tautological for a large number of free constructor terms t. If the check succeeds, then we work under the assumption that G is T-tautological. This check is only a necessary requirement. Unfortunately it could still be possible that G might not be solvable, as there are some tautological induction grammars whose formula equation is not solvable (Theorems 5.4.1 and 5.4.3). We also cannot check for solvability of formula equations (even the ones induced by induction grammars) in general, as this is undecidable by Theorem 5.4.2.

On the other hand, if we find a t such that $L(G, t)$ is not T-tautological, then we obtain a new proof π of the instance problem for the parameter t, extend the family $(L_t)_{t\in I}$ to $(L_t)_{t\in I\cup\{t\}}$ by setting $L_t = L(\pi_t)$. Note that since L_t is T-tautological, any induction grammar G' covering the extended family will satisfy that $L(G', t)$ is T-tautological. Hence we will not encounter the same counterexample t in the following iterations of the loop. The overall structure of the method, combining the refinement loop as well as the algorithms to solve formula equations from Sections 5.6 and 5.7, is shown schematically in Figure 7.1.

7.2 Implementation

The open source GAPT framework [37] (short for General Architecture for Proof Theory) contains implementations of a large variety of algorithms related to expansion proofs (as a generalization of Herbrand sequents) and proof theory in general. Since version 2.9 (released in August 2018), it also contains algorithms for the computation of covering induction grammars as in Section 4.4 and solving formula equations using forgetful inference as in Section 5.6 (a version restricted to natural numbers following the formalism of [31] was already released in version 2.0 in January 2016).

To give a short overview of the implementation, let us look at how the concepts in each chapter are implemented in GAPT. The grammars introduced in Chapter 2 are in the `gapt.grammars` package; the classes `VTRATG` and `InductionGrammar` model the respective concepts. The term encoding of formulas of Section 2.4 is implemented as `InstanceTermEncoding(sequent)`, i.e., we have a different term encoding for each sequent. Interestingly enough,

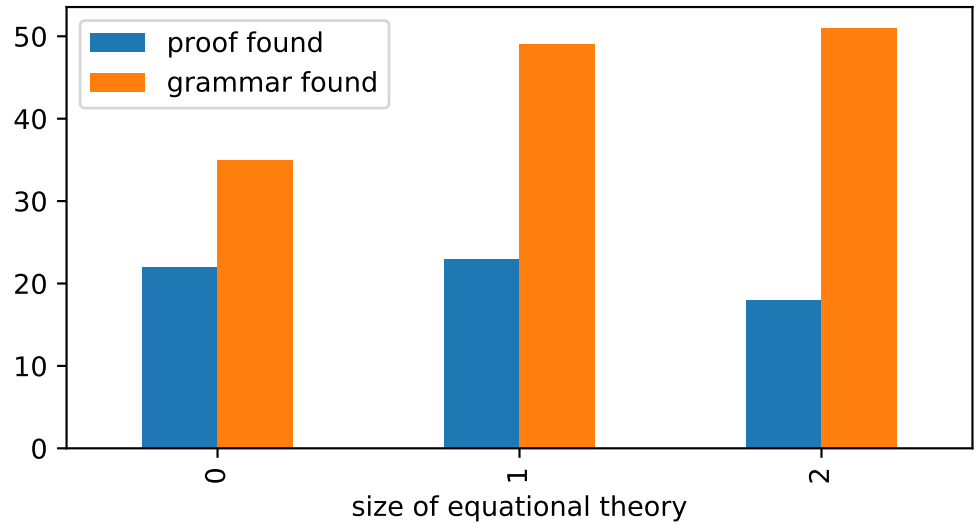


Figure 7.2: Results for the experiment where we pick an equational theory from equations in the antecedent. We tried all combinations of up to two equations. Only the solution algorithm using forgetful inference is used. The benchmark set is an earlier version of the TIP, containing only 319 problems.

there has been little practical need to implement the map that assigns to each proof a corresponding grammar; this map is only implemented for simple induction proofs. The function `extractInductionGrammar` actually operates on a slightly more general level, namely on expansion proofs (recall Section 6.2; technically, the induction inference is encoded as a higher-order assumption).

The algorithms to find covering grammars from Chapter 4 are contained in the package `gapt.grammars`, in `deltaTableAlgorithm`, `findMinimalVTRATG`, `Reforest`, and `findMinimalInductionGrammar`. For the algorithms based on MaxSAT, GAPT contains interfaces to several MaxSAT solvers. The solver we used for the evaluations is the external program `OpenWBO` [67]. Often it is useful to have a solver that does not require any extra dependencies, for example when running tests. For this reason, GAPT also bundles the Sat4j library, which includes a MaxSAT solver that is available via the `MaxSat4j` class.

Given an induction grammar, we can compute the induced formula equation using `InductionBUP.formula`[1]. The solution algorithm using forgetful inference (Section 5.6) is implemented as `hSolveQBUP`, and the the function `solveBupViaInterpolationConcreteTerms` implements the algorithm using interpolation (Section 5.7).

The refinement loop is implemented in the class `TreeGrammarProver`. There is a parser for problems in the TIP format [21] in `TipSmtParser`.

Instance proofs are obtained using standard first-order theorem provers based on superposition. GAPT contains interfaces to several external first-order provers such as Vampire, E, etc. However for the evaluation in this chapter we use the built-in Escargot prover of GAPT. Instance problems are typically easy to prove, therefore there is little need to use stronger provers like Vampire.

There is a second point in the algorithm where we make use of automated theorem provers: when we check whether $L(G, t)$ is T-tautological. The formula corresponding to $L(G, t)$ is quantifier-free. If T is empty, this formula is hence in the `QF_UF` fragment efficiently decidable by SMT solvers. If $T \neq \emptyset$, then we use the SMT solver as a semi-decision procedure: we also pass it the quantified formulas in T. The SMT solver will then apply heuristic instantiation to the formulas in T. We also use a time limit to prevent non-termination. For the evaluation, we use CVC4 as the SMT solver.

For problems imported from external benchmark sets, there is typically no clear choice of equational theory. To evaluate how much of an effect an equational theory could have, we performed an initial experiment on an earlier version of the TIP benchmark suite. For every problem, we picked an equational theory consisting of equations in the antecedent of the sequent. We tried all combinations of up to two equations. Only the solution algorithm based on forgetful inference was used. The results are shown in Figure 7.2. Using an equational theory has a small negative effect on the number of proofs found. With one equation taken from the antecedent one extra proof is found, but with two equations the number of found proofs is significantly lower. One

[1] In GAPT, formula equations are often referred to as Boolean unification problems, since they can be seen as instances of Boolean unification with predicates [33].

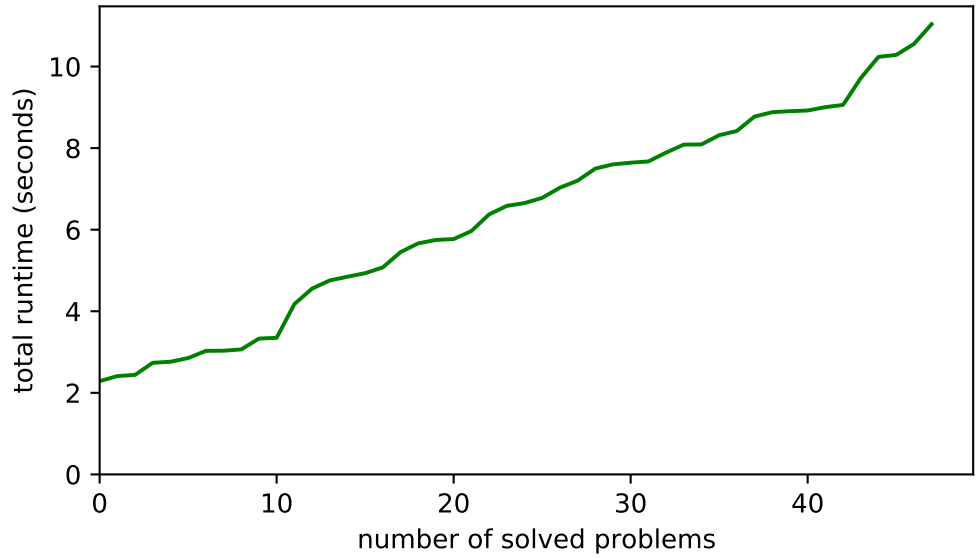


Figure 7.3: Cactus plot of the runtime on TIP benchmarks. For each successfully solved problem we plot the shortest runtime on the y-axis.

explanation is that the solution algorithm constructs the solution formula out of the instances; using an equational theory reduces the number of these instances and hence reduces the amount of "raw material" that can be used. The number of candidate grammars found increases significantly with a single equation. This is not surprising, because it is easier to cover smaller term sets. Eliminating the instances of one axiom (the equation) also reduces the amount of irregularity that could prevent us from finding a grammar. For the evaluation we hence use the empty theory $T = \emptyset$.

7.3 Evaluation as automated inductive theorem prover

First we evaluate the algorithm as an automated inductive theorem prover. Given just a sequent, we want to find a simple induction proof. As a benchmark set we use problems from the TIP benchmark suite (tons of inductive

problems [21]). This TIP benchmark suite aggregates benchmarks used to evaluate the IsaPlanner [28], Zeno [89], and HipSpec [22] provers. The benchmarks mostly concern the verification of functional programs; so far no public competitions have been held.

Some of the problems are not syntactically of the form required by simple induction proofs: their conclusion contains more than one universal quantifier, the formulas are not prenex or contain strong quantifiers. There is a small preprocessing step that Skolemizes and prenexifies the sequents. If there are multiple universal quantifiers in the conclusion, we Skolemize all but one of them (and try all choices). We use a time limit of 10 minutes.

There are 543 problems in the TIP benchmark suite, 529 of which are quantified. For 62 of these problems, the algorithm finds a candidate grammar that is tautological for the random instances that were checked. There are 38 problems where no grammar can cover the term sets produced by the first-order prover. (This can be detected by the algorithm from Section 4.4 because it is guaranteed to find such a grammar, if it exists.) This indicates very irregular instance proofs that cannot be generalized to an inductive proof. In the remaining 429 problems, the refinement loop times out.

Figure 7.3 shows a cactus plot of the runtime of the prover on the successfully solved TIP problems. We picked the minimum runtime for each problem (as we tried multiple configurations, such as for example multiple algorithms to solve the induced formula equation).

The mean number of iterations of the refinement loop is 2.4. We observe a relatively small difference in the mean number of iterations depending on whether we can find a grammar and/or a proof. In the cases where the algorithm finds a proof, the mean number of iterations is 2.15. In the cases where it finds a grammar but no proof, the mean is 2.83. We can also visually observe this difference in Figure 7.4: this figure reinforces the picture that there is exists a dichotomy between regular instance proofs, where the refinement loop converges quickly and the induced formula equation is solvable, versus irregular instance proofs, where the refinement loop converges slowly and the formula equation is not solvable.

In the cases where it cannot even find a candidate grammar, the mean number of iterations in the refinement loop is 2.39. However in this case there

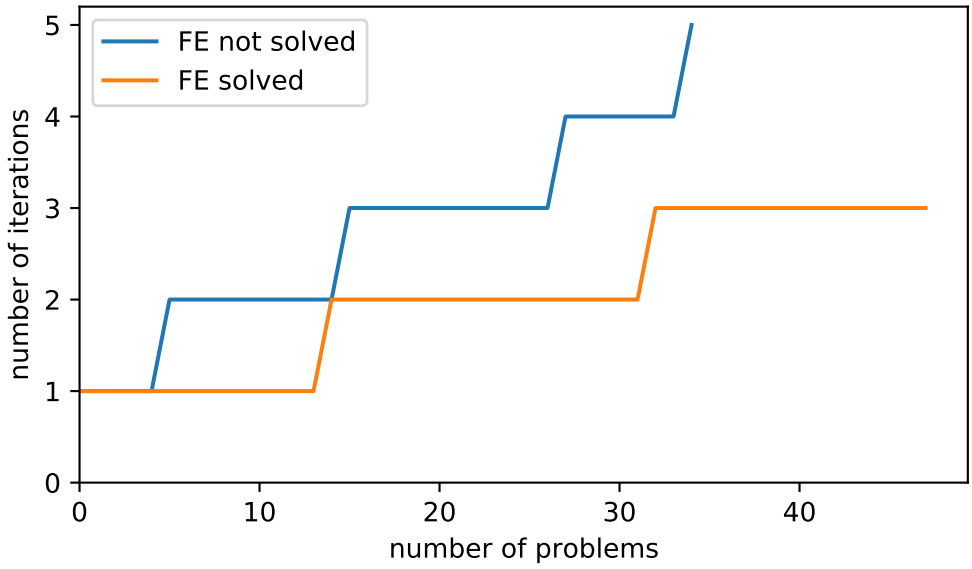


Figure 7.4: Cactus plot of the number of iterations of the refinement loop on TIP benchmarks where a candidate grammar was found.

is a larger variance. The maximal number of iterations is 17 when no grammar could be found. All grammars that have been found were found after at most 5 iterations.

We also tried different weighing schemes for the grammars, preferring grammars with 1) fewer productions or 2) smaller number of symbols. The choice of weighing scheme has almost no effect on the number of iterations or the success of the algorithm (on the TIP benchmark set).

Of the 62 problems where a grammar has been successfully found, 48 can be completed to a simple induction proof by solving the induced formula equation. We used both the method based on forgetful inference as well as the one based on interpolation described in Sections 5.6 and 5.7. The forgetful inference method solves 30 formula equations, the interpolation method solves 36 formula equations. There are hence 18 problems that both methods could solve. Figure 7.5 plots the runtime of the two algorithms. We see a very clear performance difference: the interpolation-based algorithm never takes more than a second, while the runtime of the algorithm based on forgetful inference

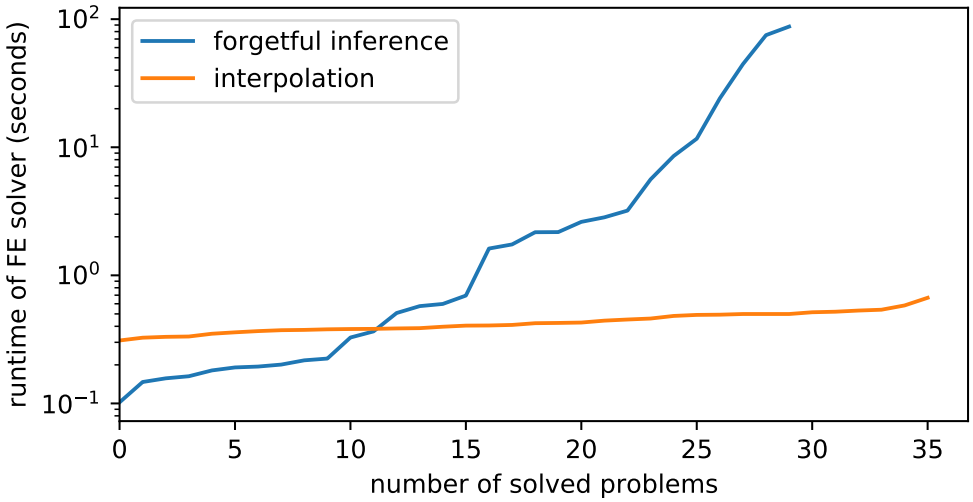


Figure 7.5: Cactus plot of the runtime of the algorithms solving the induced formula equation on TIP benchmarks. The y-axis is scaled logarithmically to make the results easier to compare given the significant difference in magnitude.

quickly grows exponentially. This comes to no surprise: it needs to compute the canonical solution, whose size is exponential in the instance term. The following step of applying forgetful inferences also has exponential runtime in the worst case.

GAPT also contains two other automated inductive theorem provers: one is an analytic induction prover which heuristically instantiates the induction scheme with variants of the goal and then runs a first-order prover. Spin integrates an induction inference in the superposition loop of the Escargot prover in GAPT [39] similar to Zipperposition [25]. On the TIP, the analytic induction prover solves 78 problems with the independent induction scheme, and 115 with the sequential scheme; Spin solves 142 problems in the default mode, 133 if generalization is disabled, and 94 if counterexample testing is disabled [39].

There is one TIP problem that the algorithm presented here can solve, but none of the other methods in GAPT: `isaplanner/prop_59`. The problem is

$\forall x \forall y\ (y = \texttt{nil} \rightarrow \texttt{last}(x +\!\!+ y) = \texttt{last}(x))$. The natural induction formula, $\forall x\ (x +\!\!+ \texttt{nil} = x)$, is not analytic and is not found by Spin either.

In this evaluation on the TIP benchmark set, the algorithm often fails to find a candidate grammar that covers the automatically generated instance proofs. This situation could arise from an algorithmic issue, where the algorithm that we use to compute covering grammars is too slow or finds "inappropriate" grammars that do not describe a proof with induction. It could also arise from an intrinsic quality of the generated instance proof: that the automatically found proofs do not possess the regularity necessary to generalize them to a proof with induction. To determine the cause of this issue, we will conduct an additional experiment in Section 7.4, where we start with induction proofs and run the algorithm on families of induction-eliminated instance proofs (which are hence regular and should be generalizable into a proof with induction). This experiment will allow us to tell whether the failure is due to the irregularity of the automatically generated instance proofs or not.

In a smaller number of cases, the algorithm also fails to find a solution to the formula equation induced by the grammar. We will provide an explanation in Section 7.5 by closely examining one of the problems as a detailed case study.

7.4 Evaluation of reversal of induction-elimination

For the second evaluation, we start out with 73 simple induction proofs manually formalized in GAPT concerning basic properties of operations on natural numbers and lists. Of these, 25 are proofs of problems in the TIP benchmark suite. GAPT contains 92 formal proofs of TIP problems as example data to test induction-elimination, and 25 of these happen to be simple induction proofs. The other 48 proofs in this evaluation are lemmas from a formalization of the fundamental theorem of arithmetic in GAPT that happen to be simple induction proofs.

For each proof, we performed essentially two experiments. First, we used induction-elimination to compute the instance proofs and perform the reconstruction. As the second experiment, we generated the instance proofs using

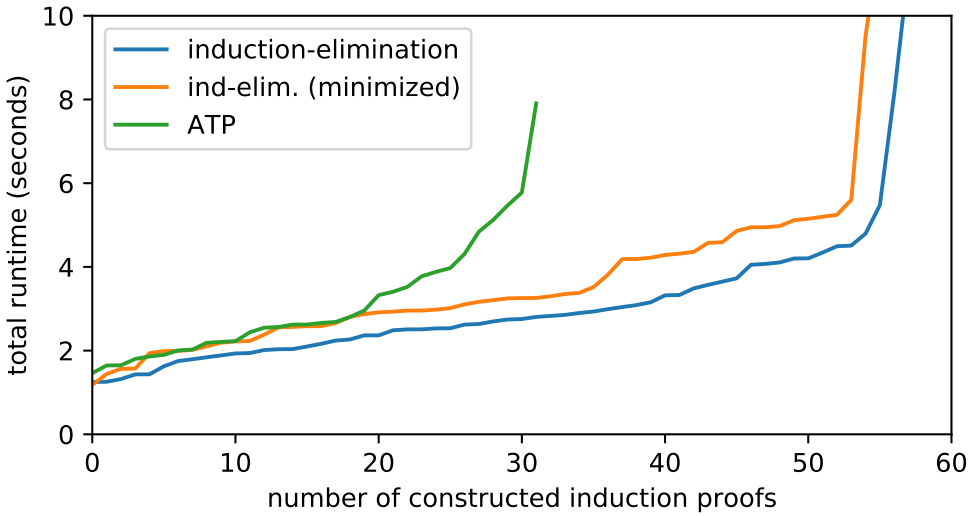


Figure 7.6: Cactus plot of the runtime on induction proofs in GAPT, depending on whether the instance proofs are generated via induction elimination or by an automatic theorem prover.

an automated theorem prover (Escargot, just as in Section 7.3). As we see in the following summary, the proofs obtained by induction-elimination are clearly different from the proofs produced by the automated theorem prover: the algorithm always finds a grammar for induction-elimination, but only for about two thirds of the ATP proofs. There is a smaller difference in the second solution-finding phase. Interestingly for every problem where the algorithm failed to solve the induced formula equation from the ATP proofs, the algorithm also failed to solve the FE for the proofs from induction-elimination.

	#problems	#grammars	#solutions
ind.-elim.	73	73	63
ATP	73	39	32

The difference is also visible from the cactus plot in Figure 7.6: the runtime for the instance proofs generated with the automated theorem prover is always larger (to little surprise), but it diverges quickly, indicating that the proofs become more irregular. Figure 7.6 also shows data for another experiment:

we computed expansion proofs for the induction-eliminated instance proofs and minimized them before giving them to the algorithm. Minimization here consists of taking a minimal subset of tautological instances. The line for the minimized instance proofs is very similar but slightly worse than the line for the non-minimized instance proofs. This shows that the regularity necessary for induction-reversal to succeed does not require instance proofs whose language is exactly the language of an induction grammar, but that their languages may also be a subset.

In many cases the computed grammar does not match the grammar of the original simple induction proof exactly. One relatively common situation occurring in 10 reconstructed proofs is that we can replace a quantified induction formula by a quantifier-free one.

This happens for example in `isaplanner.prop_06`, which proves $\forall x \forall y\, x - (x + y) = 0$ using $\forall y\, (x - (x + y) = 0)$ as the induction formula[2]. The induction grammar of the proof with the quantified induction formula is as follows, where z is a free variable in the proof:

$$\tau \to r_1(0 + \gamma) \mid r_2(\nu, \nu + \gamma) \mid r_3(\nu, \gamma) \mid r_4$$
$$\gamma \to \gamma \mid z$$

The induction proof found by the algorithm replaces the induction formula by a quantifier-free formula $x - (x + y) = 0$ where y is an eigenvariable of the end-sequent. On the level of the induction grammar, this corresponds to the elimination of the productions $\gamma \to \gamma \mid y$ (which are not very useful since the nonterminal γ will always expand to y):

$$\tau \to r_1(0 + y) \mid r_2(\nu, \nu + y) \mid r_3(\nu, y) \mid r_4$$

This experiment shows two things: on the one hand, we see that the algorithm can effective generate proofs with induction, as long as the input proofs are sufficiently regular. There are hence no obvious algorithmic deficiencies or implementation bugs. On the other hand, we also observe that there are qualitative differences in the proofs produced by induction-elimination and the proofs produced by automated theorem provers. Namely, that the proofs

[2]The binary operation $-$ denotes *truncating* subtraction, i.e. $x - y = \max\{x - y, 0\}$.

produced by the ATP often do not possess the regularity that we are interested in.

7.5 Case study: doubling

Let us now examine one of the examples in more detail, where the algorithm finds a candidate grammar but cannot solve the induced formula equation. The example is `prod/prop_01` from the TIP library, which shows that $\forall x\ d(x) = x + x$ where d is a recursively defined doubling function:

$$\begin{aligned}
&\forall x\ x + 0 = x, && (r_1)\\
&\forall x \forall y\ x + s(y) = s(x + y), && (r_2)\\
&d(0) = 0, && (r_3)\\
&\forall x\ d(s(x)) = s(s(d(x))) && (r_4)\\
&\vdash \forall x\ d(x) = x + x && (r_5)
\end{aligned}$$

To keep the size of the grammar and formula equation manageable, we put some of the formulas from the antecedent of the sequent into the background theory $T = \{d(0) = 0, d(s(x)) = s(s(d(x))), x + 0 = x\}$. This choice effectively ignores the instances of all formulas except (r_3), which contains interesting information about the induction.

Let us first consider how a "natural" simple induction proof of this sequent would look like. We might be inclined to prove $\forall x\ \forall y\ s(x) + y = s(x + y)$ as a lemma first, but a simple induction proof only contains one induction inference. Combining the goal with the lemma, an induction formula that works is $\forall \gamma\ (s(\gamma) + \nu = s(\gamma + \nu) \wedge d(\nu) = \nu + \nu)$. The quantifier instances in a simple induction proof with this induction formula can be described by the induction grammar with the following productions:

$$\begin{aligned}
&\tau \to r_2(s(\gamma), \nu) \mid r_2(\gamma, \nu) \mid r_2(s(\nu), \nu) \mid r_5\\
&\gamma \to \gamma \mid \nu \mid 0
\end{aligned}$$

This grammar induces the following formula equation:

$$\begin{aligned}
&\forall \gamma\, X(\alpha, 0, \gamma) \wedge (X(\alpha, \alpha, \alpha) \to d(\alpha) = \alpha + \alpha) \wedge \\
&\forall \nu \forall \gamma\, (X(\alpha, \nu, \gamma) \wedge X(\alpha, \nu, \nu) \wedge X(\alpha, \nu, 0) \wedge \\
&\quad s(\gamma) + s(\nu) = s(s(\gamma) + \nu) \wedge \gamma + s(\nu) = s(\gamma + \nu) \wedge \\
&\quad s(\nu) + s(\nu) = s(s(\nu) + \nu) \to X(\alpha, s(\nu), \gamma))
\end{aligned}$$

The grammars produced by the algorithm are smaller and different in an interesting way. First, let us look at the grammar computed for induction-eliminated instance proofs. Like in Section 7.4, algorithm finds a grammar with fewer productions:

$$\begin{aligned}
\tau &\to r_2(\gamma, \nu) \mid r_5 \\
\gamma &\to \gamma \mid \nu \mid \alpha
\end{aligned}$$

We do not know if the induced formula equation has a solution (modulo T) or not:

$$\begin{aligned}
\exists X \big(&\forall \gamma\, X(\alpha, 0, \gamma) \wedge (X(\alpha, \alpha, \alpha) \to d(\alpha) = \alpha + \alpha) \wedge \\
&\forall \nu \forall \gamma\, (X(\alpha, \nu, \gamma) \wedge X(\alpha, \nu, \nu) \wedge X(\alpha, \nu, \alpha) \wedge \\
&\gamma + s(\nu) = s(\gamma + \nu) \to X(\alpha, s(\nu), \gamma))\big)
\end{aligned}$$

Looking at the grammar computed for the instance proofs generated by the automated theorem prover, it is even simpler. It is so simple, it only consists of two productions:

$$\tau \to r_2(\alpha, \nu) \mid r_5$$

It is also qualitatively different: there is no occurrence of γ, so there would even be a simple induction proof with a *quantifier-free* induction formula, assuming that the induced formula equation is solvable. Alas, we do not know whether this FE is solvable either:

$$\begin{aligned}
\exists X \big(&X(\alpha, 0) \wedge (X(\alpha, \alpha) \to d(\alpha) = \alpha + \alpha) \wedge \\
&\forall \nu\, (X(\alpha, \nu) \wedge \alpha + s(\nu) = s(\alpha + \nu) \to X(\alpha, s(\nu)))\big)
\end{aligned}$$

It is instructive to compare the instance proofs from induction-elimination of Example 2.8.4 with the proofs produced by the automated theorem prover.

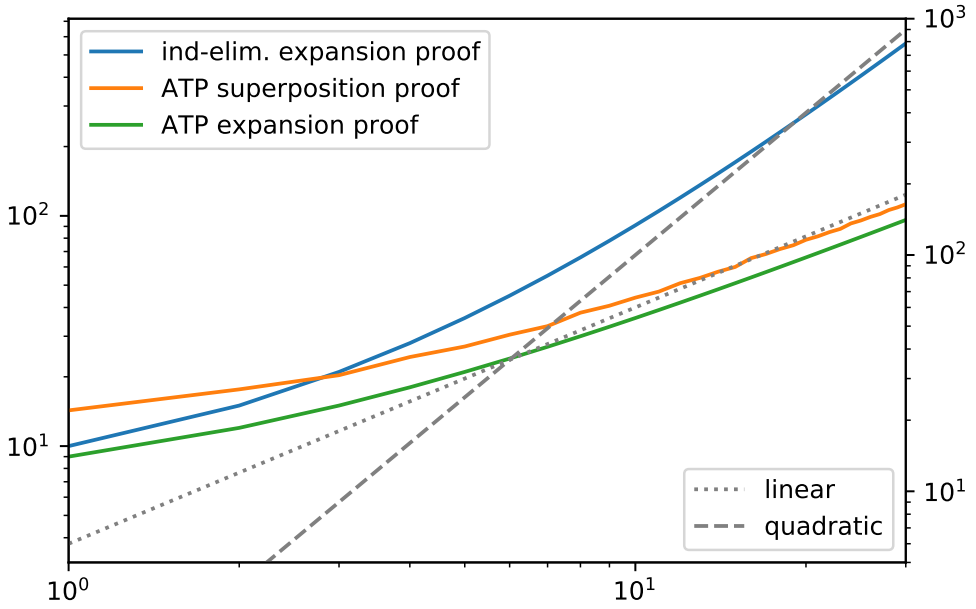


Figure 7.7: Proof size of the instance proofs from Section 7.5. The y-axis scale on the left-hand side is the expansion proof size, the scale on right-hand side is the superposition proof size. The two scales are different to make the comparison easier.

The straightforward way for a prover to prove the instance problems is to treat the assumptions (r_1)–(r_4) as a term rewriting system and to rewrite the conjectured equation $d(s^n(0)) = s^{2n}(0)$ into normal form:

$$d(s^n(0)) \overset{(r_4)}{\twoheadrightarrow} s^{2n}(d(0)) \overset{(r_3)}{\twoheadrightarrow} s^{2n}(0)$$
$$s^n(0) + s^n(0) \overset{(r_2)}{\twoheadrightarrow} s^n(s^n(0) + 0) \overset{(r_1)}{\twoheadrightarrow} s^{2n}(0)$$

Looking closely, this proof uses the instances $r_2(s^n(0), s^i(0))$ for $0 \leq i < n$. This set of instances corresponds to the formula $\alpha + s(v) = s(\alpha + v)$ in the formula equation. Note also that there are no "repeating" equations in this sequence that could give rise to an induction formula. By contrast, the proof obtained via induction-elimination contains instances for a much more

roundabout rewriting of the right-hand side, iterating the following part:

$$s^{i+1}(0) + s^{i+1}(0) \overset{(r_2)}{\twoheadrightarrow} s^{i+1}(s^{i+1}(0) + 0) \overset{(r_1)}{\twoheadrightarrow} s^{2i+2}(0)$$
$$\overset{(r_1)}{\twoheadleftarrow} s^{i+2}(s^{i}(0) + 0) \overset{(r_2)}{\twoheadleftarrow} s^{2}(s^{i}(0) + s^{i}(0))$$

Empirically, this difference is also reflected in the proof sizes. Figure 7.7 shows a double-logarithmic plot of the proof sizes of the instance proofs for $0 \leq n \leq 30$. The sequence of instance proofs obtained via induction- and cut-elimination from a simple induction proofs grows quadratically. The plot shows the size of the corresponding expansion proof: this size corresponds to the size of the instance language. The proof size is quadratic because the grammar contains the productions $\tau \to r_2(\gamma, \nu)$ and $\gamma \to \gamma \mid \nu$. The γ-production hence expands to all numerals greater than ν and $r_2(\gamma, \nu)$ expands to all $r_2(k, l)$ where $l < k \leq n$.

By contrast, the proofs that are found by the automated theorem prover are of linear size. (The built-in Escargot prover is used in this experiment.) Both the superposition proof directly produced by Escargot as well as the expansion proofs are of linear size. This is precisely because the superposition proof contains a rewriting sequence of the form described in this section.

8 Conclusion

We have presented a practical implementation of the approach to automated inductive theorem proving proposed in [31]. This required the development or extension of the algorithms presented in Chapters 4 to 6. On a theoretical side, this endeavour has lead to interesting questions. The study of the complexity of decision problems on grammars in Chapter 3 was motivated by the desire to find better algorithms producing covering grammars. The questions about formula equations leading to Theorem 5.4.3 were sparked by the necessity of algorithmically solving these formula equations. The practical evaluation of the approach lead to even more empirical questions about the nature of automatically generated proofs which we illuminated in Sections 5.5 and 7.5.

For various classes of grammars we have determined the computational complexity of the membership, containment, disjointness, equivalence, non-emptiness, and minimization problems in Chapter 3. The main open problem is the complexity of minimal cover as a decision problem, where we only know that is in NP. We were only able to show NP-completeness for the variant where the number of nonterminals is bounded. Surprisingly, the general case with an unbounded number of nonterminals is even open for the corresponding problem on regular grammars for words (Open Problem 3.7.2), even though that model has been studied much more extensively.

The reductions used for the hardness proofs generally use an unbounded number of symbols. However this is undesirable from a complexity point of view. In particular, the reductions for minimal cover require this unboundedness in an essential way to constrain the covering grammar. It remains as future work to study reductions which do not increase the number of symbols.

We have presented three algorithms in Chapter 4 that produce VTRATGs covering a given set of terms. Just as it is surprisingly hard to show that the TRATG-Cover problem is NP-complete, it is also hard to construct algorithms

that find *minimal* covering VTRATGs. Of the three algorithms, only the MaxSAT algorithm finds grammars that are guaranteed to be of minimal size.

The mathematical key insight behind the MaxSAT algorithm is that there is a polynomial time computable VTRATG which contains a minimal grammar as a subgrammar. This allows the polynomial reduction of the grammar compression problem to a MaxSAT problem for which highly efficient solvers are available.

Also from an application point of view, the MaxSAT algorithm is the most versatile and easy to extend. The only currently available algorithm to produce covering induction grammars is the one described in Section 4.4. The general theory of stable grammars behind underlying the MaxSAT algorithms makes such an extension straightforward and transparent.

In principle the MaxSAT algorithm could be adapted to any class of grammars that are closed under rewriting. It could also be extended to other notions of grammar size: [32] introduces the tree complexity measure, which precisely corresponds to the number of weak quantifier inferences in a proof (as opposed to the constant factor that we have had to introduce in Lemmas 2.7.4 and 2.8.2). The MaxSAT-reduction of the minimization problem itself in Section 4.3.5 could be extended to other classes of grammars or automata as well, for example to find tree automata with a fixed number of states and minimal number of transitions that accept a given finite set of terms.

We have discussed two algorithms to solve formula equations in Chapter 5. Both of them are heuristic: they are not complete, in the sense that they are not guaranteed to find a solution if one exists. As we have seen in Section 7.3, the two algorithms seem to be complementary in practice with the interpolation-based one having a small lead. Half of the formula equations solved by one are not solved by the other and vice versa. The success of the interpolation-based method modeled after the Duality algorithm for constrained Horn clauses suggests that the extension of CHC solvers to the theory of equality with uninterpreted function symbols is an interesting application.

On a theoretical level we could improve on a result of [31] on the existence of tautological induction grammars whose induced formula equation is not solvable. Theorem 5.4.3 both reduces the nonterminals in the grammar as well as extends it to strong background theories. It remains as future work to see

in how far this technique allows us to extend the result on the undecidability of the solvability of induction grammars (Theorem 5.4.2) in a similar way.

The transformation from resolution proofs to expansion proofs described in Chapter 6 has been used as the default method to generate expansion proofs from resolution proofs in GAPT since version 2.2. Using it, GAPT can effectively interface with six different external resolution-based provers, including SPASS and Vampire with their splitting rules. The modular integration of structural clausification makes it possible to reuse it for non-resolution provers as well: the interface to the connection-based prover LeanCoP [76, 82] in GAPT uses the same code for clausification and definition elimination.

A noteworthy limitation of the clausification in Chapter 6 is that it can expand each definition only in a single polarity. Lifting this restriction would produce cuts in the definition elimination phase. However, these cuts would contain Skolem nodes and cannot be eliminated directly. In the equality-free case, there is a reliable deskolemization procedure [7] which can be used as a preprocessing step to enable cut-elimination. Such a procedure is yet missing for proofs with equational reasoning. Reliable deskolemization would also enable a straightforward adaptation of the clausification produced by external provers using proof replay.

Many of the techniques used come from higher-order logic, both Andrew's calculus $\mathcal{R}$ and expansion trees originate in that setting. It seems only natural to extend this transformation to higher-order logic, and there seem to be no immediate obstacles except for the built-in equational inferences. But these could be straightforwardly translated to explicit Leibniz equality.

Many clausification procedures perform propositional simplification rules as a pre-processing step, for example rewriting $\varphi \wedge \top \mapsto \varphi$ or converting to negation normal form. These simplifications could be helpful in the clausification here as well, since they potentially enable sharing of subformula definitions and Skolem functions. These could be supported by adding new inference rules to the resolution calculus and adapting the expansion tree extraction in the natural way.

For simplicity the SMT refutations in proofs using Avatar are converted to resolution proofs first. However there is no fundamental reason why we need to perform this costly conversion, since the SMT refutation is purely

ground and is essentially discarded in this translation. Adding a new rule to represent this part of the proof in a single inference would deliver even greater performance.

Finally, we have evaluated the implementation of the whole approach in Chapter 7. The empirical results show that there is a clear difference in the success of the approach if we compare proofs obtained from induction-elimination and the proofs found by an automated theorem prover. In order to successfully find a proof with induction, the instance proofs must have a certain form of regularity.

One potential way to obtain more regular families of proofs is to modify the search procedure of the automated theorem prover, for example to use an outermost-first strategy for rewriting, which sometimes seems beneficial in our experience. Another option would be to perform a heuristic post-processing to regularize the instance proofs.

Instead of finding more regular families of instance proofs, we could also extend the grammars so that they can cover more irregular families. The induction grammars in this paper need to cover the term sets for the instance proofs exactly, without taking the background theory into account. We could extend the grammar generation algorithm so that it generates grammars that cover the input term sets modulo the background theory—that is, where every term in the input term set is T-equivalent to a term in the generated language.

There are several configuration options that are currently set manually or where we simply try all possibilities, such as the number of quantifiers in the induction formula (i.e., how long is $\overline{y}$ and what types do the nonterminals have) or the equational theory. For quantitative options such as the number of quantifiers, heuristics and portfolio modes could be used in a production implementation.

These empirical results also shed light on the nature of cut- and induction-elimination. A priori we might expect that it actually eliminates all traces of the cuts and inductions in the original proof. But what we have seen in the empirical results is that there is observable and significant evidence left behind by induction-elimination. And these are not just faint traces: we can even get back a proof that is closely related to the original one on the level of quantifier inferences. This phenomenon—that there is interesting structure in cut-free

proofs—is typically not studied in proof theory. Usually the focus lies on the size increase of induction- or cut-elimination.

The related concept of analyticity is usually formulated as the subformula property [88]: in first-order logic this means that every formula occurring in the proof is a substitution instance of the formula to be proven. Proofs with cut or induction are in general non-analytic, as they contain new induction invariants or cut formulas.

However this view of analyticity as being the subformula property ignores all structure contained on the term level. What we have seen is that induction-elimination shifts the structure from the non-analytic induction invariant in the formulas to a certain kind of regularity in the terms in the quantifier inferences. And there remains enough structure on the term level to get back a proof with induction.

In infinitary proofs cut-elimination also leaves behind regularity, albeit in a different form. Applying cut-elimination to infinitary derivations corresponding to proofs in PA results in proofs with rank bounded by ϵ_0 [95].

We can observe a related phenomenon in mathematical logic with a more direct reflection of formulas as term-level structures: for example the set theory ZFC is not finitely axiomatizable due to the axiom scheme of replacement, which is parameterized by a formula. By encoding this formula parameter as a term we get the finitely axiomatized conservative extension NBG [74]. The theories PA and ACA_0 are related in a similar way. These phenomena differ from our observations as the induction formula is explicitly encoded as a term, whereas we investigate the incidental regularity in the quantifier instance terms, which implicitly indicates the provenance of induction-elimination.

It would also be interesting to find more direct characterizations of regularity on the level of formal languages that give insight into the features of languages which cause irregularity, along the lines of Myhill-Nerode [72, 73] for regular word languages and regular tree languages [23]. Developing similar characterizations for regularity in TRATGs and induction grammars could help illuminate the phenomenon of regularity in inductive proofs.

Bibliography

[1] Wilhelm Ackermann. "Untersuchungen über das Eliminationsproblem der mathematischen Logik". In: *Mathematische Annalen* 110.1 (1935), pp. 390–413.

[2] Bahareh Afshari, Stefan Hetzl, and Graham E. Leigh. "Herbrand's theorem as higher order recursion". In: *Annals of Pure and Applied Logic* 171.6 (2020), p. 102792.

[3] Brian Alspach, Peter Eades, and Gordon Rose. "A lower-bound for the number of productions required for a certain class of languages". In: *Discrete Applied Mathematics* 6.2 (1983), pp. 109–115.

[4] Peter B. Andrews. "Resolution in Type Theory". In: *Journal of Symbolic Logic* 36.3 (1971), pp. 414–432.

[5] Josep Argelich, Chu Min Li, Felip Manya, and Jordi Planes. "The First and Second Max-SAT Evaluations." In: *Journal on Satisfiability, Boolean Modeling and Computation* 4.2-4 (2008), pp. 251–278.

[6] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. "Complexity of finding embeddings in a k-tree". In: *SIAM Journal on Algebraic Discrete Methods* 8.2 (1987), pp. 277–284.

[7] Matthias Baaz, Stefan Hetzl, and Daniel Weller. "On the complexity of proof deskolemization". In: *Journal of Symbolic Logic* 77.2 (2012), pp. 669–686.

[8] Matthias Baaz and Alexander Leitsch. "Cut-elimination and Redundancy-elimination by Resolution". In: *Journal of Symbolic Computation* 29.2 (2000), pp. 149–176.

[9] Leo Bachmair and Harald Ganzinger. “Resolution Theorem Proving”. In: *Handbook of Automated Reasoning (Volume 1)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 19–99.

[10] Umberto Bertelé and Francesco Brioschi. *Nonserial Dynamic Programming*. 1972.

[11] Richard Bird and Oege de Moor. *Algebra of Programming*. 1997.

[12] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. “Horn clause solvers for program verification”. In: *Fields of Logic and Computation II*. Springer, 2015, pp. 24–51.

[13] Walter Bucher. “A Note on a Problem in the Theory of Grammatical Complexity”. In: *Theoretical Computer Science* 14 (1981), pp. 337–344.

[14] Walter Bucher, Hermann A. Maurer, and Karel Culik II. “Context-Free Complexity of Finite Languages”. In: *Theoretical Computer Science* 28 (1984), pp. 277–285.

[15] Walter Bucher, Hermann A. Maurer, Karel Culik II, and Detlef Wotschke. “Concise Description of Finite Languages”. In: *Theoretical Computer Science* 14 (1981), pp. 227–246.

[16] Samuel R. Buss. “On Herbrand’s Theorem”. In: *Logic and Computational Complexity*. Vol. 960. Lecture Notes in Computer Science. Springer, 1995, pp. 195–209.

[17] Cezar Câmpeanu, Nicolae Santean, and Sheng Yu. “Minimal Cover-Automata for Finite Languages”. In: *Third International Workshop on Implementing Automata (WIA’98)*. Ed. by Jean-Marc Champarnaud, Denis Maurel, and Djelloul Ziadi. Vol. 1660. Lecture Notes in Computer Science. Springer, 1999, pp. 43–56.

[18] Cezar Câmpeanu, Nicolae Santean, and Sheng Yu. “Minimal cover-automata for finite languages”. In: *Theoretical Computer Science* 267.1-2 (2001), pp. 3–16.

[19] Katrin Casel, Henning Fernau, Serge Gaspers, Benjamin Gras, and Markus L. Schmid. “On the Complexity of Grammar-Based Compression over Fixed Alphabets”. In: *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Ed. by Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi. Vol. 55. Leibniz International Proceedings in Informatics (LIPIcs). 2016, 122:1–122:14.

[20] *CHC-COMP: Constrained Horn Clause competition*. 2018–2019. URL: `https://chc-comp.github.io/`.

[21] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. “TIP: Tons of Inductive Problems”. In: *Conferences on Intelligent Computer Mathematics*. Ed. by Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge. 2015, pp. 333–337.

[22] Koen Claessen, Dan Rosén, Moa Johansson, and Nicholas Smallbone. “Automating Inductive Proofs using Theory Exploration”. In: *24th International Conference on Automated Deduction*. 15. 2013, pp. 392–406.

[23] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007.

[24] William Craig. “Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory”. In: *The Journal of Symbolic Logic* 22.3 (1957), pp. 269–285.

[25] Simon Cruanes. “Superposition with Structural Induction”. In: *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings*. Ed. by Clare Dixon and Marcelo Finger. Vol. 10483. Lecture Notes in Computer Science. Springer, 2017, pp. 172–188.

[26] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. “A New Deconstructive Logic: Linear Logic”. In: *Journal of Symbolic Logic* 62.3 (1997), pp. 755–807.

[27] Nachum Dershowitz and David A. Plaisted. “Rewriting”. In: *Handbook of Automated Reasoning*. Vol. 1. 2001, pp. 535–610.

[28] Lucas Dixon. “A Proof Planning Framework for Isabelle”. PhD thesis. University of Edinburgh, 2005.

[29] Sebastian Eberhard, Gabriel Ebner, and Stefan Hetzl. “Algorithmic Compression of Finite Tree Languages by Rigid Acyclic Grammars”. In: *ACM Transactions on Computational Logic* 18.4 (2017), 26:1–26:20.

[30] Sebastian Eberhard, Gabriel Ebner, and Stefan Hetzl. “Complexity of Decision Problems on Totally Rigid Acyclic Tree Grammars”. In: *Developments in Language Theory - 22nd International Conference, DLT*. Ed. by Mizuho Hoshi and Shinnosuke Seki. Vol. 11088. Lecture Notes in Computer Science. Springer, 2018, pp. 291–303.

[31] Sebastian Eberhard and Stefan Hetzl. “Inductive theorem proving based on tree grammars”. In: *Annals of Pure and Applied Logic* 166.6 (2015), pp. 665–700.

[32] Sebastian Eberhard and Stefan Hetzl. “On the compressibility of finite languages and formal proofs”. In: *Information and Computation* 259 (2018), pp. 191–213.

[33] Sebastian Eberhard, Stefan Hetzl, and Daniel Weller. “Boolean unification with predicates”. In: *Journal of Logic and Computation* 27.1 (2017), pp. 109–128.

[34] Gabriel Ebner. “Fast Cut-Elimination using Proof Terms: An Empirical Study”. In: *Proceedings Seventh International Workshop on Classical Logic and Computation, CL&C 2018, Oxford (UK), 7th of July 2018*. Ed. by Stefano Berardi and Alexandre Miquel. Vol. 281. EPTCS. 2018, pp. 24–38.

[35] Gabriel Ebner. “Herbrand Constructivization for Automated Intuitionistic Theorem Proving”. In: *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings*. Ed. by Serenella Cerrito and Andrei Popescu. Vol. 11714. Lecture Notes in Computer Science. Springer, 2019, pp. 355–373.

[36] Gabriel Ebner, Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. “On the generation of quantified lemmas”. In: *Journal of Automated Reasoning* (2018), pp. 1–32.

[37] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. “System Description: GAPT 2.0”. In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. Lecture Notes in Computer Science. Springer, 2016, pp. 293–301.

[38] Grigory Fedyukovich, Samuel J Kaufman, and Rastislav Bodík. “Sampling invariants from frequency distributions”. In: *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2017, pp. 100–107.

[39] Andreas Halkjær From. “Spin – Superposition with Structural Induction”. Internship report. Technische Universität Wien, 2019.

[40] Adrià Gascón, Guillem Godoy, and Florent Jacquemard. “Closure of Tree Automata Languages under Innermost Rewriting”. In: *Electronic Notes in Theoretical Computer Science* 237 (2009). Proceedings of the 8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008), pp. 23–38.

[41] Gerhard Gentzen. “Die Widerspruchsfreiheit der reinen Zahlentheorie”. In: *Mathematische Annalen* 112 (1936), pp. 493–565.

[42] Gerhard Gentzen. “Neue Fassung des Widerspruchsfreiheitsbeweises für die reine Zahlentheorie”. In: *Forschungen zur Logik und zur Grundlegung der exakten Wissenschaften* 4 (1938), pp. 19–44.

[43] Gerhard Gentzen. “Untersuchungen über das logische Schließen I”. In: *Mathematische Zeitschrift* 39.1 (1935), pp. 176–210.

[44] Sergey Grebenshchikov, Nuno P Lopes, Corneliu Popeea, and Andrey Rybalchenko. “Synthesizing software verifiers from proof rules”. In: *ACM SIGPLAN Notices*. Vol. 47. 6. ACM. 2012, pp. 405–416.

[45] Jacques Herbrand. “Recherches sur la théorie de la démonstration”. PhD thesis. Université de Paris, 1930.

[46] Stefan Hetzl. "Applying Tree Languages in Proof Theory". In: *Language and Automata Theory and Applications*. Ed. by Adrian-Horia Dediu and Carlos Martín-Vide. Vol. 7183. Lecture Notes in Computer Science. Springer, 2012, pp. 301–312.

[47] Stefan Hetzl. "Characteristic Clause Sets and Proof Transformations". PhD thesis. Technische Universität Wien, 2007.

[48] Stefan Hetzl, Alexander Leitsch, Giselle Reis, Janos Tapolczai, and Daniel Weller. "Introducing Quantified Cuts in Logic with Equality". In: *International Joint Conference on Automated Reasonin (IJCAR)*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Vol. 8562. Lecture Notes in Computer Science. Springer, 2014, pp. 240–254.

[49] Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. *Algorithmic introduction of quantified cuts*. (This preprint contains additional material on the delta-table algorithm not present in the conference version from IJCAR.) 2014. URL: https://arxiv.org/abs/1401.4330.

[50] Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. "Algorithmic introduction of quantified cuts". In: *Theoretical Computer Science* 549 (2014), pp. 1–16.

[51] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. "Towards Algorithmic Cut-Introduction". In: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR-18)*. Vol. 7180. Lecture Notes in Computer Science. Springer, 2012, pp. 228–242.

[52] Stefan Hetzl, Tomer Libal, Martin Riener, and Mikheil Rukhaia. "Understanding Resolution Proofs through Herbrand's Theorem". In: *22nd International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX*. 2013, pp. 157–171.

[53] Stefan Hetzl and Lutz Straßburger. "Herbrand-Confluence". In: *Logical Methods in Computer Science* 9.4 (2013).

[54] Stefan Hetzl and Daniel Weller. *Expansion Trees with Cut*. 2013. URL: https://arxiv.org/abs/1308.0428.

[55] Stefan Hetzl and Sebastian Zivota. “Tree Grammars for the Elimination of Non-prenex Cuts”. In: *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*. Ed. by Stephan Kreutzer. Vol. 41. LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 110–127.

[56] Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. “μZ–an efficient engine for fixed points with constraints”. In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 457–462.

[57] Hossein Hojjat and Philipp Rümmer. “The ELDARICA Horn Solver”. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2018, pp. 1–7.

[58] Guoxiang Huang. “Constructing Craig Interpolation Formulas”. In: *Computing and Combinatorics, First Annual International Conference, COCOON ’95, Xi’an, China, August 24-26, 1995, Proceedings*. Ed. by Ding-Zhu Du and Ming Li. Vol. 959. Lecture Notes in Computer Science. Springer, 1995, pp. 181–190.

[59] Harry B. Hunt III, Daniel J. Rosenkrantz, and Thomas G. Szymanski. “On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages”. In: *Journal of Computer and System Sciences* 12.2 (1976), pp. 222–268.

[60] Florent Jacquemard, Francis Klay, and Camille Vacher. “Rigid tree automata”. In: *Language and Automata Theory and Applications (LATA) 2009*. Ed. by Adrian Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide. Vol. 5457. Lecture Notes in Computer Science. Springer, 2009, pp. 446–457.

[61] Florent Jacquemard, Francis Klay, and Camille Vacher. “Rigid Tree Automata and Applications”. In: *Information and Computation* 209.3 (2011), pp. 486–512.

[62] Bishoksan Kafle, John P Gallagher, and José F Morales. “RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata”. In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 261–268.

[63] John C. Kieffer and En-hui Yang. "Grammar Based Codes: A New Class of Universal Lossless Source Codes". In: *IEEE Transactions on Information Theory* 46.3 (2000), pp. 737–754.

[64] N. Jesper Larsson and Alistair Moffat. "Offline Dictionary-Based Compression". In: *Data Compression Conference (DCC 1999)*. IEEE Computer Society, 1999, pp. 296–305.

[65] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. "XML tree structure compression using RePair". In: *Information Systems* 38.8 (2013), pp. 1150–1167.

[66] Horst Luckhardt. "Herbrand-Analysen zweier Beweise des Satzes von Roth: Polynomiale Anzahlschranken". In: *The Journal of Symbolic Logic* 54.1 (1989), pp. 234–263.

[67] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. "Open-WBO: A Modular MaxSAT Solver," in: *Theory and Applications of Satisfiability Testing - SAT 2014*. 2014, pp. 438–445.

[68] Kenneth L. McMillan. "An interpolating theorem prover". In: *Theoretical Computer Science* 345.1 (2005), pp. 101–121.

[69] Kenneth L. McMillan and Andrey Rybalchenko. *Solving Constrained Horn Clauses using Interpolation*. Tech. rep. MSR-TR-2013-6. Microsoft Research, 2013.

[70] Albert R. Meyer and Larry J. Stockmeyer. "The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space". In: *13th Annual Symposium on Switching and Automata Theory, College Park, Maryland, USA, October 25-27, 1972*. IEEE Computer Society, 1972, pp. 125–129.

[71] Dale A. Miller. "A Compact Representation of Proofs". In: *Studia Logica* 46.4 (1987), pp. 347–370.

[72] John Myhill. *Finite automata and the representation of events*. Tech. rep. WADD TR-57-624. Wright Patterson AFB, Ohio, 1957.

[73] Anil Nerode. "Linear Automaton Transformations". In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544.

[74] John von Neumann. “Eine Axiomatisierung der Mengenlehre”. In: *Journal für die reine und angewandte Mathematik* 154 (1925), pp. 219–240.

[75] Craig G. Nevill-Manning and Ian H. Witten. “Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm”. In: *Journal of Artificial Intelligence Research* 7 (1997), pp. 67–82.

[76] Jens Otten. “leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic”. In: *4th International Joint Conference on Automated Reasoning, IJCAR*. 2008, pp. 283–291.

[77] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[78] Frank Pfenning. “Analytic and Non-analytic Proofs”. In: *7^{th} International Conference on Automated Deduction, CADE*. Ed. by Robert E. Shostak. Vol. 170. Lecture Notes in Computer Science. Springer, 1984, pp. 394–413.

[79] Gordon D. Plotkin. “A further note on inductive generalization”. In: *Machine Intelligence* 6 (1971), pp. 101–124.

[80] Gordon D. Plotkin. “A note on inductive generalization”. In: *Machine Intelligence* 5.1 (1970), pp. 153–163.

[81] Emil L Post. “A variant of a recursively unsolvable problem”. In: *Bulletin of the American Mathematical Society* 52.4 (1946), pp. 264–268.

[82] Giselle Reis. “Importing SMT and Connection proofs as expansion trees”. In: *Fourth Workshop on Proof eXchange for Theorem Proving, PxTP*. 2015, pp. 3–10.

[83] John C. Reynolds. “Transformational systems and the algebraic structure of atomic formulas”. In: *Machine Intelligence* 5.1 (1970), pp. 135–151.

[84] Neil Robertson and Paul D. Seymour. “Graph Minors. II. Algorithmic Aspects of Tree-Width”. In: *Journal of Algorithms* 7.3 (1986), pp. 309–322.

[85] Jan J. M. M. Rutten. “Relators and Metric Bisimulations”. In: *Electronic Notes in Theoretical Computer Science* 11 (1998), pp. 252–258.

[86] Sherif Sakr. "XML compression techniques: A survey and comparison". In: *Journal of Computer and System Sciences* 75.5 (2009), pp. 303–322.

[87] Stephan Schulz. "System Description: E 1.8". In: *19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. LNCS. Springer, 2013.

[88] Raymond M. Smullyan. *First-order logic*. 1968.

[89] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. "Zeno: An Automated Prover for Properties of Recursive Data Structures". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012*. Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 407–421.

[90] Larry J. Stockmeyer and Albert R. Meyer. "Word Problems Requiring Exponential Time: Preliminary Report". In: *Symposium on Theory of Computing*. Ed. by Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong. ACM, 1973, pp. 1–9.

[91] James A. Storer and Thomas G. Szymanski. "Data Compression via Textual Substitution". In: *Journal of the ACM* 29.4 (1982), pp. 928–951.

[92] James A. Storer and Thomas G. Szymanski. "The Macro Model for Data Compression (Extended Abstract)". In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC '78)*. New York, NY, USA: ACM, 1978, pp. 30–39.

[93] G. Sutcliffe. "The TPTP World - Infrastructure for Automated Reasoning". In: *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Ed. by E. Clarke and A. Voronkov. Lecture Notes in Artificial Intelligence 6355. Springer-Verlag, 2010, pp. 1–12.

[94] Geoff Sutcliffe. "The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 337–362.

[95] William Walker Tait. “Normal derivability in classical logic”. In: *The syntax and semantics of infinitary languages*. Vol. 72. Springer, 1968, pp. 204–236.

[96] O. Tange. “GNU Parallel - The Command-Line Power Tool”. In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47.

[97] Grigori S. Tseitin. “On the complexity of derivation in propositional calculus”. In: *Automation of Reasoning: Classical Papers in Computational Logic*. Vol. 2. Springer, 1983, pp. 466–483.

[98] Zsolt Tuza. “On the context-free production complexity of finite languages”. In: *Discrete Applied Mathematics* 18.3 (1987), pp. 293–304.

[99] Andrei Voronkov. “AVATAR: The Architecture for First-Order Theorem Provers”. In: *26^{th} International Conference on Computer Aided Verification, CAV 2014*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 696–710.

[100] Christoph Weidenbach. “Combining Superposition, Sorts and Splitting”. In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Vol. 2. 2001. Chap. 27, pp. 1965–2013.

[101] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. “SPASS Version 3.5”. In: *22nd International Conference on Automated Deduction (CADE)*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 140–145.

[102] Christoph Wernhard. “The Boolean Solution Problem from the Perspective of Predicate Logic”. In: *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings*. Ed. by Clare Dixon and Marcelo Finger. Vol. 10483. Lecture Notes in Computer Science. Springer, 2017, pp. 333–350.

[103] He Zhu, Stephen Magill, and Suresh Jagannathan. “A data-driven CHC solver”. In: *ACM SIGPLAN Notices*. Vol. 53. 4. ACM. 2018, pp. 707–721.

Curriculum vitae

Gabriel Ebner
gebner@gebner.org
https://gebner.org

Academic background

MSc in mathematics 2015
Technische Universität Wien, Austria
Topic: Finding loop invariants using tree grammars
Advisor: Stefan Hetzl

BSc in mathematics 2013
Technische Universität Wien, Austria
Topic: The Solovay model
Advisor: Martin Goldstern

Professional experience

Vrije Universiteit Amsterdam, The Netherlands 2019–2021
Faculty of Science, Theoretical Computer Science
Researcher

Technische Universität Wien, Austria 2014–2019
Institute for Discrete Mathematics and Geometry
Project assistant

Catalysts GmbH, Austria 2011, 2013–2015
Software engineer

Short research stays

Carnegie Mellon University 2016
Host: Jeremy Avigad

Journal articles

On the generation of quantified lemmas 2019
(with S. Hetzl, A. Leitsch, G. Reis, and D. Weller)
Journal of Automated Reasoning 63(1), 2019

Algorithmic compression of finite tree languages by rigid acyclic grammars 2017
(with S. Eberhard and S. Hetzl)
ACM Transactions on Computational Logic 18(4), 2017

Conference papers

Maintaining a library of formal mathematics 2020
(with F. van Doorn and R. Y. Lewis)
13th Conference on Intelligent Computer Mathematics (CICM 2020),
Benzmüller C. and Miller B. (eds.)

Herbrand constructivization for automated intuitionistic theorem proving 2019
28th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2019),
S. Cerrito and A. Popescu (eds.)

Complexity of decision problems on totally rigid acyclic tree grammars 2018
22nd International Conference on Developments in Language Theory (DLT 2018),
Hoshi M. and Seki S. (eds.)

A metaprogramming framework for formal verification 2017
(with S. Ullrich, J. Roesch, J. Avigad, and L. de Moura)
22nd ACM SIGPLAN International Conference on Functional Programming (ICFP 2017)

System description: GAPT 2.0 2016
(with S. Hetzl, G. Reis, M. Riener, S. Wolfsteiner, and S. Zivota)
8th International Joint Conference on Automated Reasoning (IJCAR 2016), N. Olivetti and A. Tiwari (eds.)

Peer-reviewed workshop papers

Fast cut-elimination using proof terms: an empirical study 2018
7th International Workshop on Classical Logic and Computation (CL&C 2018), S. Berardi and A. Miquel (eds.)

Efficient translation of sequent calculus proofs into natural deduction proofs 2018
(with M. Schlaipfer)
6th Workshop on Practical Aspects of Automated Reasoning (PAAR 2018), B. Konev, J. Urban, and P. Rümmer (eds.)

Membership in program committees

13th NASA Formal Methods Symposium (NFM) 2021

7th Workshop on Practical Aspects of Automated Reasoning (PAAR) 2020

22nd Symposium on Practical Aspects of Declarative Languages (PADL) 2020

Funding

Cloud computing for Lean's mathlib 2020
Microsoft Research Azure grant
Joint PI with R. Y. Lewis

Teaching

Logical Verification 2020
MSc course, Vrije Universiteit Amsterdam
Lecture together with J. C. Blanchette and R. Y. Lewis

Algebra and Discrete Mathematics 2015
Analysis 2012
Analysis 2 2012
Algebra and Discrete Mathematics 2011
Exercise part of BSc courses for computer science students,
Technische Universität Wien